Partial Order Reduction for Security Protocols*

David Baelde, Stéphanie Delaune, and Lucca Hirschi

LSV, ENS Cachan & CNRS, France
{baelde,delaune,hirschi}@lsv.ens-cachan.fr

Abstract

Security protocols are concurrent processes that communicate using cryptography with the aim of achieving various security properties. Recent work on their formal verification has brought procedures and tools for deciding trace equivalence properties (*e.g.,* anonymity, unlinkability, vote secrecy) for a bounded number of sessions. However, these procedures are based on a naive symbolic exploration of all traces of the considered processes which, unsurprisingly, greatly limits the scalability and practical impact of the verification tools.

In this paper, we mitigate this difficulty by developing partial order reduction techniques for the verification of security protocols. We provide reduced transition systems that optimally eliminate redundant traces, and which are adequate for model-checking trace equivalence properties of protocols by means of symbolic execution. We have implemented our reductions in the tool Apte, and demonstrated that it achieves the expected speedup on various protocols.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases cryptographic protocols, verification, process algebra, trace equivalence

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2015.497

1 Introduction

Security protocols are concurrent processes that use various cryptographic primitives in order to achieve security properties such as secrecy, authentication, anonymity, unlinkability, *etc.* They involve a high level of concurrency and are difficult to analyse by hand. Actually, many protocols have been shown to be flawed several years after their publication (and deployment). This has led to a flurry of research on formal verification of protocols.

A successful way of representing protocols is to use variants of the π-calculus, whose labelled transition systems naturally express how a protocol may interact with a (potentially malicious) environment whose knowledge increases as more messages are exchanged over the network. Some security properties (*e.g.,* secrecy, authentication) are then described as reachability properties, while others (*e.g.,* unlinkability, anonymity) are expressed as trace equivalence properties. In order to decide such properties, a reasonable assumption is to bound the number of protocol sessions, thereby limiting the length of execution traces. Even under this assumption, infinitely many traces remain, since each input may be fed infinitely many different messages. However, symbolic execution and dedicated constraint solving procedures have been devised to provide decision procedures for reachability [18, 12] and, more recently, equivalence properties [23, 9]. Unfortunately, the resulting tools, especially those for checking equivalence (*e.g.,* Apte [8], Spec [22]), have a very limited practical impact because they scale very badly. This is not surprising since they treat concurrency in a very naive way, exploring all possible symbolic interleavings of concurrent actions.

* This work has been partially supported by the project JCJC VIP ANR-11-JS02-006, and the Inria large scale initiative CAPPRIS.

26th International Conference on Concurrency Theory (CONCUR 2015).
Editors: Luca Aceto and David de Frutos Escrig; pp. 497–510

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Contributions. We develop partial order reduction (POR) techniques for trace equivalence checking of security protocols. Our main challenge is to do it in a way that is compatible with symbolic execution: we should provide a reduction that is effective when messages remain unknown, but leverages information about messages when it is inferred by the constraint solver. We achieve this by refining interleaving semantics in two steps, gradually eliminating redundant traces. The first refinement, called *compression*, uses the notion of polarity [2] to impose a simple strategy on traces. It does not rely on data analysis at all and can easily be used as a replacement for the usual semantics in verification algorithms. The second one, called *reduction*, takes data into account and achieves optimality in eliminating redundant traces. In practice, the reduction step can be implemented in an approximated fashion, through an extension of constraint resolution procedures. We have done so in the tool Apte, showing that our theoretical results do translate to significant practical optimisations.

Related work. The theory of partial order reduction is well developed in the context of reactive systems verification (*e.g.,* [21, 6, 16]). However, as pointed out by E. Clarke *et al.* in [11], POR techniques from traditional model-checking cannot be directly applied in the context of security protocol verification. Indeed, the application to security requires one to keep track of the knowledge of the attacker, and to refer to this knowledge in a meaningful way (in particular to know which messages can be forged at some point to feed some input). Furthermore, security protocol analysis does not rely on the internal reduction of a protocol, but has to consider arbitrary execution contexts (representing interactions with arbitrary, active attackers). Thus, any input may depend on any output, since the attacker has the liberty of constructing arbitrary messages from past outputs. This results in a dependency relation which is *a priori* very large, rendering traditional POR arguments suboptimal, and calling for domain-specific techniques.

In order to achieve our goal of improving existing tools, our techniques are designed to integrate nicely with symbolic execution. This is necessary to precisely deal with infinite, structured data, without considering an *a priori* fixed and finite set of messages, as is the case in several earlier works, *e.g.,* [11, 13]. In this task, we get some inspiration from Mödersheim *et al.* [20]. While their reduction is very limited, it brings some key insight on how POR may be combined with symbolic execution in the context of security protocols verification. All of the papers mentioned above only consider reachability properties, while we develop an approach which is adequate for model-checking trace equivalence properties. In earlier work [4] we have combined the idea of [20] with more powerful partial order reduction, in a way that is compatible with trace equivalence checking. This settled the general ideas behind the present paper, but only covered the very restrictive class of *simple processes* (parallel processes communicate on distinct channels, and replication and nested parallel composition are not allowed). Actually, we made heavy use of specific properties of those simple processes to define our reductions and prove them correct. The present work also brings a solid implementation in the tool Apte [8].

Outline. We consider in Section 2 a rich process algebra for representing security protocols. It supports arbitrary cryptographic primitives, and even includes a replication operator suitable for modelling unbounded numbers of sessions. Thus, we are not restricted to a particular fragment for which a decision procedure exists, but show the full scope of our theoretical results. We give in Section 3 an *annotated* semantics that will facilitate the following technical developments. We then define our *compressed* semantics in Section 4 and the *reduced* semantics in Section 5. In both sections, we first restrict the transition

system, then show that the restriction is adequate for checking trace equivalence under some action-determinism condition. We finally discuss how these results can be lifted to the symbolic setting in Section 6. Specifically, we describe how we have implemented our techniques in Apte, and we present experimental results showing that the optimisations are fully effective in practice.

Due to lack of space, the reader is referred to the companion technical report [5] for the missing proofs and additional details. In particular, a comparison of this work with the extensive literature about POR can be found in [5, §7].

2 Model for security protocols

In this section we introduce our process algebra, which is a variant of the applied π-calculus [1] that has been designed with the aim of modelling cryptographic protocols. Processes can exchange complex messages, represented by terms quotiented by some equational theory.

One of the key difficulties in the applied π-calculus is that it models the knowledge of the environment, seen as an attacker who listens to network communication and may also inject messages. One has to make a distinction between the content of a message (sent by the environment) and the way the message has been created (from knowledge available to the environment). While the distinction between messages and recipes came from security applications, it may be of much broader interest, as it gives a precise, intentional content to labelled transitions that we exploit to analyse data dependencies.

We study a process algebra that may seem quite restrictive: we forbid internal communication and private channels. However, this is reasonable when studying security protocols faced with the usual omnipotent attacker. In such a setting, we end up considering the worst-case scenario where any communication has to be made via the environment.

2.1 Syntax

We assume a number of disjoint and infinite sets: a set $\mathcal{C}$ of *channels*, whose elements are denoted by a, b, c; a set $\mathcal{N}$ of *private names* or *nonces*, denoted by n or k; a set $\mathcal{X}$ of *variables*, denoted by x, y, z as usual; and a set $\mathcal{W}$ of *handles*, denoted by w and used for referring to previously output terms. Next, we consider a signature Σ consisting of a finite set of function symbols together with their arity. Terms over S, written $\mathcal{T}(S)$, are inductively generated from S and function symbols from Σ. When $S \subseteq \mathcal{N}$, elements of $\mathcal{T}(S)$ are called *messages*. When $S \subseteq \mathcal{W}$, they are called *recipes* and written M, N. Intuitively, recipes express how a message has been derived by the environment from the messages obtained so far. Finally, we consider an equational theory E over terms to assign a meaning to function symbols in Σ.

▶ **Example 1.** Let $\Sigma = \{\mathsf{enc}/2, \mathsf{dec}/2, \mathsf{h}/1\}$ and E be the equational theory induced by the equation $\mathsf{dec}(\mathsf{enc}(x, y), y) = x$. Intuitively, the symbols enc and dec represent symmetric encryption and decryption, whereas h is used to model a hash function. Now, assume that the environment knows the key k as well as the ciphertext $\mathsf{enc}(n, k)$, and that these two messages are referred to by handles w and w'. The environment may decrypt the ciphertext with the key k, apply the hash function, and encrypt the result using k to get the message $m_0 = \mathsf{enc}(\mathsf{h}(n), k)$. This computation is modelled using the *recipe* $M_0 = \mathsf{enc}(\mathsf{h}(\mathsf{dec}(w', w)), w)$.

▶ **Definition 2.** Processes are defined by the following syntax where $c, a \in \mathcal{C}$, $x \in \mathcal{X}$, $u, v \in \mathcal{T}(\mathcal{N} \cup \mathcal{X})$, and $\vec{c}$ (resp. $\vec{n}$) is a sequence of channels from $\mathcal{C}$ (resp. names from $\mathcal{N}$).

$$P, Q \ ::= \ 0 \ \mid \ (P \mid Q) \ \mid \ \mathtt{in}(c, x).P \ \mid \ \mathtt{out}(c, u).P \ \mid \ \mathtt{if}\ u = v\ \mathtt{then}\ P\ \mathtt{else}\ Q \ \mid \ !^{a}_{\vec{c},\vec{n}} P$$

The last construct combines replication with channel and name restriction: $!^a_{\vec{c},\vec{n}}P$ may be read as $!(\nu\vec{c}.\mathtt{out}(a,\vec{c}).\nu\vec{n}.P)$ in standard applied π-calculus. Our goal with this compound construct is to support replication in a way that is not fundamentally incompatible with the action-determinism condition which we eventually impose on our processes. This is achieved here by advertising on the public channel a any new copy of the replicated process. At the same time, we make public the new channels $\vec{c}$ on which the copy may operate – but not the new names $\vec{n}$. While it may seem restrictive, this style is actually natural for security protocols where the attacker knows exactly to whom he is sending a message and from whom he is receiving, *e.g.,* via IP addresses.

We shall only consider *ground* processes, where each variable is bound by an input. We denote by $\mathsf{fc}(P)$ and $\mathsf{bc}(P)$ the set of *free* and *bound channels* of P.

▶ **Example 3.** The process P_0 models an agent who sends the ciphertext $\mathtt{enc}(n,k)$, and then waits for an input on c. In case the input has the expected form, the constant $\mathtt{ok}$ is emitted.

$$P_0 = \mathtt{out}(c,\mathtt{enc}(n,k)).\mathtt{in}(c,x).\mathtt{if}\ \mathtt{dec}(x,k) = \mathsf{h}(n)\ \mathtt{then}\ \mathtt{out}(c,\mathtt{ok}).0\ \mathtt{else}\ 0$$

The processes P_0 as well as $!^a_{c,n}P_0$ are ground. We have that $\mathsf{fc}(P_0) = \{c\}$ and $\mathsf{bc}(P_0) = \emptyset$ whereas $\mathsf{fc}(!^a_{c,n}P_0) = \{a\}$ and $\mathsf{bc}(!^a_{c,n}P_0) = \{c\}$.

2.2 Semantics

We only consider processes that are *normal* w.r.t. *internal reduction* $\rightsquigarrow$ defined as follows:

$$\left.\begin{array}{l}\mathtt{if}\ u = v\ \mathtt{then}\ P\ \mathtt{else}\ Q \rightsquigarrow P\ \textit{when}\ u =_{\mathsf{E}} v \qquad P \mid Q \rightsquigarrow P' \mid Q \\ \mathtt{if}\ u = v\ \mathtt{then}\ P\ \mathtt{else}\ Q \rightsquigarrow Q\ \textit{when}\ u \neq_{\mathsf{E}} v \qquad Q \mid P \rightsquigarrow Q \mid P'\end{array}\right\}\ \textit{when}\ P \rightsquigarrow P'$$

$$(P_1 \mid P_2) \mid P_3 \rightsquigarrow P_1 \mid (P_2 \mid P_3) \qquad P \mid 0 \rightsquigarrow P \qquad 0 \mid P \rightsquigarrow P$$

Any process in normal form built from parallel composition can be uniquely written as $P_1 \mid (P_2 \mid (\ldots \mid P_n))$ with $n \geq 2$, which we denote $\Pi_{i=1}^n P_i$, where each process P_i is neither a parallel composition nor the process 0.

We now define our labelled transition system. It deals with *configurations* (denoted by A, B) which are pairs $(\mathcal{P};\Phi)$ where $\mathcal{P}$ is a multiset of ground processes and Φ, called the *frame*, is a substitution mapping handles to messages that have been made available to the environment.

Given a configuration A, $\Phi(A)$ denotes its second component. Given a frame Φ, $\mathrm{dom}(\Phi)$ denotes its domain.

In	$(\{\mathtt{in}(c,x).Q\} \uplus \mathcal{P};\Phi) \xrightarrow{\mathtt{in}(c,M)} (\{Q\{M\Phi/x\}\} \uplus \mathcal{P};\Phi)$	$M \in \mathcal{T}(\mathrm{dom}(\Phi))$
Out	$(\{\mathtt{out}(c,u).Q\} \uplus \mathcal{P};\Phi) \xrightarrow{\mathtt{out}(c,w)} (\{Q\} \uplus \mathcal{P};\Phi \cup \{w \mapsto u\})$	$w \in \mathcal{W}$ fresh
Repl	$(\{!^a_{\vec{c},\vec{n}}P\} \uplus \mathcal{P};\Phi) \xrightarrow{\mathtt{sess}(a,\vec{c})} (\{P;!^a_{\vec{c},\vec{n}}P\} \uplus \mathcal{P};\Phi)$	$\vec{c},\vec{n}$ fresh
Par	$(\{\Pi_{i=1}^n P_i\} \uplus \mathcal{P};\Phi) \xrightarrow{\tau} (\{P_1,\ldots,P_n\} \uplus \mathcal{P};\Phi)$	
Zero	$(\{0\} \uplus \mathcal{P};\Phi) \xrightarrow{\tau} (\mathcal{P};\Phi)$	

Rule In expresses that an input process may receive any message that the environment can derive from the current frame. In rule Out, the frame is enriched with a new message. The last two rules simply translate the parallel structure of processes into the multiset structure of the configuration. As explained above, rule Repl combines the replication of a process together with the creation of new channels and nonces. The channels $\vec{c}$ are implicitly made public, but the newly created names $\vec{n}$ remain private. Remark that channels $\vec{c}$ and names $\vec{n}$ must be fresh, *i.e.,* they do not appear free in the original configuration. As

usual, freshness conditions do not block executions: it is always possible to rename bound channels $\vec{c}$ and names $\vec{n}$ of a process $!^a_{\vec{c},\vec{n}}P$ before applying Repl. We denote by $\mathsf{bc}(\mathsf{tr})$ the bound channels of a trace tr, *i.e.,* all the channels that occur in second argument of an action $\mathtt{sess}(a, \vec{c})$ in tr, and we consider traces where channels are bound at most once.

▶ **Example 4.** Going back to Example 3 with $\Phi_0 = \{w_1 \mapsto k\}$, we have that:

$$(\{!^a_{c,n}P_0\};\Phi_0) \xrightarrow{\mathtt{sess}(a,c)} \xrightarrow{\mathtt{out}(c,w_2)} \xrightarrow{\mathtt{in}(c,M_0)} (\{\mathtt{out}(c,\mathsf{ok}).0;!^a_{c,n}P_0\};\Phi)$$

where $\Phi = \{w_1 \mapsto k, w_2 \mapsto \mathsf{enc}(n,k)\}$ and $M_0 = \mathsf{enc}(\mathsf{h}(\mathsf{dec}(w_2,w_1)),w_1)$.

2.3 Equivalences

We are concerned with trace equivalence, which is used [7, 15] to model anonymity, untraceability, strong secrecy, etc. Finer behavioural equivalences, *e.g.,* weak bisimulation, appear to be too strong with respect to what an attacker can really observe. Intuitively, two configurations are trace equivalent if the attacker cannot tell whether he is interacting with one or the other. To make this formal, we introduce a notion of equivalence between frames.

▶ **Definition 5.** Two frames Φ and Φ' are in *static equivalence*, written $\Phi \sim \Phi'$, when $\mathrm{dom}(\Phi) = \mathrm{dom}(\Phi')$, and: $M\Phi =_{\mathsf{E}} N\Phi \Leftrightarrow M\Phi' =_{\mathsf{E}} N\Phi'$ for any terms $M, N \in \mathcal{T}(\mathrm{dom}(\Phi))$.

▶ **Example 6.** Continuing Example 4, consider $\Phi' = \{w_1 \mapsto k', w_2 \mapsto \mathsf{enc}(n,k)\}$. The test $\mathsf{enc}(\mathsf{dec}(w_2,w_1),w_1) = w_2$ is true in Φ but not in Φ', thus $\Phi \not\sim \Phi'$.

We then define $\mathsf{obs}(\mathsf{tr})$ to be the subsequence of tr obtained by erasing τ actions.

▶ **Definition 7.** Let A and B be two configurations. We say that $A \sqsubseteq B$ when, for any $A \xrightarrow{\mathsf{tr}} A'$ such that $\mathsf{bc}(\mathsf{tr}) \cap \mathsf{fc}(B) = \emptyset$, there exists $B \xrightarrow{\mathsf{tr}'} B'$ such that $\mathsf{obs}(\mathsf{tr}) = \mathsf{obs}(\mathsf{tr}')$ and $\Phi(A') \sim \Phi(B')$. They are *trace equivalent*, written $A \approx B$, when $A \sqsubseteq B$ and $B \sqsubseteq A$.

In order to lift our optimised semantics to trace equivalence, we will require configurations to be *action-deterministic*. This common assumption in POR techniques [6] is also reasonable in the context of security protocols, where the attacker knows with whom he is communicating.

▶ **Definition 8.** A configuration A is *action-deterministic* if whenever $A \xrightarrow{\mathsf{tr}} (\mathcal{P};\Phi)$, and P, Q are two elements of $\mathcal{P}$, we have that P and Q cannot perform an observable action of the same nature (`in`, `out`, or `sess`) on the same channel (*i.e.,* if both actions are of same nature, their first argument has to differ).

3 Annotated semantics

We shall now define an intermediate semantics whose transitions are equipped with more informative actions. The annotated actions will notably feature *labels* $\ell \in \mathbb{N}^*$ indicating from which concurrent processes they originate. A *labelled action* will be written $[\alpha]^\ell$ where α is an action and ℓ is a label. Similarly, a *labelled process* will be written $[P]^\ell$. When reasoning about trace equivalence between two configurations, it will be crucial to maintain a consistent labelling between configurations along the execution. In order to do so, we define *skeletons of observable actions*, which are of the form $\mathtt{in}_c$, $\mathtt{out}_c$ or $!^a$ where $a, c \in \mathcal{C}$, and we assume a total ordering over those skeletons, denoted $<$ with $\leq$ being its reflexive closure. Any process that is neither 0 nor a parallel composition induces a skeleton corresponding to its toplevel connective, and we denote it by $\mathsf{sk}(P)$ (*e.g.,* $\mathsf{sk}(\mathtt{in}(c,x).0) = \mathtt{in}_c$).

$$\textsc{In} \quad (\{[\mathtt{in}(c,x).Q]^\ell\} \uplus \mathcal{P}; \Phi) \xrightarrow{[\mathtt{in}(c,M)]^\ell}_a (\{[Q\{M\Phi/x\}]^\ell\} \uplus \mathcal{P}; \Phi) \quad M \in \mathcal{T}(\mathrm{dom}(\Phi))$$

$$\textsc{Out} \quad (\{[\mathtt{out}(c,u).Q]^\ell\} \uplus \mathcal{P}; \Phi) \xrightarrow{[\mathtt{out}(c,w)]^\ell}_a (\{[Q]^\ell\} \uplus \mathcal{P}; \Phi \cup \{w \mapsto u\}) \quad w \in \mathcal{W} \text{ fresh}$$

$$\textsc{Repl} \quad (\{[!^a_{\vec{c},\vec{n}} P_0]^\ell\} \uplus \mathcal{P}; \Phi) \xrightarrow{[\mathtt{sess}(a,\vec{c})]^\ell}_a (\{[P_0]^{\ell\cdot 1}, [!^a_{\vec{c},\vec{n}} P_0]^{\ell\cdot 2}\} \uplus \mathcal{P}; \Phi) \qquad \vec{c}, \vec{n} \text{ fresh}$$

$$\textsc{Par} \quad (\{[\Pi_{i=1}^n P_i]^\ell\} \uplus \mathcal{P}; \Phi) \xrightarrow{[\mathtt{par}(\sigma_{\pi(1)};\ldots;\sigma_{\pi(n)})]^\ell}_a (\{[P_{\pi(1)}]^{\ell\cdot 1}, \ldots, [P_{\pi(n)}]^{\ell\cdot n}\} \uplus \mathcal{P}; \Phi)$$
$$\sigma_i = \mathsf{sk}(P_i) \text{ and } \pi \text{ is a permutation over } [1,...n] \text{ such that } \sigma_{\pi(1)} \leq \ldots \leq \sigma_{\pi(n)}$$

$$\textsc{Zero} \quad (\{[0]^\ell\} \uplus \mathcal{P}; \Phi) \xrightarrow{[\mathtt{zero}]^\ell}_a (\mathcal{P}; \Phi)$$

Figure 1 Annotated semantics.

We then define in Figure 1 the *annotated semantics* $\rightarrow_a$ over configurations whose processes are labelled. In Par, note that $\mathsf{sk}(P_i)$ is well defined as P_i cannot be 0 nor a parallel composition. Note that the annotated transition system does not restrict the executions of a process but simply annotates them with labels, and replaces τ actions by more descriptive actions.

We now define how to extract dependencies from annotated traces, which will allow us to analyse concurrency in an execution without referring to configurations. We obtain sequential dependencies from labels, in a way that is similar, *e.g.,* to the use of *causal relations* in CCS [14]. We also define *recipe dependencies* which are a sort of data dependencies reflecting our specific setting, where we consider an arbitrary attacker who may interact with the process, relying on (maybe several) previously outputted messages to derive input messages.

▶ **Definition 9.** Two labels are *dependent* if one is a prefix of the other. We say that the labelled actions α and β are *sequentially dependent* when their labels are dependent, and *recipe dependent* when $\{\alpha, \beta\} = \{[\mathtt{in}(c, M)]^\ell, [\mathtt{out}(c', w)]^{\ell'}\}$ with w occurring in M. They are *dependent* when they are sequentially or recipe dependent. Otherwise, they are *independent.*

▶ **Definition 10.** A configuration $(\mathcal{P}; \Phi)$ is *well labelled* if $\mathcal{P}$ is a multiset of labelled processes such that two elements of $\mathcal{P}$ have independent labels.

Obviously, any unlabelled configuration may be well labelled. Further, it is easy to see that well labelling is preserved by $\rightarrow_a$. Thus, we shall implicitly assume to be working with well labelled configurations. Under this assumption, we obtain the following lemma.

▶ **Lemma 11.** *Let A be a (well labelled) configuration, α and β be two independent labelled actions. We have $A \xrightarrow{\alpha.\beta}_a A'$ if, and only if, $A \xrightarrow{\beta.\alpha}_a A'$.*

Symmetries of trace equivalence. We will see that, when checking $A \approx B$ for action-deterministic configurations, it is sound to require that B can perform all traces of A in the annotated semantics (and the converse). In other words, labels and detailed non-observable actions $\mathtt{zero}$ and $\mathtt{par}(\sigma_1 \ldots \sigma_n)$ are actually relevant for trace equivalence. Obviously, this can only hold if A and B are labelled consistently. In order to express this, we extend $\mathsf{sk}(P)$ to parallel and 0 processes: we let their skeletons be the associated action in the annotated semantics. Next, we define the labelled skeletons by $\mathsf{skl}([P]^\ell) = [\mathsf{sk}(P)]^\ell$. When checking for equivalence of A and B, we shall assume that $\mathsf{skl}(A) = \mathsf{skl}(B)$, *i.e.,* the configurations have the same set of labelled skeletons. This technical condition is not restrictive in practice.

▶ **Example 12.** Let $A = (\{[\mathtt{in}(a,x).((\mathtt{out}(b,m).P_1) \mid P_2)]^0\}; \Phi)$ with $P_1 = \mathtt{in}(c,y).0$ and $P_2 = \mathtt{in}(d,z).0$, and B the configuration obtained from A by swapping P_1 and P_2. We have $\mathsf{skl}(A) = \mathsf{skl}(B) = \{[\mathtt{in}_a]^0\}$. Consider the following trace:

$$\mathsf{tr} = [\mathtt{in}(a,\mathtt{ok})]^0.[\mathtt{par}(\{\mathtt{out}_b;\mathtt{in}_d\})]^0.[\mathtt{out}(b,w)]^{0\cdot1}.[\mathtt{in}(c,w)]^{0\cdot1}.[\mathtt{in}(d,w)]^{0\cdot2}$$

Assuming $\mathtt{out}_b < \mathtt{in}_d$ and $\mathsf{ok} \in \Sigma$, we have $A \xrightarrow{\mathsf{tr}}_a A'$. However, there is no B' such that $B \xrightarrow{\mathsf{tr}}_a B'$, for two reasons. First, B cannot perform the second action since skeletons of subprocesses of its parallel composition are $\{\mathtt{out}_b;\mathtt{in}_c\}$. Second, B would not be able to perform the action $\mathtt{in}(c,w)$ with the right label. Such mismatches can actually be systematically used to show $A \not\approx B$, as shown next.

▶ **Lemma 13.** *Let A and B be two action-deterministic configurations such that $A \approx B$ and $\mathsf{skl}(A) = \mathsf{skl}(B)$. For any execution $A \xrightarrow{[\alpha_1]^{\ell_1}}_a A_1 \xrightarrow{[\alpha_2]^{\ell_2}}_a A_2 \dots \xrightarrow{[\alpha_n]^{\ell_n}}_a A_n$ with $\mathsf{bc}(\alpha_1.\dots.\alpha_n) \cap \mathsf{fc}(B) = \emptyset$, there exists an execution $B \xrightarrow{[\alpha_1]^{\ell_1}}_a B_1 \xrightarrow{[\alpha_2]^{\ell_2}}_a B_2 \dots \xrightarrow{[\alpha_n]^{\ell_n}}_a B_n$ such that $\Phi(A_i) \sim \Phi(B_i)$ and $\mathsf{skl}(A_i) = \mathsf{skl}(B_i)$ for any $1 \le i \le n$.*

4 Compression

Our first refinement of the semantics, which we call compression, is closely related to focusing from proof theory [2]: we will assign a polarity to processes and constrain the shape of executed traces based on those polarities. This will provide a first significant reduction of the number of traces to consider when checking reachability-based properties such as secrecy, and more importantly, equivalence-based properties in the action-deterministic case. Moreover, compression can easily be used as a replacement for the usual semantics in verification algorithms.

▶ **Definition 14.** A process P is *positive* if it is of the form $\mathtt{in}(c,x).Q$, and it is *negative* otherwise. A multiset of processes $\mathcal{P}$ is *initial* if it contains only positive or *replicated* processes, *i.e.,* of the form $!^a_{\vec{c},\vec{n}}Q$.

The compressed semantics (see Figure 2) is built upon the annotated semantics. It constrains the traces to follow a particular strategy, alternating between *negative* and *positive* phases. It uses enriched configurations of the form $(\mathcal{P}; F; \Phi)$ where $(\mathcal{P}; \Phi)$ is a labelled configuration and F is either a process (signalling which process is *under focus* in the positive phase) or $\varnothing$ (in the negative phase). The negative phase lasts until the configuration is initial (*i.e.,* unfocused with an initial underlying multiset of processes) and in that phase we perform actions that decompose negative non-replicated processes. This is done using the Neg rule, in a completely deterministic way. When the configuration becomes initial, a positive phase can be initiated: we choose one process and start executing the actions of that process (only inputs, possibly preceded by a new session) without the ability to switch to another process of the multiset, until a negative subprocess is released and we go back to the negative phase. The active process in the positive phase is said to be *under focus*. Between any two initial configurations, the compressed semantics executes a sequence of actions, called *blocks*, of the form $\mathtt{foc}(\alpha).\mathsf{tr}^+.\mathtt{rel}.\mathsf{tr}^-$ where tr^+ is a (possibly empty) sequence of input actions, whereas tr^- is a (possibly empty) sequence of $\mathtt{out}$, $\mathtt{par}$, and $\mathtt{zero}$ actions. Note that, except for choosing recipes, the compressed semantics is completely non-branching when executing a block. It may branch only when choosing which block is performed.

$$\textsc{Start/In}\quad \frac{\mathcal{P} \text{ is initial} \quad (P;\Phi) \xrightarrow{\mathtt{in}(c,M)}_a (P';\Phi)}{(\mathcal{P} \uplus \{P\};\varnothing;\Phi) \xrightarrow{\mathtt{foc}(\mathtt{in}(c,M))}_c (\mathcal{P};P';\Phi)}$$

$$\textsc{Start/!}\quad \frac{\mathcal{P} \text{ is initial} \quad (!^a_{\vec{c},\vec{n}}P;\Phi) \xrightarrow{\mathtt{sess}(a,\vec{c})}_a (\{!^a_{\vec{c},\vec{n}}P;Q\};\Phi)}{(\mathcal{P} \uplus \{!^a_{\vec{c},\vec{n}}P\};\varnothing;\Phi) \xrightarrow{\mathtt{foc}(\mathtt{sess}(a,\vec{c}))}_c (\mathcal{P} \uplus \{!^a_{\vec{c},\vec{n}}P\};Q;\Phi)}$$

$$\textsc{Pos/In}\quad \frac{(P;\Phi) \xrightarrow{\mathtt{in}(c,M)}_a (P';\Phi)}{(\mathcal{P};P;\Phi) \xrightarrow{\mathtt{in}(c,M)}_c (\mathcal{P};P';\Phi)}$$

$$\textsc{Neg}\quad \frac{(P;\Phi) \xrightarrow{\alpha}_a (\mathcal{P}';\Phi')}{(\mathcal{P} \uplus \{P\};\varnothing;\Phi) \xrightarrow{\alpha}_c (\mathcal{P} \uplus \mathcal{P}';\varnothing;\Phi')}\ \alpha \in \{\mathtt{par}(\_),\mathtt{zero},\mathtt{out}(\_,\_)\}$$

$$\textsc{Release}\quad (\mathcal{P};[P]^\ell;\Phi) \xrightarrow{[\mathtt{rel}]^\ell}_c (\mathcal{P} \uplus \{[P]^\ell\};\varnothing;\Phi) \quad \text{when } P \text{ is negative}$$

Labels are implicitly set in the same way as in the annotated semantics. Neg is made non-branching by imposing an arbitrary order on labelled skeletons of available actions.

Figure 2 Compressed semantics.

▶ **Example 15.** Consider the process $P = !^a_{c,k}\mathtt{in}(c,x).\mathtt{out}(c,\mathsf{enc}(x,k)).0$. We have that:

$$\begin{array}{lll}(\{P\};\varnothing;\Phi) & \xrightarrow{\mathtt{foc}(\mathtt{sess}(a,c_i))}_c & (\{P\};\{\mathtt{in}(c_i,x).\mathtt{out}(c,\mathsf{enc}(x,k_i)).0\};\Phi)\\ & \xrightarrow{\mathtt{in}(c_i,M_i).\mathtt{rel}}_c & (\{P,\mathtt{out}(c,\mathsf{enc}(M_i\Phi,k_i)).0\};\varnothing;\Phi)\\ & \xrightarrow{\mathtt{out}(c_i,w_i).\mathtt{zero}}_c & (\{P\};\varnothing;\Phi').\end{array}$$

Once a replication is performed, the resulting process is under focus and must be executed in priority until the end. Note that, after executing the input, the resulting process is negative and, thus, still has priority. Thus, on this example, all compressed executions are made of blocks of the form: $\mathtt{sess}(a,c_i).\mathtt{in}(c_i,M_i).\mathtt{out}(c_i.w_i)$.

4.1 Reachability

We now formalise the relationship between traces of the compressed and annotated semantics. In order to do so, we translate between configuration and enriched configuration as follows:

$$\lceil(\mathcal{P};\Phi)\rceil = (\mathcal{P};\varnothing;\Phi),\quad \lfloor(\mathcal{P};\varnothing;\Phi)\rfloor = (\mathcal{P};\Phi)\quad \text{and } \lfloor(\mathcal{P};P;\Phi)\rfloor = (\mathcal{P}\uplus\{P\};\Phi).$$

Similarly, we map compressed traces to annotated ones:

$$\lfloor\epsilon\rfloor = \epsilon,\quad \lfloor\mathtt{foc}(\alpha).\mathsf{tr}\rfloor = \alpha.\lfloor\mathsf{tr}\rfloor,\quad \lfloor\mathtt{rel}.\mathsf{tr}\rfloor = \lfloor\mathsf{tr}\rfloor\quad \text{and } \lfloor\alpha.\mathsf{tr}\rfloor = \alpha.\lfloor\mathsf{tr}\rfloor \text{ otherwise.}$$

We observe that we can map any execution in the compressed semantics to an execution in the annotated semantics. Indeed, a compressed execution is simply an annotated execution with some extra annotations (*i.e.,* `foc` and `rel`) indicating positive/negative phase changes.

▶ **Lemma 16.** *For any configurations A, A' and* tr, $A \xrightarrow{\mathsf{tr}}_c A'$ *implies* $\lfloor A\rfloor \xrightarrow{\lfloor\mathsf{tr}\rfloor}_a \lfloor A'\rfloor$.

Going in the opposite direction is more involved. In general, mapping annotated executions to compressed ones requires to reorder actions. Compressed executions also force negative actions to be performed unconditionally and blocks to be fully executed. One way to handle this is to consider *complete* executions of a configuration, *i.e.,* executions after which no more action can be performed except possibly the ones that consist in unfolding a replication (*i.e.,* rule Repl). Inspired by the positive trunk argument of [19], we show the following lemma.

▶ **Lemma 17.** *Let A, A' be two configurations and* tr *be such that* $A \xrightarrow{\mathsf{tr}}_a A'$ *is complete. There exists a trace* tr_c*, such that* $\lfloor \mathsf{tr}_c \rfloor$ *can be obtained from* tr *by swapping independent labelled actions, and* $\lceil A \rceil \xrightarrow{\mathsf{tr}_c}_c \lceil A' \rceil$.

Proof sketch. We proceed by induction on the length of a complete execution starting from A. If A is not initial, then we need to execute some negative action using Neg: this action must be present somewhere in the complete execution, and we can permute it with preceding actions using Lemma 11. If A is initial, we analyse the prefix of input and session actions and we extract a subsequence of that prefix that corresponds to a full positive phase. ◀

4.2 Equivalence

We now define compressed trace equivalence ($\approx_c$) and prove that it coincides with $\approx$.

▶ **Definition 18.** Let A and B be two configurations. We say that $A \sqsubseteq_c B$ when, for any $A \xrightarrow{\mathsf{tr}}_c A'$ such that $\mathsf{bc}(\mathsf{tr}) \cap \mathsf{fc}(B) = \varnothing$, there exists $B \xrightarrow{\mathsf{tr}}_c B'$ such that $\Phi(A') \sim \Phi(B')$. They are *compressed trace equivalent*, denoted $A \approx_c B$, if $A \sqsubseteq_c B$ and $B \sqsubseteq_c A$.

Compressed trace equivalence can be more efficiently checked than regular trace equivalence. Obviously, it explores fewer interleavings by relying on $\to_c$ rather than $\to$. It also requires that traces of one process can be played exactly by the other, including details such as non-observable actions, labels, and focusing annotations. The subtleties shown in Example 12 are crucial for the completeness of compressed equivalence w.r.t. regular equivalence. Since the compressed semantics forces to perform available outputs before *e.g.* input actions, some non-equivalences are only detected thanks to the labels and detailed non-observable actions of our annotated semantics.

▶ **Theorem 19.** *Let A and B be two action-deterministic configurations with* $\mathsf{skl}(A) = \mathsf{skl}(B)$*. We have $A \approx B$ if, and only if,* $\lceil A \rceil \approx_c \lceil B \rceil$.

Proof sketch. ($\Rightarrow$) Consider an execution $\lceil A \rceil \xrightarrow{\mathsf{tr}}_c A'$. Using Lemma 16, we get $A \xrightarrow{\lfloor \mathsf{tr} \rfloor}_a \lfloor A' \rfloor$. Then, Lemma 13 yields $B \xrightarrow{\lfloor \mathsf{tr} \rfloor}_a B'$ for some B' such that $\Phi(\lfloor A' \rfloor) \sim \Phi(B')$ and labelled skeletons are equal all along the executions. Relying on those skeletons, we show that positive/negative phases are synchronised, and thus $\lceil B \rceil \xrightarrow{\mathsf{tr}}_c B''$ for some B'' with $\lfloor B'' \rfloor = B'$. ($\Leftarrow$) Consider an execution $A \xrightarrow{\mathsf{tr}}_a A'$. We first observe that it suffices to consider only complete executions there. This allows us to get a compressed execution $\lceil A \rceil \xrightarrow{\mathsf{tr}_c}_c \lceil A' \rceil$ by Lemma 17. Since $\lceil A \rceil \approx_c \lceil B \rceil$, there exists B' such that $\lceil B \rceil \xrightarrow{\mathsf{tr}_c}_c B'$ with $\Phi(\lceil A' \rceil) \sim \Phi(B')$. Thus we have $B \xrightarrow{\lfloor \mathsf{tr}_c \rfloor}_a \lfloor B' \rfloor$ but also $B \xrightarrow{\mathsf{tr}}_a \lfloor B' \rfloor$ thanks to Lemma 11. ◀

Improper blocks. Note that blocks of the form $\mathtt{foc}(\alpha).\mathsf{tr}^+.\mathtt{rel.zero}$ do not bring any new information to the attacker. While it would be incorrect to fully ignore such *improper* blocks, it is in fact sufficient to only consider them at the end of traces. We show in [5] that $\approx_c$ coincides with a further optimised compressed trace equivalence that only checks for *proper traces*, *i.e.,* ones that have at most one improper block and only at the end of trace.

5 Reduction

Our compressed semantics cuts down interleavings by using a simple focused strategy. However, this semantics does not analyse data dependency that happen when an input depends on an output, and is thus unable to exploit the independency of blocks to reduce interleavings. We tackle this problem now.

▶ **Definition 20.** Two blocks b_1 and b_2 are *independent*, written $b_1 \parallel b_2$, when all labelled actions $\alpha_1 \in b_1$ and $\alpha_2 \in b_2$ are independent. Otherwise they are *dependent*, written $b_1 \rightleftharpoons b_2$.

Obviously, Lemma 11 tells us that independent blocks can be permuted in a trace without affecting the executability and the result of executing that trace. But this notion is not very strong since it considers fixed recipes, which are irrelevant (in the end, only the derived messages matter) and can easily introduce spurious dependencies. Thus we define a stronger notion of equivalence over traces, which allows permutations of independent blocks but also changes of recipes that preserve messages. During these permutations, we will also require that traces remain *plausible*, which is defined as follows: tr is plausible if for any input $\mathtt{in}(c, M)$ such that $\mathsf{tr} = \mathsf{tr}_0.\mathtt{in}(c, M).\mathsf{tr}_2$ then $M \in \mathcal{T}(\mathcal{W})$ where $\mathcal{W}$ is the set of all handles occurring in tr_0. Given a block b, *i.e.,* a sequence of the form $\mathtt{foc}(\alpha).\mathsf{tr}^+.\mathtt{rel}.\mathsf{tr}^-$, we denote by b^+ (resp. b^-) the part of b corresponding to the positive (resp. negative) phase, *i.e.,* $b^+ = \alpha.\mathsf{tr}^+$ (resp. $b^- = \mathsf{tr}^-$). We note $(b_1 =_{\mathsf{E}} b_2)\Phi$ when $b_1^+\Phi =_{\mathsf{E}} b_2^+\Phi$ and $b_1^- = b_2^-$.

▶ **Definition 21.** Given a frame Φ, the relation $\equiv_\Phi$ is the smallest equivalence over plausible compressed traces such that $\mathsf{tr}.b_1.b_2.\mathsf{tr}' \equiv_\Phi \mathsf{tr}.b_2.b_1.\mathsf{tr}'$ when $b_1 \parallel b_2$, and $\mathsf{tr}.b_1.\mathsf{tr}' \equiv_\Phi \mathsf{tr}.b_2.\mathsf{tr}'$ when $(b_1 =_{\mathsf{E}} b_2)\Phi$.

▶ **Lemma 22.** *Let A and A' be two initial configurations such that $A \xrightarrow{\mathsf{tr}}_c A'$. We have that $A \xrightarrow{\mathsf{tr}'}_c A'$ for any $\mathsf{tr}' \equiv_{\Phi(A')} \mathsf{tr}$.*

We now turn to defining our reduced semantics, which is going to avoid the redundancies identified above by only executing specific representatives in equivalence classes modulo $\equiv_\Phi$. More precisely, we shall only execute minimal traces according to some order, which we now introduce. We assume an order $\prec$ on blocks that is insensitive to recipes, and such that independent blocks are always strictly ordered in one way or the other. We finally define $\prec_{\mathsf{lex}}$ on compressed traces as the lexicographic extension of $\prec$ on blocks.

In order to incrementally build representatives that are minimal with respect to $\prec_{\mathsf{lex}}$, we define a predicate that expresses whether a block b should be *authorised* after a given trace tr. Intuitively, this is the case only when, for any block $b' \succ b$ in tr, dependencies forbid to swap b and b'. We define this with recipe dependencies first, then quantify over all recipes to capture message dependencies.

▶ **Definition 23.** A block b is authorised after tr, noted $\mathsf{tr} \triangleright b$, when $\mathsf{tr} = \epsilon$; or $\mathsf{tr} = \mathsf{tr}_0.b_0$ and either *(i)* $b \rightleftharpoons b_0$ or *(ii)* $b \parallel b_0$, $b_0 \prec b$, and $\mathsf{tr}_0 \triangleright b$.

We finally define $\rightarrow_r$ as the least relation such that:

$$\textsc{Init}\ \frac{}{A \xrightarrow{\epsilon}_r A} \qquad \textsc{Block}\ \frac{A \xrightarrow{\mathsf{tr}}_r (\mathcal{P};\varnothing;\Phi) \quad (\mathcal{P};\varnothing;\Phi) \xrightarrow{b}_c A'}{A \xrightarrow{\mathsf{tr}.b}_r A'}\ \begin{array}{l}\text{if } \mathsf{tr} \triangleright b' \text{ for all } b' \\ \text{with } (b' =_{\mathsf{E}} b)\Phi\end{array}$$

Our reduced semantics only applies to initial configurations: otherwise, no block can be performed. This is not restrictive since we can, without loss of generality, pre-execute non-observable and output actions that may occur at top level.

▶ **Example 24.** We consider roles $R_i := \mathtt{in}(c_i, x).\mathtt{if}\ x = \mathsf{ok}\ \mathtt{then}\ \mathtt{out}(c_i, \mathsf{ok})$ where ok is a public constant, and then consider a parallel composition of n such processes: $P_n := \Pi_{i=1}^n R_i$. Thanks to compression, we will only consider traces made of blocks, and obtain a first exponential reduction of the state space. However, contrary to the case of a replicated process (see Example 15), we still have many interleavings to consider – blocks can be interleaved in all the possible ways. We will see that our reduced semantics cuts down these interleavings.

Assume that our order $\prec$ prioritises blocks on c_i over those on c_j when $i < j$, and consider a trace starting with $\mathtt{in}(c_j, M_j).\mathtt{out}(c_j, w_j)$. Trying to continue the exploration with a block on c_i with $i < j$, the authorisation predicate $\triangleright$ will impose that there is a dependency between the block on c_i and the previous one on c_j. In this case it must be a data dependency: the recipe of the message passed as input on c_i must make use of the previous output to derive ok. Since ok is a public constant, it is possible to derive it without using any previous output and thus the block on c_i cannot be authorised by $\triangleright$. Thus, on this simple example, the reduced semantics will not explore any trace where a block on c_i is performed after one on c_j with $i < j$.

5.1 Reachability

An easy induction on the compressed trace tr allows us to map an execution w.r.t. the reduced semantics to an execution w.r.t. the compressed semantics.

▶ **Lemma 25.** *For any configurations A and A', $A \xrightarrow{\mathsf{tr}}_r A'$ implies $A \xrightarrow{\mathsf{tr}}_c A'$.*

Next, we show that our reduced semantics only explores specific representatives. Given a frame Φ, a plausible trace tr is Φ-minimal if it is minimal in its equivalence class modulo $\equiv_\Phi$.

▶ **Lemma 26.** *Let A be an initial configuration and $A' = (\mathcal{P}; \varnothing; \Phi)$ be a configuration such that $A \xrightarrow{\mathsf{tr}}_c A'$. We have that tr is Φ-minimal if, and only if, $A \xrightarrow{\mathsf{tr}}_r A'$.*

Proof sketch. In order to relate minimality and executability in the reduced semantics, let us say that a trace is *bad* if it is of the form $\mathsf{tr}.b_0 \dots b_n.b'.\mathsf{tr}'$ where $n \geq 0$, there exists a block b'' such that $(b'' =_{\mathsf{E}} b')\Phi$, we have $b_i \parallel b''$ for all i, and $b_i \prec b'' \prec b_0$ for all $i > 0$. This pattern is directly inspired by the characterisation of lexicographic normal forms by Anisimov and Knuth in trace monoids [3]. We note that a trace that can be executed in the compressed semantics can also be executed in the reduced semantics if, and only if, it is not bad. Since the badness of a trace allows to swap b' before b_0, and thus obtain a smaller trace in the class $\equiv_\Phi$, we show that a bad trace cannot be Φ-minimal (and conversely). ◀

5.2 Equivalence

The reduced semantics induces an equivalence $\approx_r$ that we define similarly to the compressed one, and we then establish its soundness and completeness w.r.t. $\approx_c$.

▶ **Definition 27.** Let A and B be two configurations. We say that $A \sqsubseteq_r B$ when, for every $A \xrightarrow{\mathsf{tr}}_r A'$ such that $\mathsf{bc}(\mathsf{tr}) \cap \mathsf{fc}(B) = \varnothing$, there exists $B \xrightarrow{\mathsf{tr}}_r B'$ such that $\Phi(A') \sim \Phi(B')$. They are *reduced trace equivalent*, denoted $A \approx_r B$, if $A \sqsubseteq_r B$ and $B \sqsubseteq_r A$.

▶ **Theorem 28.** *Let A and B be two initial, action-deterministic configurations.*

$$A \approx_c B \text{ if, and only if, } A \approx_r B$$

Proof sketch. We first prove that $\mathsf{tr} \equiv_\Phi \mathsf{tr}'$ iff $\mathsf{tr} \equiv_\Psi \mathsf{tr}'$ when $\Phi \sim \Psi$. ($\Rightarrow$) This implication is then an easy consequence of Lemma 26. ($\Leftarrow$) We start by showing that it suffices to consider a complete execution $A \xrightarrow{\mathsf{tr}}_c A'$. Since A' is initial, by taking tr_m to be a $\Phi(A')$-minimal trace associated to tr, we obtain a reduced execution of A leading to A'. Using our hypothesis $A \approx_r B$, we obtain that $B \xrightarrow{\mathsf{tr}_m}_r B'$ with corresponding relations over frames. We finally conclude that $B \xrightarrow{\mathsf{tr}}_c B'$ using Lemma 22 and the result stated above. ◀

Improper blocks. Similarly as we did for the compressed semantics in Section 4, we can further restrict $\approx_r$ to only check proper traces.

6 Application

We have developed two successive refinements of the concrete semantics of our process algebra, eventually obtaining a reduced semantics that achieves an optimal elimination of redundant interleavings. However, the practical usability of these semantics in algorithms for checking the equivalence of replication-free processes is far from immediate: indeed, all of our semantics are still infinitely branching, because each input may be fed with arbitrary messages. We now discuss how existing decision procedures based on symbolic execution [18, 12, 23, 9] can be modified to decide our optimised equivalences rather than the regular one, before presenting our implementation and experimental results.

6.1 Symbolic execution

Our compressed semantics can easily be used as a replacement of the regular one, in any tool whose algorithm is based on a forward exploration of the set of possible traces. This modification is very lightweight, and already brings a significant optimisation. In order to make use of our final, reduced semantics, we would need to enter into the details of constraint solving. In addition to imposing the compressed strategy and the sequential dependencies imposed by our predicate $\mathsf{tr} \triangleright b$, symbolic execution should be modified to generate *dependency constraints* in order to reflect the data dependencies imposed by $\mathsf{tr} \triangleright b$. The generation of dependency constraints can be done in a similar way to [4]. The constraint solver is then modified in a non-invasive way: dependency constraints are used to dismiss configurations when it becomes obvious that they cannot be satisfied.

▶ **Example 29.** We consider the symbolic reduced executions of process P_n from Example 24. In symbolic executions, input recipes and messages are initially left unknown, and gradually discovered by constraint resolution procedures. Assume that we have already executed a block on c_j. After that, we can execute symbolically an input on c_i, with $i < j$: let us write it $\mathtt{in}(c_i, X_i)$. Because we use the reduced semantics, a dependency constraint $\mathsf{dep}(X_i, w_j)$ is generated, expressing that the recipe denoted by X_i must depend on w_j. After executing its input, process R_i makes a test $(x = \mathsf{ok})$. Its `else` branch is trivial: it leads to an improper block, allowing us to stop any further exploration. When taking the `then` branch, we add a constraint expressing that recipe X_i must derive the message ok. In a tool such as Apte, this constraint is immediately solved by instantiating $X_i := \mathsf{ok}$ (considering other ways to derive ok is useless). After this instantiation, our dependency constraint has become $\mathsf{dep}(\mathsf{ok}, w_j)$ which is obviously unsatisfiable, and thus the branch is discarded.

The modified verification algorithm may explore symbolic traces that do not correspond to Φ-minimal representatives (when dependency constraints cannot be shown to be infeasible) but we will see that this approach allows us to obtain a very effective optimisation. Finally, note that, because we may over-approximate dependency constraints, we must ensure that constraint resolution prunes executions in a symmetrical fashion for both processes being checked for equivalence.

6.2 Experimental results

The optimisations developed in the present paper have been implemented, following the above approach, in the official version of the state of the art tool Apte [10]. We now report on experimental results; sources and instructions for reproduction are available [17]. We only show examples in which equivalence holds, because the time spent on inequivalent processes is too sensitive to the order in which the (depth-first) exploration is performed.

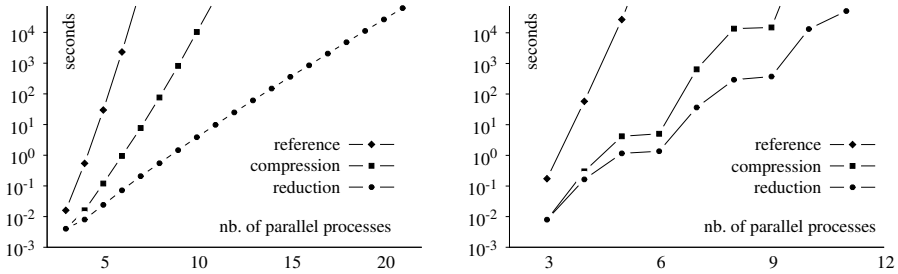

Figure 3 Impact of optimisations on toy example (left) and Denning-Sacco (right).

Toy example. We consider again our simple example described in Section 6.1. We ran Apte on $P_n \approx P_n$ for $n = 1$ to 22, on a single 2.67GHz Xeon core (memory is not relevant). We performed our tests on the reference version and the versions optimised with the compressed and reduced semantics respectively. The results are shown on the left graph of Figure 3, in logarithmic scale: it confirms that each optimisation brings an exponential speedup, as predicted by our theoretical analysis.

Denning-Sacco protocol. We ran a similar benchmark, checking that Denning-Sacco ensures strong secrecy in various scenarios. The protocol has three roles and we added processes playing those roles in turn, starting with three processes in parallel. The results are plotted on Figure 3. The fact that we add one role out of three at each step explains the irregular growth in verification time. We still observe an exponential speedup for each optimisation.

Practical impact. Finally, we illustrate how our optimisations make Apte much more useful in practice for investigating interesting scenarios. Verifying a single session of a protocol brings little assurance into its security. In order to detect replay attacks and to allow the attacker to compare messages that are exchanged, at least two sessions should be considered. This means having at least four parallel processes for two-party protocols, and six when a trusted third party is involved. This is actually beyond what the unoptimised Apte can handle in a reasonable amount of time. We show below how many parallel processes could be handled in 20 hours by the different versions of Apte on various use cases of protocols.

Protocol	ref	comp	red	Protocol	ref	comp	red
Needham Schroeder (3-party)	4	6	7	Denning-Sacco (3-party)	5	9	10
Private Authent. (2-party)	4	7	7	WMF (3-party)	6	12	13
Yahalom (3-party)	4	5	5	E-Passport PA (2-party)	4	7	9

7 Conclusion

We have developed two POR techniques that are adequate for verifying reachability and trace equivalence properties of action-deterministic security protocols. We have effectively implemented them in Apte, and shown that they yield the expected, significant benefit.

We are considering several directions for future work. Regarding the theoretical results presented here, the main question is whether we can get rid of the action-determinism condition without degrading our reductions too much. Regarding the practical application of our results, we can certainly go further. We first note that our compression technique should be applicable and useful in other verification tools, not necessarily based on symbolic execution. Next, we could investigate the role of the particular choice of the order $\prec$, to

determine heuristics for maximising the practical impact of reduction. Finally, we plan to adapt our treatment of replication to bounded replication to obtain a first symmetry elimination scheme, which should provide a significant optimisation when studying security protocols with several sessions.

References

1 M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL'01*. ACM Press, 2001.
2 J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3), 1992.
3 A.V. Anisimov and D.E. Knuth. Inhomogeneous sorting. *International Journal of Computer & Information Sciences*, 8(4):255–260, 1979.
4 D. Baelde, S. Delaune, and L. Hirschi. A reduced semantics for deciding trace equivalence using constraint systems. In *Proc. of POST'14*. Springer, 2014.
5 David Baelde, Stéphanie Delaune, and Lucca Hirschi. Partial order reduction for security protocols. *CoRR*, abs/1504.04768, 2015.
6 C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
7 Mayla Bruso, K. Chatzikokolakis, and J. den Hartog. Formal verification of privacy for RFID systems. In *Proc. of CSF'10*, 2010.
8 V. Cheval. Apte: an algorithm for proving trace equivalence. In *Proc. TACAS'14*, 2014.
9 V. Cheval, H. Comon-Lundh, and S. Delaune. Trace equivalence decision: Negative tests and non-determinism. In *Proc. of CCS'11*. ACM Press, 2011.
10 V. Cheval and L. Hirschi. sources of APTE, 2015. `https://github.com/APTE/APTE`.
11 E. Clarke, S. Jha, and W. Marrero. Efficient verification of security protocols using partial-order reductions. *Int. Journal on Software Tools for Technology Transfer*, 4(2), 2003.
12 H. Comon-Lundh, V. Cortier, and E. Zalinescu. Deciding security properties for cryptographic protocols. Application to key cycles. *ACM Transactions on Computational Logic (TOCL)*, 11(4), 2010.
13 Cas JF Cremers and Sjouke Mauw. Checking secrecy by means of partial order reduction. In *System Analysis and Modeling*. Springer, 2005.
14 Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. A partial ordering semantics for ccs. *Theoretical Computer Science*, 75(3):223–262, 1990.
15 S. Delaune, S. Kremer, and M. Ryan. Verifying privacy-type properties of electronic voting protocols: A taster. In *Towards Trustworthy Elections – New Directions in Electronic Voting*, volume 6000. Springer, 2010.
16 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.
17 L. Hirschi. APTE with POR. `http://www.lsv.ens-cachan.fr/~hirschi/apte_por`.
18 J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proceedings of CCS'01*. ACM Press, 2001.
19 D. Miller and A. Saurin. From proofs to focused proofs: A modular proof of focalization in linear logic. In *Proc. of CSL'07*, volume 4646. Springer, 2007.
20 S. Mödersheim, L. Viganò, and D. Basin. Constraint differentiation: Search-space reduction for the constraint-based analysis of security protocols. *JCS*, 18(4), 2010.
21 D. Peled. Ten years of partial order reduction. In *Proc. of CAV'98*. Springer, 1998.
22 A. Tiu. Spec: `http://users.cecs.anu.edu.au/~tiu/spec/`, 2010.
23 A. Tiu and J. E. Dawson. Automating open bisimulation checking for the spi calculus. In *Proc. of CSF'10*. IEEE Comp. Soc. Press, 2010.