Noname manuscript No.
(will be inserted by the editor)

A decidable class of security protocols for both reachability and equivalence properties

Véronique Cortier · Stéphanie Delaune · Vaishnavi Sundararajan

Received: date / Accepted: date

Abstract We identify a new decidable class of security protocols, both for reachability and equivalence properties. Our result holds for an unbounded number of sessions and for protocols with nonces. It covers all standard cryptographic primitives. Our class sets up three main assumptions. *(i)* Protocols need to be "simple", meaning that an attacker can precisely identify from which participant and which session a message originates from. We also consider protocols with no else branches (only positive test). *(ii)* Protocols should be type-compliant, which is intuitively guaranteed as soon as two encrypted messages of the protocol cannot be confused. *(iii)* Finally, we define the notion of a dependency graph, which, given a protocol, characterises how actions depend on the other ones (both sequential dependencies and data dependencies are taken into account). Whenever the graph is acyclic, then the protocol falls into our class. We show that many protocols of the literature belong to our decidable class, including for example some of the protocols embedded in the biometric passport.

Keywords Security protocols · verification · privacy properties

1 Introduction

Security protocols are notoriously difficult to design and analyse. In this context, formal methods have proved their interest to help to detect automatically flaws and provide better security guarantees. For example, they have been used during the standardisation process to detect and correct flaws in the Transport Layer Security (TLS) 1.3 protocol [24]. In the context of voting, the production of symbolic proofs is now a legal requirement in Switzerland [2].

V. Cortier
LORIA, CNRS, France E-mail: veronique.cortier@loria.fr (corresponding author)

S. Delaune, V. Sundararajan
IRISA, CNRS & Univ Rennes, France E-mail: stephanie.delaune@irisa.fr

Symbolic models for security protocols abstract away how cryptographic primitives are implemented. They instead focus on the analysis of the flow of the protocols. Thanks to this level of abstraction, security analysis is amenable to automation. Several tools can now, given the abstract specification of a protocol, automatically find flaws or prove security. Examples of popular tools are ProVerif [7], Tamarin [32], Avispa [5], Maude-NPA [28], or Scyther [22]. However, even simple security properties like confidentiality are undecidable in general [27]. To retrieve decidability, one standard assumption is to bound the number of sessions, which corresponds to analysing the protocol when it is run a finite number of times. In that case, reachability properties like confidentiality and authentication properties are (co)-NP-complete [34]. Privacy properties like anonymity, vote secrecy or unlinkability are rather expressed as equivalence properties. For example, anonymity corresponds to the fact that an attacker should not be able to distinguish whether Alice is making a payment or Bob is making a payment. Such properties have been studied more recently but can also be shown to be decidable for a bounded number of sessions [6, 12] for a large class of cryptographic primitives and protocols. Several tools have even been proposed to decide privacy properties for a bounded number of sessions such as SPEC [23], APTE [11] and Akiss [9], and Sat-Equiv [20]. However, protocols are executed a large number of times in practice (think of the number of TLS connections within a day). Some tools such as ProVerif [7] or Scyther [22] can actually handle an unbounded number of sessions although they are not guaranteed to terminate. Tamarin [32] provides an automatic mode but often requires some help from the user. This is actually one of the main features of the tool: when it cannot conclude by itself, it offers an interactive mode and the possibility to state intermediary properties (lemmas). In practice, these tools work well, at least for reachability properties. So a remaining open problem for the last ten years is to characterise a decidable fragment of security protocols that captures most real protocols.

Related work. A few decidable classes of protocols have been identified for an unbounded number of sessions. Several of them consider protocols without nonces (see *e.g.* [27,17] for reachability properties and [14,15] for equivalence). However, protocols do use nonces in practice. If we restrict our attention to protocols with nonces, there is actually no decidability result for equivalence properties, except [16] which this paper builds upon and that we discuss later on. For reachability properties, and more precisely secrecy, Lowe [31] shows decidability provided that protocols rules obey a strict format (no ciphertext forwarding for example) and assuming that agents are able to check this format when they receive messages. Typically, this result assumes that an agent can never confuse a nonce with a key, an agent name, or a ciphertext. In [33], Ramanujam and Suresh obtain decidability assuming a rather severe tagging scheme, where each ciphertext has to include a fresh, shared session identifier. They do not cope with ciphertext forwarding. Dougherty and Guttman [26] have proposed a decidability result dedicated to Diffie-Hellman protocols. The result that is closest to ours is probably the one from Sybille Fröschle [29], who

has proposed a decidability result for the "leakiness" property and the class of well-founded protocols, with encryption and concatenation only (no signature nor hash). A protocol is secure w.r.t. leakiness if all data are either public or secret. In particular, it is not possible to prove security for protocols with temporary secrets, *e.g.* session identifiers that are not immediately revealed. Moreover, ciphertext forwarding is again prohibited and as in [31] a typed model is considered.

Our result relies on a small-attack property proved in [13]. This property says that for some class of protocols, if there is an attack, then there is a well-typed attack, that is an attack where all messages comply with the expected format. This nicely restricts the search space when looking for attacks. This result was already used in [19] to derive an efficient procedure for deciding trace equivalence, but for a bounded number of sessions.

Our contribution. We identify a new class of protocols with nonces, for which both reachability and trace equivalence are decidable, for an unbounded number of sessions. Our class covers all standard cryptographic primitives (deterministic and randomised symmetric and asymmetric encryptions, hashes, and signatures). Our class makes three main assumptions.

- *simple protocols:* We assume that each role of a protocol can be written as a succession of inputs and outputs on a dedicated channel. This corresponds to the idea that each process is identified by an IP address (address of the machine) and some protocol and session identifier so that sessions cannot be mixed up. We further assume that each role has no else branch, that is, makes only positive tests (e.g. equality between a received value and a previously sent nonce).
- *type compliance:* We assume that each encrypted message of the protocol can be given an expected format (formally, a type) so that any two unifiable encrypted messages have the same format. When necessary, this can usually be enforced by tagging messages, that is, adding some tag (e.g. a number) that avoids confusion between two different messages of the protocol. This is a good practice anyway that rules out many attacks. Note that the adversary can of course deviate from the expected format.
- *acyclic dependency graph:* We associate with each protocol its *dependency graph*. A node of this graph corresponds to an action (input or output). An edge corresponds to a dependency. There are two main kinds of dependencies: sequential dependencies (some action may only happen after other previous actions) and data dependencies (an input can be built using parts of messages occurring in some outputs). Our decidability result requires this dependency graph to be acyclic.

The two first conditions are easy to check and often met by protocols from the literature. Checking the last condition (acyclicity) requires the construction of the dependency graph, which can be done by applying a simple polynomial algorithm. Our experiments on a dozen of protocols of the literature show that many protocols of the literature have an acyclic graph, including for example

some of the protocols embedded in the biometric passport. Interestingly, our decidability result provides an explicit bound on the number of sessions: for each protocol that falls into our class, we can bound the number of sessions that need to be considered to find an attack. This bound is however very high and currently clearly out of reach of existing tools like DeepSec [12] or SAT-Equiv [20] for a bounded number of sessions.

Our decidability result is established in two main steps. First, we build on the small-attack property proved in [13]. This property says that for simple and type-compliant protocols, if there is an attack, then there is a well-typed attack, that is an attack where all messages comply with the expected format. We adapt this result to our context. We introduce the notion of an honest type, for elements that only appear in key position. We show that we can always consider an attack trace that is honest-free. Such a trace does not involve any constant of honest type, even though these constants are freely available to the attacker. Then, considering the dependency graph as sketched above, we show that a well-typed execution can be mapped to a path in the graph, hence bounding the length of the execution, hence the number of sessions.

We actually prove our result first for a simple definition of dependency graph that, however, often yields to cyclic graphs. We then provide a criterion in order to soundly remove edges from the graph, hence obtaining more likely acyclicity. Our approach is flexible enough to allow further refinements of the definition of the dependency graph, if needed.

Limitations. As mentioned earlier, our result does not cope with protocols with else branches, which prevents e.g. to fully model the BAC protocol used in the passport (we cannot model the fact that the protocol sends out error messages when some checks fail). As for many other decidability results, we consider all standard cryptographic primitives but we do not allow for a general equational theory nor for operators like Exclusive Or or modular exponentiation, which are notoriously hard to verify. We are not aware of any decidability result for an unbounded number of sessions (for protocols with nonces) that would cover these aspects.

Comparison with the earlier result [16]. We build upon an earlier result presented at CSF'15 [16] that establishes decidability of trace equivalence for protocols with symmetric encryption, with the same assumptions regarding simplicity, type-compliance, and acyclicity of the dependency graph. We extend this work to protocols with phases and to all standard primitives. This required in particular to provide a new characterisation of a sufficient set of tests for static equivalence, which in turn, yields a more subtle computation on the bound on the number of sessions. Moreover, decidability was established initially in [16] for trace equivalence only, we show how to adapt the approach to reachability.

2 Security protocol model

Security protocols are often modelled through a process algebra, in the spirit of the applied-pi calculus [3], that defines a small abstract programming language, well suited for protocols. Our result is built upon a typing result [13] that guarantees that if there is an attack, there is a well-typed attack, hence reducing the search space. Therefore, we consider here the process algebra used in [13] to established this typing result. Our decidability result covers all standard primitives. Thus, for the sake of readability, we instantiate the framework of [13] to the case of all the standard primitives while [13] consider a more general class of primitives.

2.1 Term algebra

As usual, messages are modelled by terms. Intuitively, terms are enough to represent how a message has been produced and how it can be decomposed. Private data, such as long-term and short-term keys or nonces, are represented through an infinite set of *names* $\mathcal{N}$. Public data, i.e. any data known by the attacker, such as agent names or attacker's nonces or keys, are modelled relying on an infinite set Σ_0 of *constants*. Constants are all initially known to the attacker. The set of constants is infinite to allow an attacker to use an arbitrary number of nonces and keys. We also consider two sets of *variables* $\mathcal{X}$ and $\mathcal{W}$. Variables in $\mathcal{X}$ typically refer to unknown parts of messages expected by protocol participants during the protocol execution, whereas variables in $\mathcal{W}$ are used to store messages learnt by the attacker so far. All these sets are assumed to be pairwise disjoint. *Data* are either constants, variables, or names.

Cryptographic primitives are represented by function symbols. We consider the following signature $\Sigma = \Sigma_{\mathsf{c}} \cup \Sigma_{\mathsf{d}} \cup \{\mathsf{check}\}$ where:

- $\Sigma_{\mathsf{c}} = \{\mathsf{aenc}, \mathsf{raenc}, \mathsf{senc}, \mathsf{rsenc}, \mathsf{pub}, \mathsf{ok}, \mathsf{sign}, \mathsf{vk}, \mathsf{hash}\} \cup \{\langle\,\rangle_n \mid n \geq 2\}$, and
- $\Sigma_{\mathsf{d}} = \{\mathsf{adec}, \mathsf{radec}, \mathsf{sdec}, \mathsf{rsdec}, \mathsf{getmsg}\} \cup \{\mathsf{proj}_j^n \mid n \geq 2 \text{ and } 1 \leq j \leq n\}$.

This signature comprises the standard primitives. Each symbol comes with an arity. The symbol aenc, of arity 2, stands for asymmetric encryption, with public key constructed using the function pub. A randomised version of asymmetric encryption is denoted raenc, of arity 3, that takes as additional argument the randomness used to compute the ciphertext. The functions adec and radec are the corresponding decryption functions (each of arity 2). Similarly, senc and rsenc denote respectively symmetric and randomised symmetric encryptions, with sdec and rsdec as corresponding decryption functions. The hash function is modelled by hash, of arity 1, while n-tuples are built using $\langle\,\rangle_n$ (of arity n). Then proj_j^n simply retrieves the jth component of a n-tuple. Finally, signatures are expressed with sign and corresponding verification function check and verification key built using vk. While check can intuitively be seen as a destructor (it allows to inspect a signature), note that the symbol check is neither a destructor nor a constructor. This allows us to devise finer results

later on, where we show that the adversary does not need to use check when computing terms (since it only yields the ok constant term).

The set of terms built from a signature $\mathcal{F}$ and a set of data D is denoted $\mathcal{T}(\mathcal{F}, D)$. We denote $vars(u)$ the set of variables that occur in a term u, and a term u is *ground* if it contains no variable. The application of a *substitution* σ to a term u is written $u\sigma$. We denote $dom(\sigma)$ its *domain* and $img(\sigma)$ its *image*. The *positions* of a term are defined as usual. Given a term u, we denote $\mathsf{root}(u)$ the function symbol occurring at position ϵ in u. The set $St(u)$ denotes the set of *subterms* of u, and $\mathsf{Cst}(u)$ denotes the set of constants from Σ_0 occurring in u. These notations are extended as expected to sets of terms. Two terms u_1 and u_2 are *unifiable* if there exists a substitution σ such that $u_1\sigma = u_2\sigma$.

We consider two *sorts*: atom and bitstring. The elements of sort atom represents atomic data like nonces or keys while bitstring models arbitrary messages. Names in $\mathcal{N}$ have sort atom, whereas constants in Σ_0 contains an infinite number of constants of both sorts. Any $\mathsf{f} \in \Sigma_\mathsf{c}$ comes with its sorted arity:

$$\begin{array}{ll} \mathsf{senc} & : \mathsf{bitstring} \times \mathsf{atom} \to \mathsf{bitstring} \\ \mathsf{aenc} & : \mathsf{bitstring} \times \mathsf{bitstring} \to \mathsf{bitstring} \\ \mathsf{rsenc} & : \mathsf{bitstring} \times \mathsf{atom} \times \mathsf{atom} \to \mathsf{bitstring} \\ \mathsf{raenc} & : \mathsf{bitstring} \times \mathsf{bitstring} \times \mathsf{atom} \to \mathsf{bitstring} \\ \mathsf{sign} & : \mathsf{bitstring} \times \mathsf{atom} \to \mathsf{bitstring} \end{array} \qquad \begin{array}{ll} \mathsf{ok} & : \to \mathsf{bitstring} \\ \mathsf{pub} & : \mathsf{atom} \to \mathsf{bitstring} \\ \mathsf{vk} & : \mathsf{atom} \to \mathsf{bitstring} \\ \mathsf{hash} & : \mathsf{bitstring} \to \mathsf{bitstring} \end{array}$$

$$\langle\rangle_n \ : \mathsf{bitstring} \times \ldots \times \mathsf{bitstring} \to \mathsf{bitstring} \text{ with } n \geq 2$$

We sometimes write $\langle\,\rangle$ instead of $\langle\,\rangle_n$ when n is clear from the context.

Given $D \subseteq \Sigma_0 \cup \mathcal{N} \cup \mathcal{X}$, the set $\mathcal{T}_0(\Sigma_\mathsf{c}, D)$ is the set of terms $t \in \mathcal{T}(\Sigma_\mathsf{c}, D)$ that are well-sorted, and such that for any $\mathsf{aenc}(u, v) \in St(t)$ (resp. $\mathsf{raenc}(u, v, r) \in St(t)$), $v = \mathsf{pub}(v')$ for some v'. Terms in $\mathcal{T}_0(\Sigma_\mathsf{c}, \mathcal{N} \cup \Sigma_0)$ are called *messages*. Intuitively, messages are terms with atomic keys, that is, asymmetric encryption can only be used with public keys of the form $\mathsf{pub}(k)$ where k is an atom and symmetric encryption can only be used with keys that are atoms. Since sorts are only used to define messages, destructors do not have sort. Terms with destructors can be "ill-sorted", only messages are required to be well-sorted.

The properties of the cryptographic primitives are reflected through the following convergent system of rewriting rules:

$$\begin{array}{rr} \mathsf{sdec}(\mathsf{senc}(x, y), y) \to x & \mathsf{rsdec}(\mathsf{rsenc}(x, y, z), y) \to x \\ \mathsf{adec}(\mathsf{aenc}(x, \mathsf{pub}(y)), y) \to x & \mathsf{radec}(\mathsf{raenc}(x, \mathsf{pub}(y), z), y) \to x \\ \mathsf{getmsg}(\mathsf{sign}(x, y)) \to x & \mathsf{check}(\mathsf{sign}(x, y), \mathsf{vk}(y)) \to \mathsf{ok} \end{array}$$

$$\mathsf{proj}_i^n(\langle x_1, \ldots, x_n\rangle_n) \ \to \ x_i \quad \text{with } n \geq 2 \text{ and } 1 \leq i \leq n$$

A term u can be rewritten into v if there is a position p in u, and a rewriting rule $\mathsf{g}(t_1, \ldots, t_n) \to t$ such that $u|_p = \mathsf{g}(t_1, \ldots, t_n)\theta$ for some substitution θ, and $v = u[t\theta]_p$, i.e. u in which the subterm at position p has been replaced by $t\theta$. Moreover, we assume that $t_1\theta, \ldots, t_n\theta$ as well as $t\theta$ are messages. As usual, we denote by $\to^*$ the reflexive-transitive closure of $\to$, and by $u{\downarrow}$ the normal form of a term u.

An attacker builds his own messages by applying function symbols to terms he already knows and that are available through variables in $\mathcal{W}$. Formally, a computation done by the attacker is a *recipe*, i.e. a term in $\mathcal{T}(\Sigma, \mathcal{W} \cup \Sigma_0)$.

Example 1 Let $u_1 = \mathsf{aenc}(\mathsf{sign}(k', sk_a), \mathsf{pub}(\mathsf{ek}_c))$ with $k', sk_a \in \mathcal{N}$ and $\mathsf{ek}_c \in \Sigma_0$. This term represents the encryption of $\mathsf{sign}(k', sk_a)$ (the signature of k' with the signing key sk_a) with the public key $\mathsf{pub}(\mathsf{ek}_c)$. Note that the private key associated to $\mathsf{pub}(\mathsf{ek}_c)$ is the public constant ek_c and is known to the attacker. The recipe $\mathsf{getmsg}(\mathsf{adec}(u_1, \mathsf{ek}_c))$ models the application of the decryption algorithm on top of u_1 using the key ek_c followed by the application of the algorithm that allows one to extract a message from a signature.

We need also to define the notion of key position on constructor terms. In particular, this notion will be used on messages (and types).

Given a set D of data, a position p of a constructor term $u \in \mathcal{T}(\Sigma_\mathsf{c}, D)$ is a *key position* if either $p = q.1$ for some q such that $\mathsf{root}(u|_q) = \{\mathsf{pub}, \mathsf{vk}\}$; or $p = q.2$ for some q such that $\mathsf{root}(u|_q) \in \{\mathsf{senc}, \mathsf{rsenc}, \mathsf{sign}\}$. We will denote $\mathsf{KP}(u)$ the set of all key positions of a term u. We denote by $\mathsf{K}(u)$ the subterms of u occurring at a key position in u, i.e. $\mathsf{K}(u) = \{u' \mid u' = u|_p \text{ for some } p \in \mathsf{KP}(u)\}$.

Example 2 Consider the term $u_1 = \mathsf{aenc}(\mathsf{sign}(k', sk_a), \mathsf{pub}(\mathsf{ek}_c))$ and $u_2 = \mathsf{pub}(\mathsf{ek}_c)$. We have that $\mathsf{KP}(u_1) = \{21, 12\}$, and $\mathsf{KP}(u_2) = \{1\}$.

2.2 Process algebra

We assume an infinite set $\mathcal{C}h = \mathcal{C}h_0 \uplus \mathcal{C}h^{\mathsf{fresh}}$ of *channels* used to communicate, where $\mathcal{C}h_0$ and $\mathcal{C}h^{\mathsf{fresh}}$ are infinite and disjoint. Intuitively, channels of $\mathcal{C}h^{\mathsf{fresh}}$ will be used to instantiate channels when they are generated during the execution of a protocol. They should not be part of a protocol specification. We also assume an infinite set $\mathcal{L}$ used to label input and output actions of processes. Protocols are modelled through processes built by the following grammar:

$$
\begin{array}{rcl}
P, Q & := & 0 \\
& \mid & \mathsf{in}^\ell(c, u).P \\
& \mid & \mathsf{out}^\ell(c, u).P \\
& \mid & \mathsf{new}\ n.P \\
& \mid & (P \mid Q) \\
& \mid & !P \\
& \mid & \mathsf{new}\ c'.\mathsf{out}(c, c').P \\
& \mid & i : P
\end{array}
$$

where $u \in \mathcal{T}(\Sigma_\mathsf{c}, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$, $n \in \mathcal{N}$, $c, c' \in \mathcal{C}h$, $\ell \in \mathcal{L}$, and $i \in \mathbb{N}$. The process 0 simply represents the null process. A process $\mathsf{in}^\ell(c, u).P$ will receive a message of the form u on channel c, while process $\mathsf{out}^\ell(c, u).P$ emits message u on channel c. Then $\mathsf{new}\ c'.\mathsf{out}(c, c').P$ is a special construction that allows one to create a new channel c', provided it is immediately emitted on a public

channel c. This way, we allow an arbitrary number of public channels but disallow private ones. Then, as usual, $P \mid Q$ is the parallel composition of P and Q, while $!P$ represents the unlimited replication of P. Finally, $i : P$ denotes that the process P will be executed at phase i. Phases are used to model protocols that are inherently divided into several steps, such as e-voting protocols (with setup, voting, and tallying phases). They are also convenient to model several security properties expressed as a game where, e.g., the attacker is first given the opportunity to interact with the protocol, and is then given a real or random key and he has to distinguish the two cases.

We denote $fv(\mathcal{P})$ the variables occurring in $\mathcal{P}$ that are not bound by an input and we assume w.l.o.g. that variables are bound at most once. Note that once a variable is bound by an input, it may be later used in an input as a filtering argument. For example, the variable x is bound only once in $\mathsf{in}(c, x).\mathsf{in}(c, x)$. This models the fact that the process will first input any message but will then expect exactly the same one. In contrast, the variable x is bound twice in $\mathsf{in}(c, x) \mid \mathsf{in}(c, x)$, and we will not consider such a process. We will assume instead that variables have been properly renamed. We denote $\mathsf{phase}(\ell)$ the integer corresponding to the phase at which the action labelled ℓ occurs.

Example 3 We consider a variant of the Denning-Sacco protocol with signatures as given in [8]. The protocol aims at ensuring the secrecy of the message m exchanged encrypted with the symmetric key k. This key is freshly generated by A and sent to B in the first message. It can be informally described as follows:

$$A \to B : \mathsf{aenc}(\mathsf{sign}(k, \mathsf{priv}_A), \mathsf{pub}_B)$$
$$B \to A : \mathsf{senc}(m, k)$$

where pub_B is the public encryption key of the agent B, and priv_A is the private signing key of the agent A. This is a slight variant of the original protocol proposed by Denning and Sacco in which the identity of the agents A and B appear inside the signature [25]. This variant is vulnerable to an attack regarding the secrecy of the key k and the message m that will be explained in Example 4.

We model this protocol in our formalism through the two processes P_A and P_B representing respectively the role of A and the role of B.

$$P_A = \mathsf{new}\ k.\, \mathsf{out}^{\ell_1}(c_A, \mathsf{aenc}(\mathsf{sign}(k, sk_a), \mathsf{pub}(ek_b))).\ \mathsf{in}^{\ell_2}(c_A, \mathsf{senc}(x_A, k))$$

$$P_B = \mathsf{in}^{\ell_3}(c_B, \mathsf{aenc}(\mathsf{sign}(x_B, sk_a), \mathsf{pub}(ek_b))).\ \mathsf{new}\ m.\, \mathsf{out}^{\ell_4}(c_B, \mathsf{senc}(m, x_B))$$

We have that k, sk_a, ek_b, and m are names, whereas x_A and x_B are variables. The name sk_a represents the signing private key of the agent a whose associated verification key is $\mathsf{vk}(sk_a)$, and ek_b is the private encryption key of the agent b and the associated public key is $\mathsf{pub}(ek_b)$.

Then, the protocol is modelled by the parallel composition of these two processes P_A and P_B together with a process P_K that models the initial knowledge of the attacker. More precisely, P_K reveals the public encryption keys and the verification keys to the attacker during an initialisation phase.

$$P_{\mathsf{DS}} = 0 : P_K \;\mid\; 1 : \big(!\mathsf{new}\, c_A.\mathsf{out}(c_1, c_A).P_A \mid !\mathsf{new}\, c_B.\mathsf{out}(c_2, c_B).P_B\big)$$

where

$$\begin{aligned} P_K = {} & \mathsf{out}^{\ell_1^K}(c_0, \mathsf{pub}(ek_b)).\mathsf{out}^{\ell_2^K}(c_0, \mathsf{vk}(sk_b)). \\ & \mathsf{out}^{\ell_3^K}(c_0, \mathsf{pub}(ek_a)).\mathsf{out}^{\ell_4^K}(c_0, \mathsf{vk}(sk_a)) \end{aligned}$$

We may want to consider a different scenario taking into account the presence of a dishonest agent c. We give below the process P'_A that corresponds to the role A played by a with a dishonest agent c. Below, k' is a name, x'_A a variable, and ek_c is a (public) constant from Σ_0.

$$\begin{aligned} P'_A = {} & \mathsf{new}\ k'.\, \mathsf{out}^{\ell'_1}(c'_A, \mathsf{aenc}(\mathsf{sign}(k', sk_a), \mathsf{pub}(\mathsf{ek}_c))). \\ & \mathsf{in}^{\ell'_2}(c'_A, \mathsf{senc}(x'_A, k')).0 \end{aligned}$$

The one session of the initiatior role played by a with c and one session of the responder role played by b with a is modelled by the process P'_{DS} given below. For the sake of simplicity, we only consider one session of each role.

$$P'_{\mathsf{DS}} = 0 : P_K \;\mid\; 1 : \big(P'_A \mid P_B\big)$$

Note that the decryption key ek_c of the agent c is modelled as a public constant and is thus implicitly known by the attacker. We do not need to reveal it explicitly.

2.3 Semantics

The operational semantics of a process is defined using a relation over configurations. A *configuration* is a tuple $(\mathcal{P}; \phi; \sigma; i)$ with $i \in \mathbb{N}$ such that:

- $\mathcal{P}$ is a multiset of processes (not necessarily ground);
- $\phi = \{\mathsf{w}_1 \triangleright m_1, \ldots, \mathsf{w}_n \triangleright m_n\}$ is a *frame*, *i.e.* a substitution where $\mathsf{w}_1, \ldots, \mathsf{w}_n$ are variables in $\mathcal{W}$, and $m_1, \ldots, m_n$ are messages;
- σ is a substitution such that $fv(\mathcal{P}) \subseteq dom(\sigma)$, and $img(\sigma)$ are messages.

Intuitively, $\mathcal{P}$ represents the processes that still remain to be executed; ϕ represents the sequence of messages that have been learnt so far by the attacker, σ stores the value of the variables that have already been instantiated, and i is an integer that indicates the current phase. A configuration $(\mathcal{P}; \phi; \sigma; i)$ such that $\phi = \sigma = \emptyset$ and $i = 0$ is called *initial*.

The operational semantics of a configuration are induced by the relation $\xrightarrow{\alpha}$ over configurations and is defined in Figure 1. The relation $\xrightarrow{\alpha_1 \ldots \alpha_n}$ between configurations (where $\alpha_1 \ldots \alpha_n$ is a sequence of actions) is defined as the transitive closure of $\xrightarrow{\alpha}$. Given a sequence of observable actions tr, and

In

$(i : \mathsf{in}^\ell(c,u).P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\mathsf{in}^\ell(c,R)} (i : P \uplus \mathcal{P}; \phi; \sigma \uplus \sigma_0; i)$
where R is a recipe such that $R\phi\!\downarrow$ is a message, and $R\phi\!\downarrow = (u\sigma)\sigma_0$ for σ_0 with $dom(\sigma_0) = vars(u\sigma)$.

Out

$(i : \mathsf{out}^\ell(c,u).P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\mathsf{out}^\ell(c,\mathsf{w})} (i : P \uplus \mathcal{P}; \phi \cup \{\mathsf{w} \triangleright u\sigma\}; \sigma; i)$
with w a fresh variable from $\mathcal{W}$, and $u\sigma$ is a message.

New
$(i : \mathsf{new}\, n.P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\tau} (i : P\{^m/_n\} \uplus \mathcal{P}; \phi; \sigma; i)$ with $m \in \mathcal{N}$ fresh.

Par $(i : (P \mid Q) \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\tau} (i : P \uplus i : Q \uplus \mathcal{P}; \phi; \sigma; i)$

Rep
$(i : !P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\tau} (i : P' \uplus i : !P \uplus \mathcal{P}; \phi; \sigma; i)$ with P' a copy of P where variables bound in the inputs are α-renamed.

Out-Ch

$(i : \mathsf{new}\, c'.\mathsf{out}(c,c').P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\mathsf{out}(c,c'')} (i : P\{^{c''}/_{c'}\} \uplus \mathcal{P}; \phi; \sigma; i)$
with c'' a fresh channel.

Move $(\mathcal{P}; \phi; \sigma; i) \xrightarrow{\mathsf{phase}\ i'} (\mathcal{P}; \phi; \sigma; i')$ with $i' > i$.

Phase $(i' : i'' : P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\tau} (i'' : P \uplus \mathcal{P}; \phi; \sigma; i)$

Null $(i : 0 \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\tau} (\mathcal{P}; \phi; \sigma; i)$

Fig. 1 Semantics of our process algebra

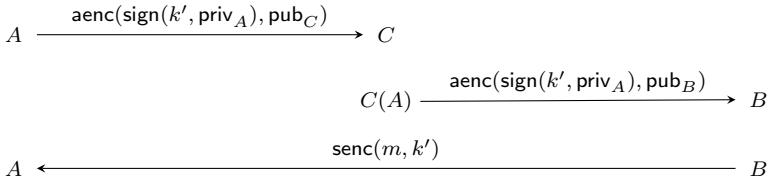

Fig. 2 Attack trace.

two configurations $\mathcal{K}$ and $\mathcal{K}'$, we write $\mathcal{K} \xRightarrow{\mathsf{tr}} \mathcal{K}'$ if there exists a sequence $\alpha_1 \dots \alpha_n$ such that $\mathcal{K} \xrightarrow{\alpha_1 \dots \alpha_n} \mathcal{K}'$ and tr is obtained from $\alpha_1 \dots \alpha_n$ by erasing all occurrences of τ and erasing all labels: $\mathsf{out}^\ell(c,\mathsf{w})$ is replaced by $\mathsf{out}(c,\mathsf{w})$ and $\mathsf{in}^\ell(c,R)$ is replaced by $\mathsf{in}(c,R)$. Labels may also be omitted when writing $\mathcal{K} \xrightarrow{\alpha_1 \dots \alpha_n} \mathcal{K}'$ when they are not relevant.

Definition 1 Given a configuration $\mathcal{K} = (\mathcal{P}; \phi; \sigma; i)$, we denote $\mathsf{trace}(\mathcal{K})$ the set of traces defined as follows:

$$\mathsf{trace}(\mathcal{K}) = \{(\mathsf{tr}, \phi') \mid \mathcal{K} \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi'; \sigma'; i') \text{ for some configuration } (\mathcal{P}; \phi'; \sigma'; i')\}.$$

Example 4 Continuing Example 3, we consider the initial configuration $\mathcal{K}_0 = (P'_{\mathsf{DS}}; \emptyset; \emptyset; 0)$. As mentioned in Example 3, this variant of the Denning-Sacco

protocol is vulnerable to an attack depicted in Figure 2. The attack relies on the fact that agent A starts a session with the dishonest agent C leading the agent C to know a valid signature $\mathsf{sign}(k', \mathsf{priv}_A)$ on a key k' that he can deduce. Then, agent C can pretend to be A and send this signature to B encrypted with the public key of B. Agent B will then accept this message and send his private message m intended to A, encrypted with the key k' since he believes that he shares this key with A (whereas this key is known by the attacker). This attack is reflected by the following sequence tr_0:

$$\mathsf{out}(c_0, \mathsf{w}_1).\mathsf{phase}\,1.\mathsf{out}(c'_A, \mathsf{w}_2).\mathsf{in}(c_B, R_0).\mathsf{out}(c_B, \mathsf{w}_3)$$

where $R_0 = \mathsf{aenc}(\mathsf{adec}(\mathsf{w}_2, \mathsf{ek}_c), \mathsf{w}_1)$ corresponds to the message manipulation done by the attacker: he decrypts the message received from A and re-encrypt it with the public key of B. This sequence of actions leads to the frame ϕ_0 defined as follows:

$$\phi_0 = \{\mathsf{w}_1 \triangleright \mathsf{pub}(ek_b),\ \mathsf{w}_2 \triangleright \mathsf{aenc}(\mathsf{sign}(k', sk_a), \mathsf{pub}(\mathsf{ek}_c)),\ \mathsf{w}_3 \triangleright \mathsf{senc}(m, k')\}.$$

We have that $(\mathsf{tr}_0, \phi_0) \in \mathsf{trace}(\mathcal{K}_0)$.

2.4 Trace equivalence

Privacy properties are often modelled as equivalence of processes. We first start with the notion of *static equivalence*. Intuitively, two frames, that is two sequences of messages, are in static equivalence when an attacker cannot tell them apart.

Definition 2 Two frames ϕ_1 and ϕ_2 are in *static inclusion*, written $\phi_1 \sqsubseteq_s \phi_2$, when $dom(\phi_1) = dom(\phi_2)$, and:

- for any recipe R, we have that $R\phi_1{\downarrow}$ is a message implies that $R\phi_2{\downarrow}$ is a message; and
- for any recipes R, R' such that $R\phi_1{\downarrow}$, $R'\phi_1{\downarrow}$ are messages, we have that: $R\phi_1{\downarrow} = R'\phi_1{\downarrow}$ implies $R\phi_2{\downarrow} = R'\phi_2{\downarrow}$.

They are in *static equivalence*, written $\phi_1 \sim_s \phi_2$, if $\phi_1 \sqsubseteq_s \phi_2$ and $\phi_2 \sqsubseteq_s \phi_1$.

Example 5 Continuing Example 4, we consider the two following frames:

- $\phi_1 = \{\mathsf{w}_1 \triangleright \mathsf{pub}(ek_b),\ \mathsf{w}_2 \triangleright \mathsf{aenc}(\mathsf{sign}(k', sk_a), \mathsf{pub}(\mathsf{ek}_c)),\ \mathsf{w}_3 \triangleright \mathsf{senc}(\mathsf{m}_1, k')\}$;
- $\phi_2 = \{\mathsf{w}_1 \triangleright \mathsf{pub}(ek_b),\ \mathsf{w}_2 \triangleright \mathsf{aenc}(\mathsf{sign}(k', sk_a), \mathsf{pub}(\mathsf{ek}_c)),\ \mathsf{w}_3 \triangleright \mathsf{senc}(\mathsf{m}_2, k')\}$

where m_1 and m_2 are public constants from Σ_0.

We have that $\mathsf{sdec}(\mathsf{w}_3, R_k)\phi_1{\downarrow} = \mathsf{m}_1\phi_1{\downarrow}$ where $R_k = \mathsf{getmsg}(\mathsf{adec}(\mathsf{w}_2, \mathsf{ek}_c))$. This equality does not hold in ϕ_2, hence ϕ_1 and ϕ_2 are not in static equivalence.

We can now define equivalence over processes. Intuitively, equivalence is meant to model the fact that an attacker cannot distinguish between two processes P and Q. We consider here the notion of trace equivalence stating that any observable sequence of actions in P can also be observed in Q and vice-versa and that the resulting sequences of messages are in static equivalence.

Definition 3 A configuration $\mathcal{K}$ is trace included in a configuration $\mathcal{K}'$, written $\mathcal{K} \sqsubseteq_t \mathcal{K}'$, if for every $(\mathsf{tr}, \phi) \in \mathsf{trace}(\mathcal{K})$, there exist $(\mathsf{tr}', \phi') \in \mathsf{trace}(\mathcal{K}')$ such that $\mathsf{tr} = \mathsf{tr}'$, and $\phi \sqsubseteq_s \phi'$. They are in trace equivalence, written $\mathcal{K} \approx_t \mathcal{K}'$, if $\mathcal{K} \sqsubseteq_t \mathcal{K}'$ and $\mathcal{K}' \sqsubseteq_t \mathcal{K}$.

Assume given two configurations $\mathcal{K}$ and $\mathcal{K}'$ such that $\mathcal{K} \not\sqsubseteq_t \mathcal{K}'$. A witness of this non-inclusion is a trace tr for which there exists ϕ such that $(\mathsf{tr}, \phi) \in \mathsf{trace}(\mathcal{K})$ and:

- either there is no ϕ' such that $(\mathsf{tr}, \phi') \in \mathsf{trace}(\mathcal{K}')$;
- or $\phi \not\sqsubseteq_s \phi'$ for any ϕ' such that $(\mathsf{tr}, \phi') \in \mathsf{trace}(\mathcal{K}')$.

This notion of trace equivalence slightly differs from the original one introduced by Abadi and Fournet in [3]. In the original definition, frames are required to be in static equivalence $\phi \sim_s \phi'$ instead of static inclusion $\phi \sqsubseteq_s \phi'$. Actually, these two notions of equivalence coincide for determinate protocols [10]. Intuitively, a protocol can be modelled as a determinate process if no agent makes a non deterministic choice and if all agents emit on distinguishable channels.

Definition 4 A configuration $\mathcal{K}$ is *determinate* if whenever $\mathcal{K} \stackrel{\mathsf{tr}}{\Longrightarrow} \mathcal{K}_1$ and $\mathcal{K} \stackrel{\mathsf{tr}}{\Longrightarrow} \mathcal{K}_2$ for some tr, $\mathcal{K}_1 = (\mathcal{P}_1; \phi_1; \sigma_1; i_1)$, and $\mathcal{K}_2 = (\mathcal{P}_2; \phi_2; \sigma_2; i_2)$ we have that $\phi_1 \sim_s \phi_2$.

Hence, our reduction and decidability results are applicable to standard equivalence for determinate protocols. For protocols that are not determinate, our results still hold but only apply for our own notion of trace equivalence that is slightly weaker than standard trace equivalence.

Example 6 Continuing Example 3, we consider the protocol P'_{DS} that models a role of A played by a with c, and a role of B played by b with a. To model the fact that the message m sent by B for A should remain secret, we require that even if the attacker knows two possible values for m, say m_1 and m_2, he should not be able to distinguish which of these values has been exchanged. The corresponding processes are P^1_{DS} and P^2_{DS} which are P'_{DS} in which m has been replaced respectively by m_1 and m_2. Then, we consider the configuration $\mathcal{K}_1 = (P^1_{\mathsf{DS}}; \emptyset; \emptyset; 0)$ and $\mathcal{K}_2 = (P^2_{\mathsf{DS}}; \emptyset; \emptyset; 0)$. We can show that $\mathcal{K}_1 \not\sqsubseteq_t \mathcal{K}_2$ (and $\mathcal{K}_2 \not\sqsubseteq_t \mathcal{K}_1$) since m is not strongly secret due to the attack depicted in Figure 2. This is exemplified by the trace tr_0 given in Example 4 which leads to the frame ϕ_1 (resp. ϕ_2) starting with the configuration $\mathcal{K}_1$ (resp. $\mathcal{K}_2$), i.e. $(\mathsf{tr}_0, \phi_1) \in \mathsf{trace}(\mathcal{K}_1)$ and $(\mathsf{tr}_0, \phi_2) \in \mathsf{trace}(\mathcal{K}_2)$. We have that $\phi_1 \not\sqsubseteq_s \phi_2$ (and $\phi_2 \not\sqsubseteq_s \phi_1$ as well) as explained in Example 5.

To avoid this attack, names of the agents should be included in the signature sent by A to B.

2.5 Simple processes

We consider a fragment of processes, the class of *simple processes*, similar to the one introduced in [16]. This corresponds to protocols where each role can

be seen as a sequence of inputs and outputs on a specific channel, distinct for each session.

Definition 5 A *simple protocol* P is a ground process of the form

$$!\mathsf{new}\ c'_1.\mathsf{out}(c_1, c'_1).B_1 \mid \ldots \mid !\mathsf{new}\ c'_m.\mathsf{out}(c_m, c'_m).B_m \mid B_{m+1} \mid \ldots \mid B_{m+n}$$

where each B_i with $1 \leq i \leq m$ (resp. $m < i \leq m+n$) is a ground process on channel c'_i (resp. c_i) built using the following grammar:

$$B := 0 \mid \mathsf{in}^{\ell}(c'_i, u).B \mid \mathsf{out}^{\ell'}(c'_i, u).B \mid \mathsf{new}\, n.B \mid j : B$$

where $u \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$, and $j \in \mathbb{N}$. Moreover, we assume that the channel names $c_1, \ldots, c_n, c_{n+1}, \ldots, c_{n+m}$ are pairwise distinct.

Example 7 Note that the processes P_{DS} and P'_{DS} given in Example 3 are simple.

We sometime denote by P the initial configuration $(\{P\}; \emptyset; \emptyset; 0)$, and we denote by $\mathsf{Terms}(P)$ the set of all terms occurring in P, i.e. terms u such that $\mathsf{out}(c, u)$ (resp. $\mathsf{in}(c, u)$) occurs in P.

Given a simple protocol P, and $\ell_1, \ell_2 \in \mathcal{L}(P)$, we say that ℓ_2 *directly follows* ℓ_1 in P if both actions are in sequence in the description of P, with ℓ_2 after ℓ_1, and no other visible action in between. When some other visible actions occur between ℓ_1 and ℓ_2, we simply say that ℓ_2 *follows* ℓ_1.

Simple protocols are determinate. Actually, given $\mathcal{K}_1$ and $\mathcal{K}_2$ such that $P \overset{\mathsf{tr}}{\Longrightarrow} \mathcal{K}_1 = (\mathcal{P}_1; \phi_1; \sigma_1; i_1)$ and $P \overset{\mathsf{tr}}{\Longrightarrow} \mathcal{K}_2 = (\mathcal{P}_2; \phi_2; \sigma_2; i_2)$ for some tr, we have that ϕ_1 and ϕ_2 are equal up to some α-renaming. Thus, considering two simple protocols P and Q, a witness of non-inclusion for $P \sqsubseteq_t Q$ is a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ such that:

- either there is no ϕ' such that that $(\mathsf{tr}, \phi') \in \mathsf{trace}(Q)$;
- or in case such a ϕ' exists, we have that $\phi \not\sqsubseteq_s \phi'$.

Why do we consider simple protocols? This assumption is very helpful to prove trace equivalence. It allows to precisely map each action of P to an action of Q thanks to the fact that given a channel name and a trace, only one process can produce this action. We could probably relax this assumption when we consider reachability properties only. However, the proofs are easier when assuming simple protocols.

3 First decidability results for reachability and trace equivalence

In this section, we state our first decidability result for reachability and trace equivalence. We introduce the notion of type-compliance, which intuitively enforces that messages that can be confused within a protocol must have the same type. We also define the dependency graph of a protocol, which reflects dependencies between messages of a protocol. We show that reachability properties as well as trace equivalence are both decidable for simple protocols that are type-compliant and that have an acyclic dependency graph.

3.1 Type-compliance

We consider here typing systems that preserve the structure of terms. They are defined as follows:

Definition 6 A typing system is a pair (Δ_0, δ_0) where Δ_0 is a set of elements called *initial types*, and δ_0 is a function mapping data in $\Sigma_0 \uplus \mathcal{N} \uplus \mathcal{X}$ to types τ generated using the following grammar:

$$\tau, \tau_1, \tau_2 = \tau_0 \mid \mathsf{f}(\tau_1, \ldots, \tau_n) \text{ with } \mathsf{f} \in \Sigma_{\mathsf{c}} \text{ and } \tau_0 \in \Delta_0$$

We further assume the existence of an infinite number of constants in Σ_0 (resp. variables in $\mathcal{X}$, names in $\mathcal{N}$) of any type. Then, δ_0 is extended to constructor terms as follows:

$$\delta_0(\mathsf{f}(t_1, \ldots, t_n)) = \mathsf{f}(\delta_0(t_1), \ldots, \delta_0(t_n)) \text{ with } \mathsf{f} \in \Sigma_{\mathsf{c}}.$$

A type can be seen as a term, we will assume that initial types are of sort atom, and we extend the notion of being well-sorted from terms to types.

Consider a configuration $\mathcal{K}$ and a typing system (Δ_0, δ_0). An execution $\mathcal{K} \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ is *well-typed* if σ is a well-typed substitution, i.e. every variable of its domain has the same type as its image.

Example 8 We continue our running example with simple protocols P^1_{DS} and P^2_{DS} as defined in Example 7. We consider the typing system generated from the set $\mathcal{T}_{\mathsf{DS}} = \{\tau_{sk}, \tau_{skc}, \tau_{ek}, \tau_{ekc}, \tau_k, \tau_m\}$ of initial types, and the function δ_{DS} that associates the expected type to each constant/name ($\delta_{\mathsf{DS}}(sk_a) = \delta_{\mathsf{DS}}(sk_b) = \tau_{sk}$, $\delta(\mathsf{sk}_c) = \tau_{skc}$, $\delta_{\mathsf{DS}}(m) = \delta_{\mathsf{DS}}(\mathsf{m}_1) = \delta_{\mathsf{DS}}(\mathsf{m}_2) = \tau_m$, $\delta_{\mathsf{DS}}(k') = \tau_k$, ...), and the following types to the variables: $\delta_{\mathsf{DS}}(x'_A) = \tau_m$, and $\delta_{\mathsf{DS}}(x_B) = \tau_k$.

We now introduce the notion of *encrypted subterms*. We write $ESt(t)$ for the set of encrypted subterms of t, i.e. the set of subterms that are not headed by a tuple operator.

$$ESt(t) = \{u \in St(t) \mid \mathsf{root}(u) \in \{\mathsf{aenc}, \mathsf{raenc}, \mathsf{senc}, \mathsf{rsenc}, \mathsf{sign}, \mathsf{hash}, \mathsf{pub}, \mathsf{vk}\}\}$$

This notion is extended as expected to sets of terms, processes, frames, and initial configurations.

We will require that any two unifiable encrypted subterms appearing in the specification of a protocol have the same type. For this, we need to rename variables under replication since, intuitively, such variables are refreshed at each new session. Formally, given a simple protocol P, we define its 2-unfolding, denoted $\mathsf{unfold}^2(P)$, to be the protocol such that every occurrence of a process $!Q$ in P is replaced by $Q \mid Q$, assuming that a type-preserving α-renaming is performed on one copy to avoid variables and names capture. Note that, in case P is replication-free, we have that $\mathsf{unfold}^2(P) = P$.

Definition 7 A simple protocol P is *type-compliant* w.r.t. a typing system (Δ_0, δ_0) if for every $t, t' \in ESt(\mathsf{unfold}^2(P))$ we have that:

$$t \text{ and } t' \text{ unifiable implies that } \delta_0(t) = \delta_0(t').$$

Example 9 Consider again our running example with P^1_{DS}. We have that P^1_{DS} is type-compliant w.r.t. the typing system given in Example 8. Indeed, the encrypted subterms of P^1_{DS} are:

- $\mathsf{pub}(ek_b)$, $\mathsf{pub}(ek_a)$, $\mathsf{pub}(\mathsf{ek}_c)$, $\mathsf{vk}(sk_b)$, and $\mathsf{vk}(sk_a)$;
- $\mathsf{sign}(k', sk_a)$ and $\mathsf{sign}(x_B, sk_a)$;
- $\mathsf{aenc}(\mathsf{sign}(k', sk_a), \mathsf{pub}(\mathsf{ek}_c))$ and $\mathsf{aenc}(\mathsf{sign}(x_B, sk_a), \mathsf{pub}(ek_b))$;
- $\mathsf{senc}(x'_A, k')$ and $\mathsf{senc}(\mathsf{m}_1, x_B)$.

It is easy to verify that each pair of unifiable encrypted subterms have the same type. To check type-compliance, we have to consider the encrypted subterms occurring in $\mathsf{unfold}^2(P^1_{\mathsf{DS}})$ but since our process P^1_{DS} does not contain any replication, we have that $\mathsf{unfold}^2(P^1_{\mathsf{DS}}) = P^1_{\mathsf{DS}}$.

On this example, type-compliance also holds when considering a more complex scenario with replication, and actually we can also consider a more refined typing system where $\delta_{\mathsf{DS}}(sk_a) = \tau_{ska}$ and $\delta_{\mathsf{DS}}(sk_b) = \tau_{skb}$.

3.2 Honest type

We introduce the notion of *honest type* that will intuitively guarantee that a term of honest type is never revealed. It will typically be used for long term secret keys.

Definition 8 Consider a simple protocol P and a typing system (Δ_0, δ_0). An atomic type τ_h is *honest for* P if for any $u \in \mathsf{Terms}(P)$, and any position p such that $(u\delta_0)|_p = \tau_h$, we have that p is a key position of $u\delta_0$, that is, $p \in \mathsf{KP}(u\delta_0)$; and $\tau_h \neq \delta_0(a)$ for any constant/variable a occurring in P.

We say that a term is *honest-free* if it does not contain any constant of honest type. This notion is lifted to traces, frames, and configurations as expected.

Intuitively, a type is honest for P if terms of that type only occur at key positions in P. Note in particular that constants occurring in a protocol have a type that cannot be honest. Indeed, constants are actually public to the attacker. Note also that terms occurring in a configuration of the form $(P; \emptyset; \emptyset; 0)$ are, by definition, honest-free. They may only contain names of honest type. In what follows, we will see that it is sufficient to consider honest-free traces.

Example 10 Continuing our running example, we have that the atomic types τ_{sk} and τ_{ek} are honest types. Indeed, we have no constant/variable having such a type occurring in P^1_{DS} (resp. P^2_{DS}). Moreover, the only occurrence of a term having such a type in P^1_{DS} is indeed in key position, i.e. under a symbol pub, vk, or as a second argument of the symbol sign. Note that τ_{ekc} can not be considered as an honest type since ek_c is a constant of type τ_{ekc}.

We can show that, in well-typed executions, names of honest type are never revealed to the attacker.

Lemma 1 *Let P be a simple protocol, (Δ_0, δ_0) be a typing system, and $P \xrightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ be a well-typed execution. Let τ_h be an honest type, n be a name such that $\delta_0(n) = \tau_h$. We have that $R\phi\downarrow \neq n$ for any recipe R.*

Proof Assume towards a contradiction that there exists R such that $R\phi\downarrow = n$. Thanks to Lemma 4 (this lemma is proved later on - see page 21), we know that there exists such a recipe R that is simple, and since n is a name, we have that R is a destructor-only recipe. Let w be the variable occurring at the leftmost position of R. We have that n occurs at a plaintext (extractable) position p in $\mathsf{w}\phi$. By construction of the frame ϕ, it must be the case that $\mathsf{w}\phi = u\sigma$ where u is a term appearing in P in some output. Since the execution is well-typed, we know that $\delta_0(\mathsf{w}\phi) = \delta_0(u\sigma) = \delta_0(u)$. This means that p corresponds to a plaintext position of $\delta_0(u)$ and thus $\delta_0(n)$ is not an honest type by definition. □

3.3 Dependency graph

The type of a term will be used to compute on which other terms it can depend. For example, a term of composed type $\mathsf{senc}(\tau_1, \tau_2)$ may be obtained by composition from a term of type τ_1 and a term of type τ_2.

Formally, we define two functions ρ_{out} and ρ_{in}. The function ρ_{out}, computes the types of the terms that can be extracted from a term of type τ while ρ_{in} computes the set of types that could be used to build a term of type τ. More precisely, ρ_{in} returns a set of types whereas ρ_{out} returns a set of tuples of the form $(\tau, p)\#(S; A)$ where τ is a type, p a position, and S and A are multisets of terms. Intuitively, S collects the types of symmetric keys while A collects the types of asymmetric keys.

Definition 9 Given a well-sorted type τ, we define $\rho_{\mathsf{out}}(\tau)$ to be $\rho_{\mathsf{out}}(\tau, \epsilon, \emptyset, \emptyset)$ where $\rho_{\mathsf{out}}(\tau, p, S, A)$ is recursively defined as the set $\{(\tau, p)\#(S; A)\} \cup E$ where E is defined as follows:

- $E = \emptyset$ when τ is an initial type or $\mathsf{root}(\tau) \in \{\mathsf{pub}, \mathsf{vk}, \mathsf{hash}\}$;
- $E = \bigcup_{i=1}^{n} \rho_{\mathsf{out}}(\tau_i, p.i, S, A)$ when $\tau = \langle \tau_1, \ldots, \tau_n \rangle$;
- $E = \rho_{\mathsf{out}}(\tau_1, p.1, S, A)$ when $\tau = \mathsf{sign}(\tau_1, \tau_2)$;
- when $\mathsf{root}(\tau) \in \{\mathsf{senc}, \mathsf{rsenc}\}$, $E = \emptyset$ if $\tau|_2$ is an honest type, and $E = \rho_{\mathsf{out}}(\tau|_1, p.1, S \uplus \{\tau|_2\}, A)$ otherwise;
- when $\mathsf{root}(\tau) \in \{\mathsf{aenc}, \mathsf{raenc}\}$, $E = \emptyset$ if $\tau|_{21}$ is an honest type, and $E = \rho_{\mathsf{out}}(\tau|_1, p.1, S, A \uplus \{\tau|_{21}\})$ otherwise.

Example 11 Consider the types $\tau_0^1 = \mathsf{aenc}(\mathsf{sign}(\tau_k, \tau_{sk}), \mathsf{pub}(\tau_{ekc}))$ and $\tau_0^2 = \mathsf{senc}(\tau_m, \tau_k)$, we have that:

- $\rho_{\mathsf{out}}(\tau_0^1) = \{(\tau_0^1, \epsilon)\#(\emptyset; \emptyset), (\mathsf{sign}(\tau_k, \tau_{sk}), 1)\#(\emptyset; \tau_{ekc}), (\tau_k, 11)\#(\emptyset; \tau_{ekc})\}$;

– $\rho_{\mathsf{out}}(\tau_0^2) = \{(\tau_0^2, \epsilon)\#(\emptyset;\emptyset),\ (\tau_m, 1)\#(\tau_k;\emptyset)\}$.

Definition 10 Given a well-sorted type τ, we define $\rho_{\mathsf{in}}(\tau)$ as follows:

– case where τ is an initial type: $\rho_{\mathsf{in}}(\tau) = \{\tau\}$;
– case where $\tau = \mathsf{f}(\tau_1, \ldots, \tau_n)$ for some $\mathsf{f} \in \Sigma_{\mathsf{c}}$:
  – $\rho_{\mathsf{in}}(\tau) = \{\tau\}$ if there exists $1 \leq i \leq n$ such that τ_i is an honest type;
  – $\rho_{\mathsf{in}}(\tau) = \{\tau\} \cup \bigcup_{i=1}^{n} \rho_{\mathsf{in}}(\tau_i)$ otherwise.

Example 12 Consider the types $\tau_i^1 = \mathsf{aenc}(\mathsf{sign}(\tau_k, \tau_{sk}), \mathsf{pub}(\tau_{ek}))$ and $\tau_i^2 = \mathsf{senc}(\tau_m, \tau_k)$, we have that:

– $\rho_{\mathsf{in}}(\tau_i^1) = \{\tau_i^1, \mathsf{pub}(\tau_{ek}), \mathsf{sign}(\tau_k, \tau_{sk})\}$;
– $\rho_{\mathsf{in}}(\tau_i^2) = \{\tau_i^2, \tau_m, \tau_k\}$.

We can now define the dependency graph associated to a protocol.

Definition 11 Let (Δ_0, δ_0) be a typing system, and P be a simple protocol. The *dependency graph* associated to P is a graph having $\mathcal{L}(P)$ as vertices, and which are connected as follows:

1. for any action with label ℓ in P that directly follows an action with label ℓ' in P, there is an edge $\ell \to \ell'$;
2. for any "$\ell_{\mathsf{in}} : \mathsf{in}(c, u)$" and "$\ell_{\mathsf{out}} : \mathsf{out}(d, v)$" in P such that $\mathsf{phase}(\ell_{\mathsf{out}}) \leq \mathsf{phase}(\ell_{\mathsf{in}})$, there is an edge $\ell_{\mathsf{in}} \to^p \ell_{\mathsf{out}}$ if there exists $\tau \in \rho_{\mathsf{in}}(u\delta_0)$ such that $(\tau, p)\#(S; A) \in \rho_{\mathsf{out}}(v\delta_0)$ (for some S and A);
3. for any "$\ell : \mathsf{out}(c, u)$" and "$\ell' : \mathsf{out}(d, v)$" in P, there is an edge $\ell \to^p \ell'$ if

$$(\tau, q)\#(S \uplus \{\tau_k\}; A) \in \rho_{\mathsf{out}}(u\delta_0) \text{ or } (\tau, q)\#(S; A \uplus \{\tau_k\}) \in \rho_{\mathsf{out}}(u\delta_0)$$

for some τ, q, S, A and τ_k such that $(\tau_k, p)\#(S'; A') \in \rho_{\mathsf{out}}(v\delta_0)$ (for some S' and A').

Intuitively, edges of type 1 simply record that some action occurs after another one. Edges of type 2 reflect how some term u expected as input may be built from terms terms output by the protocol. These dependencies are inferred from the type of the terms. Finally, edges of type 3 are there to record when a term, intuitively a ciphertext, can be opened using key material output by the protocol.

Example 13 The dependency graph for the protocol P_{DS}^1 defined in Example 6 w.r.t. the typing system $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$ given in Example 8 is depicted in Figure 3. The vertical arrows (in blue) correspond to sequential dependencies (item 1), the arrows in red are due to item 2, and the arrow in green are due to item 3.

Intuitively, the arrows from ℓ_3 to ℓ_1^K, ℓ_3^K, and ℓ_1' mean that the input ℓ_3 may depend on the outputs ℓ_1^K, ℓ_3^K, and ℓ_3, that is, the output term may be partially used to fill the input. The part of the output that could be used is indicated by the position p written on the arrow. Note that these dependencies are computed relying solely on types. The dependency graph for the protocol P_{DS}^2 will be exactly the same as the one obtained for P_{DS}^1.

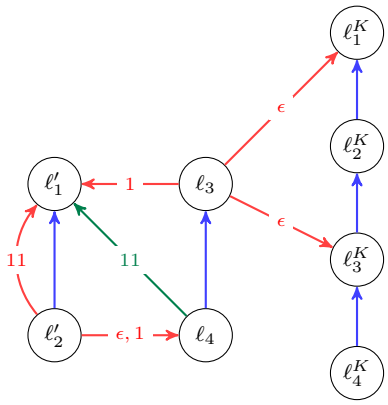

Fig. 3 Dependency graph for the simple protocol P^1_{DS}

3.4 Decidability Results

Our main result consists in showing that reachability properties and trace equivalence are decidable for protocols that are simple, type-compliant and with acyclic dependancy graphs.

3.4.1 Reachability.

We first establish decidability for reachability properties. This result is formally stated below and proved in Section 4

Theorem 1 *Let P be a simple protocol type-compliant w.r.t. some typing system (Δ_0, δ_0) and with an acyclic dependency graph. Let $\ell \in \mathcal{L}(P)$. The problem of deciding whether ℓ is reachable in P, i.e. whether*

$$P \xRightarrow{\mathsf{tr}} (\{i : \mathsf{io}^{\ell}(c, u).Q\} \cup \mathcal{P}; \phi; \sigma; i)$$

for some trace tr*, and* $\mathsf{io} \in \{\mathsf{in}, \mathsf{out}\}$ *is decidable.*

It is easy to encode a secrecy property as reachability of a particular action, as illustrated in the following example.

Example 14 To illustrate this result, we consider P''_{DS} which is as P'_{DS} but we add $\mathsf{in}^{\ell_5}(c_B, x_B)$ at the end of the process P_B. In case ℓ_5 is reachable in P''_{DS}, it means that the value of k as received by the agent B is deducible by the attacker (and thus k is not secret). We have that:

$$P'_{\mathsf{DS}} = 0 : P_K \mid 1 : \left(P'_A \mid P'_B\right)$$

where P'_A and P'_B are as follows:

$$P'_A = \mathsf{new}\ k'.\,\mathsf{out}^{\ell'_1}(c'_A, \mathsf{aenc}(\mathsf{sign}(k', sk_a), \mathsf{pub}(\mathsf{ek}_c))).\\ \mathsf{in}^{\ell'_2}(c'_A, \mathsf{senc}(x'_A, k')).0$$

$$P'_B = \mathsf{in}^{\ell_3}(c_B, \mathsf{aenc}(\mathsf{sign}(x_B, sk_a), \mathsf{pub}(ek_b))).\\ \mathsf{new}\ m.\,\mathsf{out}^{\ell_4}(c_B, \mathsf{senc}(m, x_B)).\mathsf{in}^{\ell_5}(c_B, x_B)$$

For the same reasons as P'_{DS}, we have that P''_{DS} is a simple protocol and it is type-compliant w.r.t. $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$. The dependency graph associated to P''_{DS} is similar to the one associated to P^1_{DS} and contains an additional node labelled ℓ_5 with:

- an additional arrow (of type 1) from ℓ_5 to ℓ_4;
- an additional arrow (of type 2) from ℓ_5 to ℓ'_1 with label 11 since a value of type $\delta_{\mathsf{DS}}(x_B) = \tau_k$ can be extracted at position 11 from the term outputted at ℓ'_1.

Note that the resulting dependency graph is still acyclic, and thus this protocol/scenario falls into our decidable class.

For sake of simplicity we have considered so far a simple protocol P''_{DS} that does not feature any replication (and thus the fact that it falls into our decidable class is not surprising). However, as detailed in the following example, our result applies also considering a richer scenario.

Example 15 We now consider the scenario $P^!_{\mathsf{DS}}$ which is similar to the process P''_{DS} given in Example 14 but includes replication (except for the role encoding key disclosure):

$$P^!_{\mathsf{DS}} = 0 : P_K \ \mid\ 1 : \big(!\mathsf{new}\, c'_A.\mathsf{out}(c_1, c'_A).P'_A \mid !\mathsf{new}\, c_B.\mathsf{out}(c_2, c_B).P'_B\big)$$

We have that $P^!_{\mathsf{DS}}$ is a simple protocol and is type-compliant w.r.t. $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$. Since replication does not play any role in the construction of the dependency graph, the dependency graph associated to $P^!_{\mathsf{DS}}$ is the same as the one for P''_{DS}. It is therefore acyclic meaning that this richer scenario falls also into our decidable class.

We consider more protocols, in an even richer scenario in Section 7.

3.4.2 Equivalence.

We can similarly decide trace equivalence by deciding trace inclusion.

Theorem 2 *Le P be a simple protocol type-compliant w.r.t. some typing system (Δ_P, δ_P) and with an acyclic dependency graph. Let Q be another simple protocol. The problem of deciding whether P is trace included in Q is decidable.*

Section 5 is dedicated to the proof of Theorem 2.

Example 16 We have that P^1_{DS} and P^2_{DS} are simple protocol type-compliant w.r.t. $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$ (see Example 9), and we have seen that the dependency graph associated to P^1_{DS} is acyclic (see Example 13). Thus this protocol/scenario falls into our decidable class.

Again, for sake of simplicity we have considered so far a simple protocol P^1_{DS} that does not feature any replication but our result applies for a richer scenario with replicated processes.

4 Decidability result for reachability

The main ingredient of the proof of Theorem 1 is to show that an execution trace corresponds to a path in the dependency graph. This is not true for all traces but for well-typed traces where the recipes used by the adversary follow certain conditions. The goal of this section is to introduce these conditions and show that they can be fulfilled. Part of the results established in this section will also be used for the proof of Theorem 2 on equivalence.

4.1 Well-typed traces involving only simple recipes

To establish our decidability result, we first show that we can consider well-typed executions involving only simple recipes. Simple recipes are recipes that are of the form destructors over constructors.

Definition 12 We say that a recipe is simple if there exist destructor-only recipes $R_1, \ldots, R_k$, i.e. recipes in $\mathcal{T}(\Sigma_{\mathsf{d}}, \mathcal{W} \cup \Sigma_0) \setminus \Sigma_0$, and a context C made of constructors, i.e. function symbols in $\Sigma_{\mathsf{c}} \cup \Sigma_0$, such that $R = C[R_1, \ldots, R_k]$.

When we consider a simple recipe R of the form $C[R_1, \ldots, R_k]$, we implicitly refer to the decomposition expressed above meaning that $R_1, \ldots, R_k$ are the maximal destructor-only recipes occurring in R. As formally stated in the next lemma, destructor-only recipes may only deduce a subterm of the initial frame.

Lemma 2 *Let ϕ be a frame, and R be a destructor-only recipe such that $R\phi{\downarrow}$ is a message. We have that $R\phi{\downarrow} \in St(\phi)$.*

Proof We prove this result by structural induction on R.

Base case: $R \in \mathcal{W}$. In such a case, the result trivially holds.

Induction case. In such a case, we have that

- either $R = \mathsf{g}(R_1, R_2)$ with $\mathsf{g} \in \{\mathsf{sdec}, \mathsf{rsdec}, \mathsf{adec}, \mathsf{radec}\}$;
- or $R = \mathsf{g}(R_1)$ with $\mathsf{g} \in \{\mathsf{getmsg}\} \cup \{\mathsf{proj}^j_n \mid n \geq 2 \text{ and } 1 \leq j \leq n\}$.

In both cases, we know that R_1 is a destructor-only recipe, and thus, by induction hypothesis, we have that $R_1\phi{\downarrow} \in St(\phi)$. Actually, we have that $R\phi{\downarrow} \in St(R_1\phi{\downarrow})$. Therefore, we have that $R\phi{\downarrow} \in St(\phi)$. □

As in [13], we introduce the notion of *forced normal form*, denoted $u{\downarrow}$. It intuitively corresponds to applying the rewriting rules even when some equalities are not satisfied, e.g. decrypting even with the wrong key. The idea is to pre-compute what can be deduced at best using the real rewriting system. We give below the rules $\mathcal{R}_f$ associated to the rewriting system given in Section 2.

$$\begin{array}{rr} \mathsf{sdec}(\mathsf{senc}(x,y_1),y_2) \twoheadrightarrow x & \mathsf{rsdec}(\mathsf{rsenc}(x,y_1,y_2),y_3) \twoheadrightarrow x \\ \mathsf{adec}(\mathsf{aenc}(x,y_1),y_2) \twoheadrightarrow x & \mathsf{radec}(\mathsf{raenc}(x,y_1,y_2),y_3) \twoheadrightarrow x \\ \mathsf{getmsg}(\mathsf{sign}(x,y_1)) \twoheadrightarrow x & \mathsf{check}(x,y) \twoheadrightarrow \mathsf{ok} \end{array}$$

$$\mathsf{proj}_i^n(\langle x_1,\ldots,x_n\rangle_n) \twoheadrightarrow x_i \quad \text{with } n \geq 2 \text{ and } 1 \leq i \leq n$$

A term u can be rewritten in v using the $\mathcal{R}_f$ if there exists a position p in u, and a rewriting rule $\mathsf{g}(t_1,\ldots,t_n) \twoheadrightarrow t$ such that $u|_p = \mathsf{g}(t_1,\ldots,t_n)\theta$ for some substitution θ, and $v = u[t\theta]_p$. As usual, we denote $\twoheadrightarrow^*$ the reflexive and transitive closure of $\twoheadrightarrow$. For example, $\mathsf{sdec}(\mathsf{senc}(a,k),k') \twoheadrightarrow a$ but $\mathsf{sdec}(\mathsf{aenc}(a,\mathsf{pub}(k)),k') \not\twoheadrightarrow a$.

The forced rewriting system allows more rewriting steps than the original system. Nevertheless, the following lemma (stated and proved in [13]) ensures that if R can be used to obtain a message then $R{\downarrow}$ computes the same message.

Lemma 3 *[13] Let ϕ be a frame, R be a recipe such that $R\phi{\downarrow}$ is a message, and R' be such that $R \twoheadrightarrow R'$. We have that R' is a recipe and $R'\phi{\downarrow} = R\phi{\downarrow}$.*

We show that we can always chose simple recipes, simply by considering recipes in forced normal form. This result is similar to the one established in [13] but for a slightly different notion of simple terms (because [13] considers a more general equational theory).

Lemma 4 *Let ϕ be a frame and u be a message deducible from ϕ, i.e. such that $R\phi{\downarrow} = u$ for some R. We have that $R{\downarrow}\phi{\downarrow} = u$ and $R{\downarrow}$ is a simple recipe.*

Proof Let R be a recipe such that $R\phi{\downarrow} = u$. Thanks to Lemma 3, we have that $R{\downarrow}\phi{\downarrow} = u$. We now prove that $R' = R{\downarrow}$ is simple by structural induction on R'.
Base case: $R' \in \mathcal{W} \cup \Sigma_0$. In both cases, it is easy to see that R' is indeed simple.

Induction case: $R' = \mathsf{f}(R_1,\ldots,R_k)$ for some $\mathsf{f} \in \Sigma$ and $R_1,\ldots,R_k$ are in forced normal form. We distinguish two cases:

- *Case* $\mathsf{f} \in \Sigma_\mathsf{c}$. Applying our induction hypothesis on R_i $(1 \leq i \leq k)$, we easily conclude.
- *Case* $\mathsf{f} \in \Sigma_\mathsf{d} \cup \{\mathsf{check}\}$. Note that the case where $\mathsf{f} = \mathsf{check}$ is impossible since R' is in forced normal form. Now assume that $\mathsf{f} = \mathsf{adec}$, and thus $R' = \mathsf{adec}(R_1,R_2)$. As $R'\phi{\downarrow}$ is a message, $R_1\phi{\downarrow}$ and $R_2\phi{\downarrow}$ are messages. Applying our induction hypothesis, we know that both R_1 and R_2 are simple. As $R_2\phi{\downarrow}$ is an atomic message, we know that R_2 is either destructor-only or a constant. Assume that $R_1 = \mathsf{g}(R'_1,\ldots,R'_n)$ for some $\mathsf{g} \in \Sigma_\mathsf{c}$. As

$R'\phi\downarrow$ is a message, we have that $\mathsf{g} = \mathsf{aenc}$ contradicting the fact that R' is in forced normal form. Thus R_1 is destructor-only, and therefore R' is simple. The other cases can be done in a similar way. □

Interestingly, simple recipes do not use spurious constants: all constants that appear in a simple recipe R remain in the deduced message unless they already appear in the initial frame.

Lemma 5 *Let ϕ be a frame and R be a simple recipe such that $R\phi\downarrow$ is a message. We have that $\mathsf{Cst}(R) \subseteq \mathsf{Cst}(\phi) \cup \mathsf{Cst}(R\phi\downarrow)$.*

Proof We prove this result by structural induction on R.

Base case: $R \in \mathcal{W} \cup \Sigma_0$. In such a case, the result trivially holds.

Induction case: $R = \mathsf{f}(R_1, \ldots, R_k)$ *for some* $\mathsf{f} \in \Sigma_\mathsf{c} \cup \Sigma_\mathsf{d}$.

- *Case* $\mathsf{f} \in \Sigma_\mathsf{c}$. In such a case, we have that $R\phi\downarrow = \mathsf{f}(R_1\phi\downarrow, \ldots, R_k\phi\downarrow)$, and we easily conclude relying on our induction hypothesis.
- *Case* $\mathsf{f} \in \Sigma_\mathsf{d}$. In such a case, we have that
  - either $R = \mathsf{g}(R_1, R_2)$ with $\mathsf{g} \in \{\mathsf{sdec}, \mathsf{rsdec}, \mathsf{adec}, \mathsf{radec}\}$;
  - or $R = \mathsf{g}(R_1)$ with $\mathsf{g} \in \{\mathsf{getmsg}\} \cup \{\mathsf{proj}_n^j \mid n \geq 2 \text{ and } 1 \leq j \leq n\}$.

  In both cases, we know that R_1 is a destructor-only recipe, and thus, by Lemma 2, we have that $R_1\phi\downarrow \in St(\phi)$. Regarding R_2, when it exists, we have that $R_2\phi\downarrow \in St(R_1\phi\downarrow)$. Therefore, applying our induction hypothesis on both R_1 and R_2 that are simple, we have that:

$$\begin{aligned}\mathsf{Cst}(R) &= \mathsf{Cst}(R_1) \cup \mathsf{Cst}(R_2)\\ &\subseteq \mathsf{Cst}(\phi) \cup \mathsf{Cst}(R_1\phi\downarrow) \cup \mathsf{Cst}(R_2\phi\downarrow)\\ &= \mathsf{Cst}(\phi)\end{aligned}$$

This concludes the proof. □

A key step for decidability is that we can consider only well-typed traces thanks to [13]. We state here this result and show in addition that it is enough to consider simple, honest-free recipes. When restricting ourselves to well-typed traces, we still preserve the general form of the trace. Formally, $\overline{\mathsf{tr}}$ is obtained from tr by replacing any action $\mathsf{in}(c, R)$ by $\mathsf{in}(c, \_)$, and keeping the other visible actions, i.e. $\mathsf{out}(c, \mathsf{w})$, $\mathsf{out}(c, c')$ and the phase instruction, unchanged.

Theorem 3 *Let $\mathcal{K}_0$ be an initial configuration type-compliant w.r.t. (Δ_0, δ_0) such that $\mathcal{K}_0 \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$. We have that there exists a well-typed execution $\mathcal{K}_0 \overset{\mathsf{tr}'}{\Longrightarrow} (\mathcal{P}; \phi'; \sigma'; i)$ involving only simple recipes such that $\overline{\mathsf{tr}'} = \overline{\mathsf{tr}}$, and tr' as well as ϕ' are honest-free.*

Proof Most of the theorem is a direct consequence of the typing result that has been established in [13]. Then, the fact that we can consider simple recipes is an easy consequence of Lemma 4. It remains to establish that we can consider tr' and ϕ' to be honest-free.

According to Proposition 4.11 stated and proved in [13], we know that the well-typed substitution σ' is such that $\sigma' = \sigma_S\rho$ where:

- σ_S is the most general unifier (denoted mgu) of $\Gamma = \{(u, v) \mid u, v \in ESt(\mathcal{K}_0)$ such that $u\sigma = v\sigma\}$; and
- ρ is a bijective renaming from variables in $dom(\sigma) \setminus dom(\sigma_S)$ to some fresh constants preserving type.

In Lemma 4.10 of [13], it has been shown that $ESt(\mathcal{K}_0\sigma_S) \subseteq ESt(\mathcal{K}_0)\sigma_S$, and since ρ is a renaming, we have that:

$$ESt(\mathcal{K}_0\sigma') \subseteq ESt(\mathcal{K}_0)\sigma' \tag{1}$$

We now show that $\mathsf{tr}'\phi'\downarrow$ is honest-free. Assume by contradiction that there exists a constant c_h of honest type occurring in $\mathsf{tr}'\phi'\downarrow$. In other words, c_h occurs in an instantiation by σ' of an input or output action of the initial processes, possibly after renaming names and variables (when unfolding replication). Thus the constant c_h must occur in $\mathcal{K}_0\sigma'$.

By definition of being an honest type, c_h can only occur in key position in $\mathcal{K}_0\sigma'$. This means that there exists either $\mathsf{f}(u, c_h) \in ESt(\mathcal{K}_0\sigma')$ with $\mathsf{f} \in \{\mathsf{senc}, \mathsf{rsenc}, \mathsf{sign}\}$, or $\mathsf{pub}(c_h) \in ESt(\mathcal{K}_0\sigma')$. Thanks to (1), we deduce that there exists either $\mathsf{f}(u', v') \in ESt(\mathcal{K}_0)$ such that $\mathsf{f}(u', v')\sigma' = \mathsf{f}(u, c_h)$, or $\mathsf{pub}(v') \in ESt(\mathcal{K}_0)$ such that $\mathsf{pub}(v')\sigma' = \mathsf{pub}(c_h)$. This implies that v' is either a variable of honest type, or the constant c_h, and both are forbidden according to the definition of honest type. This allows us to conclude that $\mathsf{tr}'\phi'\downarrow$ is honest-free.

We have that $\mathsf{Terms}(\phi') \subseteq \mathsf{Terms}(\mathsf{tr}'\phi'\downarrow)$, and thus we easily deduce that ϕ' is honest-free. Now, regarding recipes occurring in tr', we know that they are simple, and thanks to Lemma 5, we have that $\mathsf{Cst}(R) \subseteq \mathsf{Cst}(\phi') \cup \mathsf{Cst}(R\phi'\downarrow)$ for any recipe R occurring in tr'. We have that constants occurring in R already occur in ϕ' or $\mathsf{tr}'\phi'\downarrow$, and since we have seen that no constant of honest type occurs in ϕ' and $\mathsf{tr}'\phi\downarrow$, we are done. □

4.2 Exploiting the dependency graph

We are almost ready to show that any (well-typed, simple, honest-free) trace can be mapped to a path of the dependency graph. For this, we define ρ_{out} and ρ_{in} on terms. Intuitively, ρ_{in} computes how a term can be built by an attacker.

Definition 13 Given a well-sorted term t and a typing system (Δ_0, δ_0), we define $\rho_{\mathsf{in}}(t)$ as follows:

- case where t is atomic: $\rho_{\mathsf{in}}(t) = \{t\}$;
- case where $t = \mathsf{f}(t_1, \ldots, t_n)$ for some $\mathsf{f} \in \Sigma_{\mathsf{c}}$:
  - $\rho_{\mathsf{in}}(t) = \{t\}$ if there exists $1 \leq i \leq n$ such that t_i has an honest type;
  - $\rho_{\mathsf{in}}(t) = \{t\} \cup \bigcup_{i=1}^{n} \rho_{\mathsf{in}}(t_i)$ otherwise.

Lemma 6 *Let $\mathcal{K}_0$ be an initial configuration, (Δ_0, δ_0) be a typing system, and $\mathcal{K}_0 \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$ be a well-typed honest-free execution involving only simple recipes. In particular, no constant of honest type occurs in tr and ϕ.*

Let $R = C[R_1, \ldots, R_n]$ be a simple recipe such that $R\phi{\downarrow}$ is a message and R is honest-free. We have that $R_i\phi{\downarrow} \in \rho_{\mathsf{in}}(R\phi{\downarrow})$ for all $i \in \{1, \ldots, n\}$.

Proof We establish the result by structural induction on C.

Base case: C is the empty context or a constant. In such a case, the result trivially holds.

Induction case. We have that $C = \mathsf{f}(C_1, \ldots, C_k)$ for some $\mathsf{f} \in \Sigma_{\mathsf{c}}$. In such a case, we have that $R\phi{\downarrow} = \mathsf{f}(t_1, \ldots, t_k)$, and we have recipes, namely $R'_1, \ldots, R'_k$, allowing one to deduce $t_1, \ldots, t_k$. First, assume that there exists $i \in \{1, \ldots, k\}$ such that t_i has an honest type. In such a case, t_i is atomic, and actually t_i is neither a constant (otherwise, an honest constant would occur in R or in ϕ), nor a name (by Lemma 1). Thus, we know that no t_i has an honest type. Thus, by definition of $\rho_{\mathsf{in}}(R\phi{\downarrow})$, we have that

$$\rho_{\mathsf{in}}(R\phi{\downarrow}) = \{R\phi{\downarrow}\} \cup \rho_{\mathsf{in}}(t_1) \cup \ldots \cup \rho_{\mathsf{in}}(t_k)$$

Applying our induction hypothesis on $R'_i = C_i[R_1, \ldots, R_n]$ with $1 \leq i \leq k$, we have that for any $j \in \{1, \ldots, n\}$, $R_j\phi{\downarrow} \in \rho_{\mathsf{in}}(C_i[R_1, \ldots, R_n]\phi{\downarrow}) = \rho_{\mathsf{in}}(t_i)$.

Therefore, we have that $R_i\phi{\downarrow} \in \rho_{\mathsf{in}}(R\phi{\downarrow})$ for any $i \in \{1, \ldots, n\}$. □

We now define ρ_{out} on terms. It intuitively computes which terms can be deduced from a term t, tracking respectively the symmetric and asymmetric keys needed for that.

Definition 14 Given a well-sorted term t and a typing system (Δ_0, δ_0), we define $\rho_{\mathsf{out}}(t)$ to be $\rho_{\mathsf{out}}(t, \epsilon, \emptyset, \emptyset)$ where $\rho_{\mathsf{out}}(t, p, S, A)$ is recursively defined as the set $\{(t, p)\#(S; A)\} \cup E$ where E is defined as follows:

- $E = \emptyset$ when t is atomic or $\mathsf{root}(t) \in \{\mathsf{pub}, \mathsf{vk}, \mathsf{hash}\}$;
- $E = \bigcup_{i=1}^{n} \rho_{\mathsf{out}}(t_i, p.i, S, A)$ when $t = \langle t_1, \ldots, t_n \rangle$;
- $E = \rho_{\mathsf{out}}(t_1, p.1, S, A)$ when $t = \mathsf{sign}(t_1, t_2)$;
- when $\mathsf{root}(t) \in \{\mathsf{senc}, \mathsf{rsenc}\}$, $E = \emptyset$ if $t|_2$ has an honest type, and $E = \rho_{\mathsf{out}}(t_1, p.1, S \uplus \{t|_2\}, A)$ otherwise;
- when $\mathsf{root}(t) \in \{\mathsf{aenc}, \mathsf{raenc}\}$, $E = \emptyset$ if $t|_{21}$ has an honest type, and $E = \rho_{\mathsf{out}}(t_1, p.1, S, A \uplus \{t|_{21}\})$ otherwise.

A destructor-only recipe R intuitively tries to decrypt and project one term u. Such a recipe deconstructs the term u to extract its subterm at position $\mathsf{target}(R)$ in u, where $\mathsf{target}(R)$ is defined as follows:

$$\begin{cases} \epsilon & \text{if } R \text{ is a variable } \mathsf{w} \\ \mathsf{target}(R|_1).1 & \text{if } \mathsf{root}(R) \in \{\mathsf{proj}_1^n \mid n \geq 2\} \cup \{\mathsf{getmsg}, \mathsf{sdec}, \mathsf{rsdec}, \mathsf{adec}, \mathsf{radec}\} \\ \mathsf{target}(R|_1).i & \text{if } \mathsf{root}(R) = \mathsf{proj}_i^n \text{ for some } n \geq 2 \text{ and some } i \in \{2, \ldots, n\} \end{cases}$$

For a destructor-only recipe R with a variable w at its leftmost leaf, $\mathsf{target}(R)$ is the position of the subterm computed by R inside $\mathsf{w}\phi$. Note that, thanks to Lemma 2, we know that the result of normalising R must be a subterm of $\mathsf{w}\phi$, since R is destructor-only and applied to w. We show that, if $R\phi{\downarrow}$ is a message, then it was indeed computed by $\rho_{\mathsf{out}}(\mathsf{w}\phi)$.

Lemma 7 *Let $\mathcal{K}_0$ be an initial configuration, (Δ_0, δ_0) be a typing system, and $\mathcal{K}_0 \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ be a well-typed honest-free execution involving only simple recipes. Let R be a destructor-only recipe with the variable w at its leftmost position such that $R\phi\!\downarrow$ is a message. We have that $(R\phi\!\downarrow, \mathsf{target}(R))\#(S, A) \in \rho_{\mathsf{out}}(\mathsf{w}\phi)$. Moreover $R|_2\phi\!\downarrow \in S$ (resp $R|_2\phi\!\downarrow \in A$) when $\mathsf{root}(R) = \mathsf{sdec}$ or rsdec (resp. adec or radec).*

Proof We establish this result by structural induction on R.
Base case: $R = \mathsf{w}$. Indeed, we have that $(\mathsf{w}\phi, \epsilon)\#(\emptyset; \emptyset) \in \rho_{\mathsf{out}}(\mathsf{w}\phi)$.

Induction case. In this case, we have that:

- either $R = \mathsf{g}(R_1, R_2)$ with $\mathsf{g} \in \{\mathsf{sdec}, \mathsf{rsdec}, \mathsf{adec}, \mathsf{radec}\}$;
- or $R = \mathsf{g}(R_1)$ with $\mathsf{g} \in \{\mathsf{getmsg}\} \cup \{\mathsf{proj}_n^j \mid n \geq 2 \text{ and } 1 \leq j \leq n\}$.

From now on, consider the case where $\mathsf{g} = \mathsf{adec}$. By induction hypothesis, we have that $(R_1\phi\!\downarrow, \mathsf{target}(R_1))\#(S_1; A_1) \in \rho_{\mathsf{out}}(\mathsf{w}\phi)$ for some A_1 and S_1. We also know that $t = R_1\phi\!\downarrow = \mathsf{aenc}(t_1, \mathsf{pub}(a))$ for some t_1 and some atom a. First, we note that $t|_{21} = a$ does not have an honest type. Indeed, this case is impossible since a is necessarily a name occurring in ϕ and a name with an honest type is not deducible (Lemma 1). Therefore, according to the definition of ρ_{out}, we have that $\rho_{\mathsf{out}}(t_1, \mathsf{target}(R_1).1, S_1, A_1 \uplus \{a\}) \subseteq \rho_{\mathsf{out}}(\mathsf{w}\phi)$. Thus, we have that $(t_1, \mathsf{target}(R_1).1)\#(S_1; A_1 \uplus \{a\}) \in \rho_{\mathsf{out}}(\mathsf{w}\phi)$, i.e. $(R\phi\!\downarrow, \mathsf{target}(R))\#(S; A) \in \rho_{\mathsf{out}}(\mathsf{w}\phi)$ for some S and some A. Moreover, we have $R|_2\phi\!\downarrow = R_2\phi\!\downarrow = a$ in A and $\mathsf{root}(R) = \mathsf{adec}$. This concludes this case, and the other cases can be handled in a similar way. □

Of course, we can link the definitions of ρ_{in} and ρ_{out} on types and the ones on terms.

Lemma 8 *Let (Δ_0, δ_0) be a typing system and u be a well-sorted term. We have that:*

- *$v \in \rho_{\mathsf{in}}(u)$ implies $\delta_0(v) \in \rho_{\mathsf{in}}(\delta_0(u))$; and*
- *$(v, p)\#(S; A) \in \rho_{\mathsf{out}}(u)$ implies $(\delta_0(v), p)\#(\delta_0(S); \delta_0(A)) \in \rho_{\mathsf{out}}(\delta_0(u))$.*

Proof Regarding the first result about ρ_{in}, it can be easily proven by structural induction on u. To establish the second result about ρ_{out}, we prove the following result by structural induction on u:

$$(v, p)\#(S; A) \in \rho_{\mathsf{out}}(u, p_0, S_0, A_0) \text{ implies}$$
$$(\delta_0(v), p)\#(\delta_0(S); \delta_0(A)) \in \rho_{\mathsf{out}}(\delta_0(u), p_0, \delta_0(S_0), \delta_0(A_0)).$$

The result is a direct consequence of this property. □

Note that the converse implications (from types to terms) do not hold: the structure of the type may be finer that the structure of the corresponding term, e.g. $\delta(x) = \langle \tau_1, \tau_2 \rangle$.

Any dependency arising in an execution of a protocol can be mapped to its dependency graph.

Proposition 1 *Let P be a simple protocol, (Δ_0, δ_0) be a typing system, and $P \xrightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; j_0)$ be a well-typed honest-free execution involving only simple recipes.*

For any pair of actions $\mathsf{in}^{\ell_{\mathsf{in}}}(d, C[R_1, \ldots, R_k])/\mathsf{out}^{\ell_{\mathsf{out}}}(c, \mathsf{w})$ occurring in tr with $\mathsf{w} \in vars(R_{i_0})$ $(1 \leq i_0 \leq k)$, we have that:

$$\ell_{\mathsf{in}} \to \ell^0_{\mathsf{out}} \to^* \ell_{\mathsf{out}} \text{ is a path in the dependency graph associated to } P$$

where ℓ^0_{out} is the label associated to w_0, the handle occurring at the leftmost position in R_{i_0}. Moreover, the length of the path from ℓ^0_{out} to ℓ_{out} is equal to the number of occurrences of 2 in p (the position at which w occurs in R_{i_0}).

Proof Let $R = C[R_1, \ldots, R_k]$. Note that since $\mathsf{w} \in vars(R)$, we have that $\mathsf{phase}(\ell_{\mathsf{out}}) \leq \mathsf{phase}(\ell_{\mathsf{in}})$. The execution being honest-free, we have that no constant of honest type occurs in tr. Let $i_0 \in \{1, \ldots, k\}$ be such that w occurs in R_{i_0}, and let p be a position at which w occurs in R_{i_0}. We show the result by induction on the number of occurrences of 2 in p.

1. *Base case: p is a possibly empty sequence of 1.* In this case, we have that w occurs at the leftmost position of R_{i_0}. Applying Lemma 7, we have that $(R_{i_0}\phi{\downarrow}, \mathsf{target}(R_{i_0}))\#(S; A) \in \rho_{\mathsf{out}}(\mathsf{w}\phi)$. Then, thanks to Lemma 6, we have that $R_{i_0}\phi{\downarrow} \in \rho_{\mathsf{in}}(R\phi{\downarrow})$. We rely on Lemma 8 to transfer these relations on types and we conclude the existence of an edge $\ell_{\mathsf{in}} \to^{\mathsf{target}(R_{i_0})} \ell_{\mathsf{out}}$ of type 2 in the dependency graph.
2. *Induction case: $p = p_0.2.1 \ldots 1$ with a possibly empty sequence of 1 at the end.* Applying Lemma 7 on $R_{i_0}|_{p_0}$ and denoting w_{p_0} the variable occurring at its leftmost position, we have that:

   $$(R_{i_0}|_{p_0}\phi{\downarrow}, \mathsf{target}(R_{i_0}|_{p_0}))\#(S; A) \in \rho_{\mathsf{out}}(\mathsf{w}_{p_0})$$

   for some S and A such that $R_{i_0}|_{p_0.2}\phi{\downarrow} \in S$ (resp. A) when $\mathsf{root}(R_{i_0}|_{p_0}) = \mathsf{sdec}$ or rsdec (resp. adec or radec).
   Applying Lemma 7 on $R_{i_0}|_{p_0.2}$ (note that the variable occurring at its leftmost position is w), we have that

   $$(R_{i_0}|_{p_0.2}\phi{\downarrow}, \mathsf{target}(R_{i_0}|_{p_0.2}))\#(S'; A') \in \rho_{\mathsf{out}}(\mathsf{w})$$

   for some S' and A'. We rely on Lemma 8 to transfer these relations on types and we conclude to the existence of an edge $\ell'_{\mathsf{out}} \to^{\mathsf{target}(R_{i_0}|_{p_0.2})} \ell_{\mathsf{out}}$ of type 3 in the dependency graph – where ℓ'_{out} is the label associated to w_{p_0}.
   By induction hypothesis, we have that: $\ell_{\mathsf{in}} \to^+ \ell'_{\mathsf{out}}$ since w_{p_0} occurs at position $p_0.1 \ldots 1$ in R_{i_0} and this position contains less occurrences of 2 than $p_0.2.1 \ldots 1$. This gives us the expected result. □

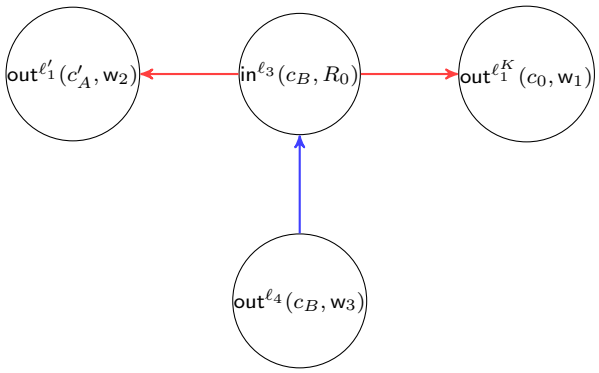

Fig. 4 Execution graph associated to tr_0 for the simple protocol P'_{DS}

4.3 Bounding the length of a minimal witness

Given a trace (tr, ϕ) of a simple protocol P, we can represent it as a dag D (directed acyclic graph) whose vertices are input/output actions of tr, and edges represent sequential dependencies and data dependencies. Note that such a dag can be computed simply from tr. Indeed, for simple protocols, sequential dependencies may be inferred from the channel names occurring in tr, and data dependencies are inferred from input recipes occurring in tr. Our ultimate goal is to bound the length of a trace tr witnessing the existence of an attack.

We first define the notion of execution graph.

Definition 15 Let P be a simple protocol. The *execution graph* associated to an execution ex starting from P is a directed acyclic graph whose vertices are the actions of ex of the form $\mathsf{in}^{\ell}(c, R)$ and $\mathsf{out}^{\ell}(c, w)$, and whose edges denoted $\mapsto$, are defined as follows:

- there is an edge from an action a_2 with label ℓ_2 to an other action a_1 with label ℓ_1 if both actions are on the same channel, and ℓ_2 directly follows ℓ_1 in P;
- there is an edge from $\mathsf{in}^{\ell}(c, R)$ to an action $\mathsf{out}^{\ell'}(c', \mathsf{w})$ if $\mathsf{w} \in \mathit{vars}(R)$.

We note that for a simple protocol P, the execution graph associated to a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ is unique. The actions of the form $\mathsf{out}(c, c'')$ with c'' a channel name are not part of the execution graph.

Example 17 The execution graph associated to the trace $(\mathsf{tr}_0, \phi_0) \in \mathsf{trace}(\mathcal{K}_0)$ given in Example 4 is depicted in Figure 4. Remember that

$$\mathsf{tr}_0 = \mathsf{out}(c_0, \mathsf{w}_1).\mathsf{phase}\,1.\mathsf{out}(c'_A, \mathsf{w}_2).\mathsf{in}(c_B, R_0).\mathsf{out}(c_B, \mathsf{w}_3)$$

where $R_0 = \mathsf{aenc}(\mathsf{adec}(\mathsf{w}_2, \mathsf{ek}_c), \mathsf{w}_1)$.

Given a directed acyclic graph D, the *width* of D, denoted $\mathsf{width}(D)$, is the maximum outgoing degree of any vertex of D. The *depth* of D, denoted $\mathsf{depth}(D)$, is the length of the longest path in D, and we denote $\mathsf{nbroot}(D)$ its *number of roots*, i.e. vertices with no ingoing edge.

Lemma 9 *Let (Δ_0, δ_0) be a typing system and P be a simple protocol whose associated dependency graph G is acyclic. Let $P \xrightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i_0)$ be a well-typed honest-free execution involving only simple recipes, and D its associated execution graph. We have that:*

$$\mathsf{depth}(D) \leq \mathsf{depth}(G).$$

Proof We denote ρ the function which associates to an action a occurring in tr, its label $\ell \in \mathcal{L}(P)$. Given an arrow $a \mapsto b$ between two actions of the execution graph D, we show that $\rho(a) \rightarrow^+ \rho(b)$ in the dependency graph G. By definition, this arrow either corresponds to a sequential dependency or corresponds to a data dependency. In case of a sequential dependency, the same arrow exists in the dependency graph. In case of a data dependency, Proposition 1 ensures that $\rho(a) \rightarrow^+ \rho(b)$. This allows us to conclude. □

Given P a simple protocol which is type-compliant w.r.t. some typing system (Δ_0, δ_0), we consider the sizes induced by the types that appear in input and output actions of the protocol. Let $\|u\|$ denote the size of u. Then we define:

- $\|\mathsf{in}_P\| = max\{\|u\delta_0\| \mid \mathsf{in}(c, u) \text{ occurs in } P \text{ for some } c\}$;
- $\|\mathsf{out}_P\| = max\{\|u\delta_0\| \mid \mathsf{out}(c, u) \text{ occurs in } P \text{ for some } c\}$.

We show that the width of the execution graph of a trace of P can be bounded depending on the size of the types appearing in P.

Lemma 10 *Let (Δ_0, δ_0) be a typing system, and P be a simple protocol whose associated dependency graph G is acyclic. Let $P \xrightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i_0)$ be a well-typed honest-free execution involving only simple recipes, and D its associated execution graph. We have that:*

$$\mathsf{width}(D) \leq 1 + \|\mathsf{out}_P\|^{\mathsf{depth}(G)} \times \|\mathsf{in}_P\|.$$

Proof Any node has at most one sequential predecessor. Now, in case of an input, we have to take into account data dependencies. We know that the involved recipe R is of the form $C[R_1, \ldots, R_k]$ with R_i $(1 \leq i \leq k)$ destructor-only and $k \leq \|\mathsf{in}_P\|$ since C is at most of size $\|\mathsf{in}_P\|$. Moreover, thanks to Proposition 1, we have that, for any $i \in \{1, \ldots, k\}$, the maximal number of occurrences of 2 in any sequence in $pos(R_i)$ (the positions of R_i) is bounded by $\mathsf{depth}(G)$.

To conclude, it remains to show that we can bound the number of variables occurring in such recipes. Let S_ℓ be the set of all destructor-only recipes R such that any sequence in $pos(R)$ (the positions of R) contains at most ℓ occurrences of 2. We aim at bounding $\mathsf{nb}_\ell = Max\{\#\{w \in vars(R)\} \mid R \in S_\ell\}$. Clearly, for

$\ell = 0$, we have that $\mathsf{nb}_\ell = 1$. Moreover, $\mathsf{nb}_{\ell+1}$ can be bound by $\|\mathsf{out}_P\| \times \mathsf{nb}_\ell$ since the number of consecutive 1 in a path is bounded by $\|\mathsf{out}_P\|$, the maximal size of a term to which the recipe can be applied. Therefore, each R_i involves no more than $\|\mathsf{out}_P\|^{\mathsf{depth}(G)}$ variables, and this allows us to conclude. □

To establish our result, we show that pruning an execution graph w.r.t. a set of nodes still yields a valid trace for P. This notion of pruning preserves all sequential and data dependencies and is formally defined as follows.

Definition 16 Given an execution graph $D = (V, E)$ and a set $R \subseteq V$, we define the pruning $D_R = (V_R, E_R)$ of D w.r.t. R as follows:

- $V_R = \{v \in V \mid r \mapsto^* v \text{ for some } r \in R\}$;
- $E_R = \{(u, v) \in E \mid u, v \in V_R\}$

where $\mapsto^*$ denotes the transitive closure of the relation induced by E.

Let P be a simple protocol and D the execution graph associated to a given trace tr. We note that the pruning of D w.r.t. some nodes still corresponds to an execution of P. The underlying trace tr' is actually a subtrace of tr. More formally, we have the following result.

Lemma 11 *Let P be a simple protocol, $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and D the execution graph of tr w.r.t. P. Let $R = \{v_1, \ldots, v_p\}$ be a set of nodes of D and D_R the pruning of D w.r.t. R. Then, there exists $(\mathsf{tr}_R, \phi_R) \in \mathsf{trace}(P)$ such that:*

- *D_R is the execution graph of tr_R w.r.t. P;*
- *tr_R is a subtrace of tr, and ϕ_R is a subframe of ϕ.*

Proof The execution graph captures all the dependencies, and thus the closure of D_R ensures that any action occurring in tr_R has the needed predecessors. □

Finally, we can bound the number of nodes of an execution graph (for well chosen executions), hence decide reachability.

Theorem 1 *Let P be a simple protocol type-compliant w.r.t. some typing system (Δ_0, δ_0) and with an acyclic dependency graph. Let $\ell \in \mathcal{L}(P)$. The problem of deciding whether ℓ is reachable in P, i.e. whether*

$$P \xRightarrow{\mathsf{tr}} (\{i : \mathsf{io}^\ell(c, u).Q\} \cup \mathcal{P}; \phi; \sigma; i)$$

for some trace tr, and $\mathsf{io} \in \{\mathsf{in}, \mathsf{out}\}$ is decidable.

Proof To establish this result, we show that there is a trace witnessing this fact whose execution graph $D = (V, E)$ is such that:

$$\#V \leq (2 + \|\mathsf{out}_P\|^{\mathsf{depth}(G)} \times \|\mathsf{in}_P\|)^{\mathsf{depth}(G)+1}.$$

First, thanks to Theorem 3, we can consider that there is a trace (tr, ϕ) witnessing that ℓ is reachable that is well-typed, simple, and honest-free. Then,

we consider the execution graph D associated to this trace (tr, ϕ). We know that one node (at least) is labelled with $\mathsf{io}^{\ell}(c, \_)$. Let D' be the execution graph corresponding to the pruning of D w.r.t. such a node. Thanks to Lemma 11, we have that $D' = (V', E')$ is an execution graph corresponding to an execution leading to the action $\mathsf{io}^{\ell}(c, \_)$. Moreover, we have that:

$$\begin{aligned}\#V' &\leq 1 + \mathsf{width}(D') + \mathsf{width}(D')^2 + \ldots + \mathsf{width}(D')^{\mathsf{depth}(D')} \\ &\leq max(2, \mathsf{width}(D'))^{\mathsf{depth}(D')+1}.\end{aligned}$$

Thanks to Lemma 9 and Lemma 10, we deduce that:

$$\#V' \leq (2 + \|\mathsf{out}_P\|^{\mathsf{depth}(G)} \times \|\mathsf{in}_P\|)^{\mathsf{depth}(G)+1}.$$

All these results together yield the expected bound on the number of vertices of the execution graph D of the trace (tr, ϕ). This, in turn, bounds the length of the trace tr but does not take into account the actions $\mathsf{out}(c, c')$ since they do not appear in the execution graph. In other words, we have bounded the number of sessions that involve at least one visible action that is not an output on a channel. Actually, actions of the form $\mathsf{out}(c, c')$ are unnecessary (except possibly one) if they are not followed by some actions on the channel c'. Thus, this gives us a bound on the number of sessions involved in an attack trace. Since reachability for a bounded number of sessions is known to be decidable [34], this allows us to conclude. □

Alternatively, we could also directly bound the total size of a minimal witness tr. To do so, we would need to show that we can bound the size of recipes as well. This would allow us to conclude the proof without relying on the decidability of reachability for a bounded number of sessions.

Example 18 Continuing Example 13, the dependency graph G for the protocol P^1_{DS} is of depth $\mathsf{depth}(G) = 4$. We have $\|\mathsf{out}_{P^1_{\mathsf{DS}}}\| = \|\mathsf{in}_{P^1_{\mathsf{DS}}}\| = 6$, hence the number of vertices of the execution graph is bounded by $(2+6^5)^5$, hence more than 10^{19}.

We see on this very simple example that the bound on the number of sessions is huge and way beyond the reach of existing tools. There is however clearly room for improvements. In particular, we could improve our bound by exploiting the fact our witness is well-typed.

5 Decidability result for equivalence

The goal of this section is to provide the main ingredients of the proof of Theorem 2, that states that trace inclusion is decidable for well-typed protocols that have an acyclic dependency graph. The proof follows a similar structure than the reachability case.

1. We first show that if P is not trace included in Q then there exists a witness of non inclusion that is well-typed, honest-free, and involves only simple recipes.
2. We already know from the reachability case that any such well-typed, honest-free, simple trace can be mapped to a path in the dependency graph.
3. This allows us to compute a bound on the length of such a witness of non inclusion, hence deciding trace inclusion. To compute this bound in the case of equivalence, we provide a new characterisation of static inclusion, following the approach of [13].

5.1 Well-typed witnesses involving simple recipes

As for the reachability case, we first show that we can focus on witnesses of non trace inclusion that have a particular form.

Theorem 4 *Let $\mathcal{K}_P$ be an initial configuration type-compliant w.r.t. (Δ_P, δ_P) and $\mathcal{K}_Q$ be another configuration. We have that $\mathcal{K}_P \not\sqsubseteq_t \mathcal{K}_Q$ if, and only if, there exists a well-typed execution $\mathcal{K}_P \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma_P; i_P)$ involving only simple recipes witnessing this fact. Moreover, we may assume that tr and ϕ are honest-free.*

Proof The existence of a well-typed witness of non-inclusion is a direct consequence of the typing result that has been established in [13]. Then, it remains to justify the fact that we can consider such a witness with simple recipes. To establish this, we consider a well-typed execution $\mathcal{K}_P \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma_P; i_P)$ witnessing this non-inclusion of minimal length, and we denote $\overline{\mathsf{tr}}$ the trace obtained from tr by replacing any recipe R occurring in it by a simple recipe $\overline{R}$ deducing the exact same term as $R\phi{\downarrow}$ (such a recipe exists according to Lemma 4). Our aim is to show that $\overline{\mathsf{tr}}$ is still a witness of non-inclusion.

In case $|\mathsf{tr}| = 0$, we have that $\mathsf{tr} = \overline{\mathsf{tr}}$ and thus the result trivially holds. We consider $\mathsf{tr} = \mathsf{tr}'.\alpha$, and a well-typed execution corresponding to this minimal witness of non-inclusion:

$$\mathcal{K}_P \xRightarrow{\mathsf{tr}'} (\mathcal{P}'; \phi'; \sigma'_P; i'_P) \xRightarrow{\alpha} (\mathcal{P}; \phi; \sigma_P; i_P).$$

Note that $\mathcal{K}_P \xRightarrow{\overline{\mathsf{tr}'}} (\mathcal{P}'; \phi'; \sigma'_P; i'_P) \xRightarrow{\overline{\alpha}} (\mathcal{P}; \phi; \sigma_P; i_P)$ by definition of $\overline{\mathsf{tr}}$ which is computed w.r.t. ϕ. Since tr is minimal and $|\mathsf{tr}'| < |\mathsf{tr}|$, we know that there exists $\mathcal{K}' = (\mathcal{Q}'; \psi'; \sigma'_Q; i'_Q)$ such that $\mathcal{K}_Q \xRightarrow{\mathsf{tr}'} \mathcal{K}'$ and $\phi' \sqsubseteq_s \psi'$. Such a configuration $\mathcal{K}'$ is not necessarily unique. We denote $\mathcal{K}'_1, \ldots, \mathcal{K}'_k$ the configurations that satisfy this requirement, and we denote $\psi'_1, \ldots, \psi'_k$ their associated frame. For each configuration $\mathcal{K}'_i$ (with its associated frame ψ'_i), we have that: for any recipe R occurring in tr', we have that $R\phi'{\downarrow} = \overline{R}\phi'{\downarrow}$, and thus $R\psi'_i{\downarrow} = \overline{R}\psi'_i{\downarrow}$. Therefore, we have that $\overline{\mathsf{tr}'}$ can be executed from $\mathcal{K}_Q$ and leads to the exact same configuration $\mathcal{K}'_i$ as before with frame ψ'_i. In other words, the configurations $\mathcal{K}'_1, \ldots, \mathcal{K}'_k$ are still reachable starting from $\mathcal{K}_Q$ when executing $\overline{\mathsf{tr}'}$.

In case $\overline{\mathsf{tr}}$ is a witness of non-inclusion, we are done. Now, assume that $\overline{\mathsf{tr}}$ passes in $\mathcal{K}_Q$, i.e. $\overline{\alpha}$ can be executed from $\mathcal{K}'_i$ (for some i) and the resulting frame ψ_i is such that $\phi \sqsubseteq_s \psi_i$. In case $\overline{\alpha} = \alpha$, then this means that tr passes in $\mathcal{K}_Q$ and the resulting frame ψ_i is such that $\phi \sqsubseteq_s \psi_i$. This contradicts the fact that tr is a witness of non-inclusion. Otherwise, we have that $\alpha = \mathsf{in}(c, R)$, and $\overline{\alpha} = \mathsf{in}(c, \overline{R})$. Since $\phi' \sqsubseteq_s \psi'_i$, and $R\phi'{\downarrow} = \overline{R}\phi'{\downarrow}$, we deduce that $R\psi'_i{\downarrow} = \overline{R}\psi'_i{\downarrow}$. Therefore, the fact that $\overline{\mathsf{tr}}$ passes in $\mathcal{K}_Q$ leading to frame ψ such that $\phi \sqsubseteq_s \psi$ implies that tr also passes in $\mathcal{K}_Q$ and leads to the exact same frame ψ. This contradicts the fact that tr is a witness of non-inclusion.

It remains to establish that we can assume tr and ϕ to be honest-free. Actually, considering $\mathcal{K}_P \overset{\mathsf{tr}'}{\Longrightarrow} \mathcal{K}'_P$ a witness of non-inclusion with underlying substitution σ'_P, according to Proposition 5.4 stated and proved in [13], we know that the well-typed substitution σ_P is such that $\sigma_P = \sigma_S\rho$ where:

- σ_S is the most general unifier (denoted mgu) of $\Gamma = \{(u, v) \mid u, v \in ESt(\mathcal{K}_0)$ such that $u\sigma_P = v\sigma_P\}$; and
- ρ is a bijective renaming from variables in $dom(\sigma_P) \setminus dom(\sigma_S)$ to some fresh constants preserving type.

In order to conclude, we apply the same reasoning as in the case of reachability, as done at the end of Theorem 3. □

5.2 Exploiting the dependency graph

Thanks to Proposition 1, we know that any well-typed, simple, and honest-free trace can be mapped to a path of the dependency graph. There is nothing to add in the case of trace inclusion.

5.3 Bounding the length of a minimal witness

The last step of the proof of Theorem 2 consists in bounding the size of a (minimal) witness of trace inclusion, that is well-typed, simple, and honest-free. We use a similar technique as in the reachability case, exploiting the dependency graph. However, in case non-inclusion is due to a non static inclusion, pruning the execution w.r.t. a single node is not sufficient. We first establish a bound on the number of nodes involved in a witness of non-inclusion. For this, we have to characterise the form of the test involved in such a witness. We use for that the alternative definition of static inclusion already introduced in [19].

Definition 17 Let ϕ, ψ be such that $dom(\phi) = dom(\psi)$. We write $\phi \sqsubseteq_s^{\mathsf{simple}} \psi$ if:

1. For each destructor-only recipe R such that $R\phi{\downarrow}$ is a (resp. atomic) message, $R\psi{\downarrow}$ is a (resp. atomic) message.
2. For each simple recipe R and destructor-only recipe R' such that $R\phi{\downarrow}, R'\phi{\downarrow}$ are messages and $R\phi{\downarrow} = R'\phi{\downarrow}$, we have that $R\psi{\downarrow} = R'\psi{\downarrow}$.

3. For each destructor-only recipes R, R', if $R\phi\downarrow = \mathsf{sign}(t, s)$, and $R'\phi\downarrow = \mathsf{vk}(s)$ for some term t and atom s, then $R\psi\downarrow = \mathsf{sign}(t', s')$, and $R'\psi\downarrow = \mathsf{vk}(s')$ for some term t' and atom s'.
4. For each destructor-only recipe R, such that $R\phi\downarrow = \mathsf{pub}(s)$ for atom s, $R\psi\downarrow = \mathsf{pub}(s')$ for some atom s'.

As established in [19] for a slightly different set of primitives, this notion of static inclusion is equivalent to the original one.

Lemma 12 *Let ϕ and ψ be two frames having the same domain. We have that:*

$$\phi \sqsubseteq_s \psi \;\Leftrightarrow\; \phi \sqsubseteq_s^{\mathsf{simple}} \psi.$$

Proof It is easy to see that $\phi \sqsubseteq_s \psi \;\Rightarrow\; \phi \sqsubseteq_s^{\mathsf{simple}} \psi$. Indeed, item 1 and item 2 are straightforward. Given two recipes R and R' satisfying the assumptions of item 3, we have $\mathsf{check}(R, R') = \mathsf{ok}$ for ϕ hence for ψ, hence the result. Then given a recipe R satisfying the assumptions of item 4, we have that $\mathsf{aenc}(\mathsf{ok}, R)$ is a message in ϕ, hence in ψ, hence the result. Thus, we only consider the other implication. To establish the other implication, we consider another alternative definition of static inclusion, denoted $\sqsubseteq'_s$. This notion is the same than the one given in Definition 17 but considering arbitrary recipes instead of simple/destructor-only recipes. Clearly, we have that $\phi \sqsubseteq'_s \psi \;\Rightarrow \phi \sqsubseteq_s \psi$, and thus to conclude, it remains to establish $\phi \sqsubseteq_s^{\mathsf{simple}} \psi \;\Rightarrow \phi \sqsubseteq'_s \psi$.

So we now assume $\phi \sqsubseteq_s^{\mathsf{simple}} \psi$ and we show $\phi \sqsubseteq'_s \psi$ by induction on the size of the tests, i.e. the recipes R and R' involved in the test. More precisely, given an arbitrary test T that holds in ϕ w.r.t. the notion $\sqsubseteq'_s$, we show that T also holds in ψ assuming that any test smaller than T have alreasoningady been transferred from ϕ to ψ. We consider the following measure μ where $|R|$ is simply the size of R, i.e. the number of function symbols occurring in it.

1. If T is a recipe (message/atomic message/public key), then $\mu(T) = |R|$
2. If T is made of two recipes R and R' (equality test/signature test), then $\mu(T) = |R| + |R'|$.

We show that the four items of the definition of $\sqsubseteq'_s$ are satisfied.
The test T is a recipe R such that $R\phi\downarrow$ is a message (resp. atomic message).

- Case where R is *not* in normal form w.r.t. $\twoheadrightarrow$. Consider R' such that $R \twoheadrightarrow R'$. We have that $R'\phi\downarrow$ is a message (Lemma 3). By induction hypothesis $R'\psi\downarrow$ is a message too. It remains to show that $R\psi\downarrow$ is a message. Actually, we have that $R = C[\mathsf{adec}(\mathsf{aenc}(R_1, R_2), R_3)]$ and $R' = C[R_1]$ (other cases are similar). Since $R\phi\downarrow$ is a message, we know that $R_2\phi\downarrow = \mathsf{pub}(R_3)\phi\downarrow$. By induction hypothesis $R_2\psi\downarrow = \mathsf{pub}(R_3)\psi\downarrow$, and this allows us to conclude.
- Case where R is in normal form w.r.t. $\twoheadrightarrow$. In this case, we know that R is simple (Lemma 4), i.e. $R = C[R_1, \ldots, R_k]$, where C is a constructor context and R_i are destructor-only recipes. If C is empty, then R is destructor-only.

We conclude by relying on our hypothesis. Otherwise $R = \mathsf{f}(R'_1, \ldots, R'_n)$. By induction hypothesis, we know that $R'_i\psi\downarrow$ is a message ($1 \leq i \leq n$). We have to prove that $C[R_1, \ldots, R_k]\psi\downarrow = \mathsf{f}(R'_1, \ldots, R'_n)\psi\downarrow$ is a message. We have atomic messages at key positions (thanks to our induction hypothesis). In case $\mathsf{f} = \mathsf{aenc}$ (resp. $\mathsf{f} = \mathsf{raenc}$) and thus $n = 2$, we have to ensure that $R'_2\psi\downarrow$ is of the form $\mathsf{pub}(s)$. This is given by item 4 of Definition 17.

The test T is of the form $R = R'$ such that R and R' are recipes, $R\phi\downarrow$, $R'\phi\downarrow$ are messages, and $R\phi\downarrow = R'\phi\downarrow$.

- Case R (resp. R') is *not* in normal form w.r.t. $\twoheadrightarrow$. Let $R'' = R\downarrow$. Since $R\phi\downarrow$ and $R\psi\downarrow$ are messages, we deduce that $R''\phi\downarrow = R\phi\downarrow$ and $R''\psi\downarrow = R\psi\downarrow$. We have that $R''\phi\downarrow = R\phi\downarrow = R'\phi\downarrow$. Relying on our induction hypothesis applied on the test $R'' = R'$, we deduce that $R''\psi\downarrow = R'\psi\downarrow$, and thus $R\psi\downarrow = R'\psi\downarrow$.
- Otherwise, thanks to Lemma 4, we know that R and R' are simple, i.e. $R = C[R_1, \ldots, R_k]$ and $R' = C'[R'_1, \ldots, R'_\ell]$, where C, C' are constructor contexts and R_i ($1 \leq i \leq k$) as well as R'_j ($1 \leq j \leq \ell'$) are destructor-only recipes. If neither C nor C' is empty (that is, neither R nor R' is destructor-only) then $\mathsf{root}(R) = \mathsf{root}(R')$, and thus we conclude relying on our induction hypothesis. Otherwise, we conclude relying on our hypothesis that $\phi \sqsubseteq_s^{\mathsf{simple}} \psi$.

The test T is of the form $R = R'$ such that R and R' are recipes, $R\phi\downarrow = \mathsf{sign}(t, s)$, and $R'\phi\downarrow = \mathsf{vk}(s)$ for some term t and some atom s.

- Case R (resp. R') is *not* in normal form w.r.t. $\twoheadrightarrow$. $R\downarrow$ (resp. $R'\downarrow$) is a smaller recipe than R (resp. R'). By Lemma 3, $R\downarrow\phi\downarrow = R\phi\downarrow$ (resp. $R'\downarrow\phi\downarrow = R'\phi\downarrow$). So $R\downarrow, R'$ (resp. $R, R'\downarrow$) gives us a smaller test than R, R'. By induction hypothesis we get that $R\downarrow\psi\downarrow = \mathsf{sign}(t', s')$ and $R'\psi\downarrow = \mathsf{vk}(s')$ for some term t' and atom s'. We already considered the case where $R\phi\downarrow$ is a message, so we can assume $R\psi\downarrow$ is a message. Then we deduce that $R\psi\downarrow = R\downarrow\psi\downarrow$ by Lemma 3, and concluding this case.
- Otherwise, thanks to Lemma 4, we know that R and R' are both simple recipes. In case they are both destructor-only recipes we conclude relying on our hypothesis. Otherwise, assume first $R = \mathsf{sign}(R_1, R_2)$. In such a case, we have that the test $\mathsf{vk}(R_2) = R'$ is smaller than $R = R'$, and it holds in ϕ, and thus it can be transferred from ϕ to ψ by induction hypothesis. Moreover, $R_2\psi\downarrow$ is an atom by induction hypothesis. Hence, we have that $R\psi\downarrow = \mathsf{sign}(R_1\psi\downarrow, R_2\psi\downarrow)$, and $R'\psi\downarrow = \mathsf{vk}(R_2\psi\downarrow)$. Hence, the result. Now, assume that R is destructor-only and R' is not, i.e. $R' = \mathsf{vk}(R'_1)$. Since $R'_1\phi\downarrow$ is an atomic message, R'_1 is destructor only, and thus $\mathsf{sign}(\mathsf{getmsg}(R), R'_1)$ is simple. We know that $\mathsf{sign}(\mathsf{getmsg}(R), R'_1) = R$ holds in ϕ. By hypothesis, it also holds in ψ. This allows us to conclude that $R\psi\downarrow = \mathsf{sign}(t', s')$ with $R'_1\psi\downarrow = s'$. Hence, the result.

The test T is a recipe R such that $R\phi\downarrow = \mathsf{pub}(s)$ for some atom s.

- Case R is *not* in forced normal form, we have that $R{\downarrow}$ is a smaller recipe than R. By Lemma 3, $R{\downarrow}\phi{\downarrow} = R\phi{\downarrow}$. So by induction hypothesis $R{\downarrow}\psi{\downarrow} = \mathsf{pub}(s)$ for some atom s. We have already proved that, as $R\phi{\downarrow}$ is a message, $R\psi{\downarrow}$ is a message. So by Lemma 3, $R{\downarrow}\psi{\downarrow} = R\psi{\downarrow} = \mathsf{pub}(s)$.
- Case R is a simple recipe. In case R is a destructor-only recipe, we conclude relying on our hypothesis. Otherwise $R = \mathsf{pub}(R_1)$ and $R_1\phi{\downarrow}$ is an atom, thus R_1 is destructor-only. We conclude that $R_1\psi{\downarrow}$ is an atom too relying on our induction hypothesis, and thus $R\psi{\downarrow} = \mathsf{pub}(s')$ for some atom s'.

□

We use this new characterisation of static inclusion to bound the size of a minimal witness.

Lemma 13 *Let P be a simple protocol type-compliant w.r.t. some typing system (Δ_P, δ_P), and let G be its associated dependency graph which is supposed to be acyclic. Let Q be another simple protocol such that $P \not\sqsubseteq Q$. Let (tr, ϕ) be a witness of non-inclusion which is well-typed and with minimal length. We have that:*

$$\mathsf{nbroot}(D) \leq 2 \times (1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$$

where D is the execution graph associated to (tr, ϕ) w.r.t. P.

Proof Thanks to Theorem 4, we know the existence of a well-typed witness of non-inclusion, and we choose one having a minimal length. We denote D the execution graph associated to (tr, ϕ) w.r.t. P. We distinguish two cases.

There does not exist ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$. In such a case, we have that $\mathsf{tr} = \mathsf{tr}' \cdot \alpha$. This last action α is necessarily a visible action. In case, it corresponds to an input (resp. output) of a message (not a channel name), then we prune D w.r.t. this single action. We denote D' the resulting execution graph and (tr_0, ϕ_0) the corresponding trace of P (the one mentioned in Lemma 11). Actually, by definition of pruning, we have that $\mathsf{tr}_0 = \mathsf{tr}'_0 \cdot \alpha$ where tr'_0 is the trace obtained by pruning D w.r.t. the set of nodes R_α corresponding to all the dependencies of α. We have that tr'_0 passes in P and also in Q by minimality of the witness tr. Assume now that $\mathsf{tr}'_0.\alpha$ does not pass in Q. Then, we have built a smaller witness of non inclusion (contradiction), unless $D = D'$. In the latter case, we are done since $\mathsf{nbroot}(D) = \mathsf{nbroot}(D') = 1$. Otherwise, we have that $\mathsf{tr}'_0.\alpha$ passes in Q meaning that α is available after the execution of tr'_0 and we can show that this action is still there after the execution of tr'. Thus, contradiction.

In case, this last action corresponds to an output of a channel name then we have that $\mathsf{out}(c, c'')$ is available in P and not in Q, and due to the form of our processes (they are simple), we have that $(\mathsf{tr}, \phi) = (\mathsf{out}(c, c''), \emptyset)$ is a witness of non-inclusion, and its associated execution graph D' is such that $\mathsf{nbroot}(D') = 0$ (such a graph is empty).

There exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ *but* $\phi \not\sqsubseteq_s \psi$. From Lemma 12, we can consider distinguishing tests that satisfy Definition 17. We now compute a bound b on the number of distinct variables that may occur in such a test.

- In case the test involved one (resp. two) destructor-only recipe(s), following the proof of Lemma 10, we can show that a destructor-only recipe contains at most $\|\mathsf{out}_P\|^{\mathsf{depth}(G)}$ variables, and this leads us to the bound $b = 2 \times \|\mathsf{out}_P\|^{\mathsf{depth}(G)}$.
- In case the test involved a simple recipe R. Since we know that $R\phi\downarrow = R'\phi\downarrow$ with R' destructor-only, and (tr, ϕ) is a well-typed witness of non-inclusion, we deduce that $R = C[R_1, \ldots, R_k]$ with $k \leq \|R'\phi\downarrow\| \leq \|\mathsf{out}_P\|$. Therefore, we deduce that such a simple recipe involved at most $(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)} \times \|\mathsf{out}_P\|$ distinct variables, and this leads us to the bound $b = (1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$.

Let W be the set of all the variables occurring in the test witnessing the non-inclusion. We have seen that $\#\mathsf{W} \leq b$. We consider the actions of tr labelled with $\mathsf{out}(c, \mathsf{w})$ with $\mathsf{w} \in \mathsf{W}$, and we prune D w.r.t. this set of actions. Let D' be the resulting execution graph. We obtain $(\mathsf{tr}_0, \phi_0) \in \mathsf{trace}(P)$ (by Lemma 11). Note that the definition of pruning does not depend on the underlying protocol, and thus, since $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$, we have also that D' is the execution graph of tr_0 w.r.t. Q, and ψ_0 is a subframe of ψ. Actually, we have that there exists W' such that $\mathsf{W} \subseteq \mathsf{W}'$, $\phi_0 = \phi|_{\mathsf{W}'}$, and $\psi_0 = \psi|_{\mathsf{W}'}$. This allows us to ensure that (tr_0, ϕ_0) with execution graph D' is still a witness of non-inclusion which satisfies our requirements. □

We can now conclude that trace inclusion is decidable.

Theorem 2 *Le P be a simple protocol type-compliant w.r.t. some typing system (Δ_P, δ_P) and with an acyclic dependency graph. Let Q be another simple protocol. The problem of deciding whether P is trace included in Q is decidable.*

Proof Let G be the dependency graph associated to P. To establish this result, we show that there is a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ witnessing this fact whose execution graph $D = (V, E)$ is such that:

$$\#V \leq 2(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1} \times (2 + \|\mathsf{out}_P\|^{\mathsf{depth}(G)} \times \|\mathsf{in}_P\|)^{\mathsf{depth}(G)+1}.$$

First, thanks to Theorem 4, we consider a well-typed witness (tr, ϕ) of non-inclusion which is also honest-free and that only involve simple recipes, and we consider one of minimal length. Thanks to Lemma 13, we know that:

$$\mathsf{nbroot}(D) \leq 2(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$$

where D is the execution graph associated to (tr, ϕ) w.r.t. P.

We aim at bounding the number of vertices in D. Actually, we have that:

$$\begin{aligned}\#V &\leq \mathsf{nbroot}(D)(1 + \mathsf{width}(D) + \mathsf{width}(D)^2 + \ldots + \mathsf{width}(D)^{\mathsf{depth}(D)}) \\ &\leq \mathsf{nbroot}(D) \times max(2, \mathsf{width}(D))^{\mathsf{depth}(D)+1}\end{aligned}$$

Thanks to Lemma 9 and Lemma 10, we know that:

- $\mathsf{width}(D) \leq 1 + \|\mathsf{out}_P\|^{\mathsf{depth}(G)} \times \|\mathsf{in}_P\|$, and

– $\mathsf{depth}(D) \leq \mathsf{depth}(G)$.

All these results together yield the expected bound on the number of vertices of the execution graph D of the trace (tr, ϕ). This, in turn, bounds the length of the trace tr up to the actions $\mathsf{out}(c, c')$ since they do not appear in the execution graph. As for the reachability case, this gives us a bound on the number of sessions. Hence it is sufficient to decide trace equivalence for a bounded number of sessions, which is known to be decidable [6,12]. □

6 An improved version of our decidability result

Unfortunately, our initial definition of dependency graph often yields to cyclic graphs (actually, in most cases!). Hence we devise a refined dependency graph such that our results still hold. The idea is to *mark* some positions of output actions of a protocol P and disallow arrows that point to such positions. Recall that many arrows of the dependency graph aim at identifying, for each input action, which output actions could be used to build the input message. We show that it is sound to remove arrows that points to marked positions, as long as we know that terms appearing at such positions in an output can already be deduced from earlier messages. Such a marking is then called *appropriate*. We provide two simple syntactic criteria that, when satisfied, guarantees that a marking is appropriate.

6.1 Simple asap recipes

To show that our results still hold for our refined notion of dependency graph, we will first show that we can consider executions where recipes are not only simple but also *asap*, that is, to build a message, a recipe should use messages that have been introduced as early as possible.

Definition 18 Let ϕ be a frame with a total ordering $<$ on $dom(\phi)$, and m be a message such that $R\phi\downarrow = m$. We say that R is an *asap recipe* of m if R is minimal among the recipes $\{R' \mid R'\phi\downarrow = m\}$ for the following measure: for any two recipes R and R', it is the case that $R < R'$ if $vars^{\#}(R) <_{\text{mul}} vars^{\#}(R')$, where $vars^{\#}(R)$ denotes the multiset of variables occurring in R, and $<_{\text{mul}}$ is the multiset extension of $<$.

Example 19 Consider the frame $\phi = \{\mathsf{w}_1 \triangleright \langle k, k' \rangle, \mathsf{w}_2 \triangleright k\}$. Then the recipe w_2 allows to deduce k since $\mathsf{w}_2\phi\downarrow = k$ but is not asap. Instead $\mathsf{proj}_1(\mathsf{w}_1)$ is an asap recipe of k.

Whenever a message is deducible, we can also find a simple and asap recipe of the message.

Lemma 14 *Let ϕ be a frame (with a total ordering on $dom(\phi)$) and u be a message deducible from ϕ, i.e. such that $R\phi\downarrow = u$ for some R. We have that there exists R' simple and asap such that $R'\phi\downarrow = u$.*

Proof We first chose among all the recipes $\{R' \mid R'\phi{\downarrow} = u\}$, one which is minimal. Let R_0 be such a recipe, and then we consider $R_0{\downarrow}$. Thanks to Lemma 4, we have that $R_0{\downarrow}$ is simple, and it is a recipe for u. It is also asap since $R_1 \twoheadrightarrow R_2$ implies that $R_2 \leq R_1$. This allows us to conclude. □

We show that, for both reachability and equivalence, we can consider witnesses that are well-typed, honest-free, and that involve simple and asap recipes.

Theorem 5 *Let $\mathcal{K}_0$ be an initial configuration type-compliant w.r.t. (Δ_0, δ_0). If $\mathcal{K}_0 \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$ then there exists a well-typed execution $\mathcal{K}_0 \overset{\mathsf{tr}'}{\Longrightarrow} (\mathcal{P}; \phi'; \sigma'; i)$ involving only simple asap recipes such that $\overline{\mathsf{tr}'} = \overline{\mathsf{tr}}$. Moreover, we may assume that tr' and ϕ' are honest-free.*

Proof This theorem is a consequence of the typing result that has been established in [13], and can be established following the same lines as the proof of Theorem 3. To justify the fact that we can consider simple asap recipes, we rely on Lemma 14 (instead of Lemma 4). Then, it remains to justify that tr' and ϕ are honest-free, and the proof is similar to one of Theorem 3. □

Theorem 6 *Let $\mathcal{K}_P$ be an initial configuration type-compliant w.r.t. (Δ_P, δ_P) and $\mathcal{K}_Q$ be another configuration. We have that $\mathcal{K}_P \not\sqsubseteq_t \mathcal{K}_Q$ if, and only if, there exists a well-typed execution $\mathcal{K}_P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma_P; i_P)$ involving only simple asap recipes witnessing this fact. Moreover, we may assume that tr and ϕ are honest-free.*

Proof This theoroem is a consequence of the typing result that has been established in [13], and can be established following the same lines as the proof of Theorem 4. To justify the fact that we can consider simple asap recipes, we rely on Lemma 14 (instead of Lemma 4). Then, it remains to justify that tr and ϕ are honest-free, and the proof is similar to the one of Theorem 4. □

6.2 Marking (semantic criterion) and refined dependency graph

We first devise a general (semantic) criterion in order to mark some output actions of a protocol and remove accordingly some of the edges of the dependency graph.

Definition 19 A *marked position* of a simple protocol P w.r.t. a typing system (Δ_0, δ_0) is a pair (ℓ, p) where $\mathsf{out}^\ell(c, u)$ is an output action occurring in P, and p is a position of the term $\delta_0(u)$. A *marking* of P w.r.t. (Δ_0, δ_0) is a set of marked positions of P.

We consider that a marking strategy is appropriate for our dependency graph if it indicates subterms that, whenever deducible in a well-typed execution, are deducible earlier in any well-typed execution. This will guarantee that it is sound to remove arrows pointing to marked positions.

Definition 20 Let P be a simple protocol. A marked position (ℓ, p) of P w.r.t. (Δ_0, δ_0) is *appropriate* if for any well-typed execution $P \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; j)$, for any $\mathsf{out}^{\ell}(c, \mathsf{w})$ occurring in tr, for any destructor-only recipe R with w at its leftmost position and such that $\mathsf{target}(R) = p$ and $R\phi{\downarrow} = m$ is a message, we have that R is not an asap recipe of m (considering the frame ϕ and the ordering induced by tr).

Deciding whether a marked position is appropriate is not an easy task. We will provide some syntactic criteria in the following section. Relying on this notion of marking, we are now able to define our notion of refined dependency graph associated to an initial configuration $\mathcal{K}_0$. It is simply obtained by removing arrows pointing to marked positions.

Definition 21 Let (Δ_0, δ_0) be a typing system, P be a simple protocol, and $\mathcal{M}$ be a marking of P w.r.t. (Δ_0, δ_0). The *refined dependency graph* associated to P and $\mathcal{M}$ is obtained from the dependency graph of P by simply removing any arrow of the form $\ell \rightarrow^{p} \ell'$ for which $(\ell', p) \in \mathcal{M}$.

We can again link dependencies arising in executions of a protocol to its refined dependency graphs. More precisely, we show that any dependency arising in an execution can be mapped to a path in the (refined) dependency graph provided that the underlying execution is well-typed, honest-free, and that it involves simple and asap recipes.

Proposition 2 *Let (Δ_0, δ_0) be a typing system, P be a simple protocol with $\mathcal{M}$ an appropriate marking, (Δ_0, δ_0) be a typing system, and $P \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; j_0)$ be a well-typed honest-free execution involving simple asap recipes.*

For any pair of actions $\mathsf{in}^{\ell_{\mathsf{in}}}(d, C[R_1, \ldots, R_k])/\mathsf{out}^{\ell_{\mathsf{out}}}(c, \mathsf{w})$ occurring in tr with $\mathsf{w} \in vars(R_{i_0})$ $(1 \leq i_0 \leq k)$, we have that:

$$\ell_{\mathsf{in}} \rightarrow \ell^0_{\mathsf{out}} \rightarrow^* \ell_{\mathsf{out}} \text{ in the refined dependency graph associated to } P$$

where ℓ^0_{out} is the label associated to w_0, the handle occurring at the leftmost position in R_{i_0}. Moreover, the length of the path from ℓ^0_{out} to ℓ_{out} is equal to the number of occurrences of 2 in p (the position at which w occurs in R_{i_0}).

Proof The proof is similar to the one performed to establish Proposition 1. We follow the same reasoning but we now have to justify in addition that the arrow we consider is not removed by marking. We perform the proof by induction on the number of occurrences of 2 in p (the position at which w occurs in R_{i_0}.

1. Following the proof of Proposition 1, we conclude the existence of an edge $\ell_{\mathsf{in}} \rightarrow^{\mathsf{target}(R_{i_0})} \ell_{\mathsf{out}}$ of type 2 in the dependency graph. Now, we have to justify that this edge is still present in the refined dependency graph. Assume by contradiction that it is not the case, i.e. $(\ell_{\mathsf{out}}, \mathsf{target}(R_{i_0})) \in \mathcal{M}$. Since $\mathcal{M}$ is appropriate, we easily deduce that R_{i_0} is not an asap recipe of $R_{i_0}\phi{\downarrow}$, and thus R (occurring in tr) is not asap. This leads to a contradiction.

2. Following the proof of Proposition 1, we conclude the existence of an edge $\ell'_{\mathsf{out}} \rightarrow^{\mathsf{target}(R_{i_0}|_{p_0.2})} \ell_{\mathsf{out}}$ of type 3 in the dependency graph – where ℓ'_{out} is the label associated to w_{p_0}. Now, we have to justify that this edge is still present in the refined dependency graph. Assume by contradiction that it is not the case, i.e. $(\ell_{\mathsf{out}}, \mathsf{target}(R_{i_0})) \in \mathcal{M}$. Since $\mathcal{M}$ is appropriate, we easily deduce that $R_{i_0}|_{p_0.2}$ is not an asap recipe of $R_{i_0}|_{p_0.2}\phi\downarrow$, and thus R (occurring in tr) is not asap. This leads to a contradiction.

We obtain the expected result. □

6.3 Marking - syntactic criteria

We can use any marking that is appropriate. However, checking that a particular marking is appropriate is far from easy and is actually very likely undecidable. So instead, we provide two syntactic criteria that allow to mark a position and we prove that such a marking is appropriate.

Our first criterion allows us to remove arrows towards terms having a public type. This notion is defined below.

Definition 22 Given a simple protocol P and a typing system (Δ_0, δ_0). A type τ_p is *public* if for any name n occurring in P, we have that $\delta_0(n) \notin St(\tau_p)$.

The intuition is that, in a well-typed execution, terms having a public type are terms built using public constants only, and are thus deducible from the beginning of the execution. This is formally established in the following lemma.

Lemma 15 *Let P be a simple protocol, (Δ_0, δ_0) be a typing system, and u be a term having a public type. Let $P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$ be a well-typed execution such that $R\phi\downarrow = u$ for some recipe R, then $u \in \mathcal{T}(\Sigma_{\mathsf{c}}, \Sigma_0)$.*

Proof Let τ_p be the type of u. In order to establish that $u \in \mathcal{T}(\Sigma_{\mathsf{c}}, \Sigma_0)$, we show that each leaf of $u \in \Sigma_0 \cup \{\mathsf{ok}\}$. Consider an arbitrary leaf a of u. We have that a is either a name or a constant. In case $a \in \Sigma_0$, then we are done. Now assume that $a \in \mathcal{N}$, and thus we have that $\delta_0(a) \in St(\tau_p)$. Since $R\phi\downarrow = u$ for some recipe R, we have that a occurs somewhere in ϕ and thus a name n such that $\delta_0(n) = \delta_0(a)$ occurs in P. Therefore we have that $\delta_0(n) \in St(\tau_p)$ for some name n occurring in P. This is impossible by definition of being a public type, and this allows us to conclude. □

We conclude that marking a position that has a public type is appropriate.

Lemma 16 *Let (ℓ, p) be a marked position of a simple protocol P w.r.t. a typing system (Δ_0, δ_0). Let u be the term such that $\mathsf{out}^{\ell}(c, u)$ occurs in P. If $\delta_0(u)|_p$ has a public type then (ℓ, p) is appropriate.*

Proof We consider a well-typed execution $P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$ and $\mathsf{out}^{\ell}(c, \mathsf{w})$ occurring in tr. Let R be a destructor-only recipe with w at its leftmost position, and such that $\mathsf{target}(R) = p$ and $R\phi\downarrow = m$ is a message. By definition of

$\mathsf{target}(R)$, we have that $R\phi\downarrow = \mathsf{w}\phi|_p$, and since we are considering a well-typed execution, we know that $\delta_0(\mathsf{w}\phi|_p) = \delta_0(\mathsf{w}\phi)|_p$ has public type. Thus, we have that $\mathsf{w}\phi|_p \in \mathcal{T}(\Sigma_\mathsf{c}, \Sigma_0)$ thanks to Lemma 15, and therefore R (which contains w) is not an asap recipe for $R\phi\downarrow = m = \mathsf{w}\phi|_p$. □

Example 20 Going back to our running example, we may decide to declare τ_{skc} as a public type. Indeed, there is no name in P^1_{DS} having such a type. However, this will not change the resulting dependency graph as no term having such a public type is extractable from an output of the protocol.

Our second criterion is a *precedence criterion*: a position p of an output action ℓ can be safely marked if there exists a previous action ℓ' such that the term at position p in ℓ can always be accessed in ℓ' with a smaller set of keys. We use this criterion to obtain acyclic graphs for several protocols such as the Needham-Schroeder symmetric key protocol, as explained in the next section.

Lemma 17 *Let (ℓ, p) be a marked position of a simple protocol P w.r.t. a typing system (Δ_0, δ_0) such that there exists an action labelled ℓ' in P with:*

- $\mathsf{out}^\ell(c, u)$ *follows the action ℓ' involving term v in P ;*
- $(u|_p, p)\#(S; A) \in \rho_{\mathsf{out}}(u)$ *for some S and some A;*
- $(u|_p, q)\#(S'; A') \in \rho_{\mathsf{out}}(v)$ *for some q, S' and A' such that $S' \subseteq_{\mathsf{mul}} S$ and $A' \subseteq_{\mathsf{mul}} A$.*

We have that $(\ell, p.p')$ is appropriate for any p' such that $\delta_0(u)|_{p.p'}$ is well-defined.

Intuitively, the sets S and A represent the keys needed to access $u|_p$ following the path p, while the sets S' and A' represent the keys needed to access $u|_p$ in v following the path q. Hence, if action ℓ' precedes action ℓ and if $u|_p$ can be accessed more easily in action ℓ' than in action ℓ (using only keys of $S' \cup A'$ rather than keys in $S \cup A$), then it is not longer necessary to take into account what can be built from action ℓ at position p.

Proof We consider a well-typed execution $P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$ and an action $\mathsf{out}^\ell(c, \mathsf{w})$ occurring in tr. Let R be a destructor-only recipe with w at its leftmost position, and such that $\mathsf{target}(R) = p.p'$ and $R\phi\downarrow$ is a message. By definition of $\mathsf{target}(R)$, we have that $R\phi\downarrow = \mathsf{w}\phi|_{p.p'}$. In order to conclude, we want to show that R is not an asap recipe for $R\phi\downarrow = \mathsf{w}\phi|_{p.p'}$.

Let R_0 be $R|_{1\ldots1}$ such that $\mathsf{target}(R_0) = p$. We have that $R_0\phi\downarrow = \mathsf{w}\phi|_p$, and we denote by $R^1_{\mathsf{key}}, \ldots, R^j_{\mathsf{key}}$ recipes occurring at position of the form $1 \ldots 1.2$ in R_0. According to our hypothesis, we know that $\mathsf{in}^{\ell'}(\_, R_{\mathsf{in}})$ (resp. $\mathsf{out}^{\ell'}(\_, \mathsf{w}')$) occurs before $\mathsf{out}^\ell(c, \mathsf{w})$ in tr with $R_{\mathsf{in}}\phi\downarrow|_q = \mathsf{w}\phi|_p$ in case ℓ' corresponds to an input (resp. $\mathsf{w}'\phi|_q = \mathsf{w}\phi|_p$ in case ℓ' corresponds to an output). Moreover, reusing some elements of the multiset $\{R^1_{\mathsf{key}}, \ldots, R^j_{\mathsf{key}}\}$, we can build a recipe starting from R_{in} (resp. w') and adding destructors in order to extract the subterm at position q in $R_{\mathsf{in}}\phi\downarrow$ (resp. $\mathsf{w}'\phi$) using elements of the multiset $\{R^1_{\mathsf{key}}, \ldots, R^j_{\mathsf{key}}\}$ at key positions. We denote $\overline{R_0}$ such a recipe. We have that

$\overline{R_0}$ is smaller than R_0 since we replace one occurrence of w (the one occurring at the leftmost position in R_0) by a smaller recipe (R_{in} or w'), and regarding recipes occurring at position $1 \dots 1.2$ in $\overline{R_0}$, they form a submultiset of those occurring at position $1 \dots 1.2$ in R_0. Note that $\overline{R_0}\phi{\downarrow} = R_0\phi{\downarrow} = \mathsf{w}\phi|_p$. Thus, we have that R_0 is not an asap recipe for $\mathsf{w}\phi|_p$, and therefore R (recipe which contains R_0) is not an asap recipe for $R\phi{\downarrow} = \mathsf{w}\phi|_{p.p'}$. □

Our syntactic criteria are easy to check and it would be also easy to automate the marking of a graph following these two criteria.

6.4 Main results

It remains to conclude that we can decide reachability and trace equivalence whenever the *refined* dependency graph is acyclic. Thanks to Proposition 2 that ensures that a minimal well-typed, honest-free, simple and asap trace can be mapped to the refined dependency graph, the remaining proof can be adapted in a straightforward way.

First, we have the analog of Lemma 9 and Lemma 10, that is, we can bound the depth and width of the refined dependency graph of a minimal well-typed, honest-free, simple and asap trace, thanks to Proposition 2.

Lemma 18 *Let (Δ_0, δ_0) be a typing system and P be a simple protocol whose associated refined dependency graph G is acyclic. Let $P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i_0)$ be a well-typed honest-free execution involving simple asap recipes, and D its associated execution graph. We have that:*

$$\mathsf{depth}(D) \leq \mathsf{depth}(G).$$

Lemma 19 *Let (Δ_0, δ_0) be a typing system, and P be a simple protocol whose associated refined dependency graph G is acyclic. Let $P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i_0)$ be a well-typed honest-free execution involving simple asap recipes, and D its associated execution graph. We have that:*

$$\mathsf{width}(D) \leq 1 + \|\mathsf{out}_P\|^{\mathsf{depth}(G)} \times \|\mathsf{in}_P\|.$$

Decidability of reachability and trace equivalence then follow.

Theorem 7 *Let P be a simple protocol type-compliant w.r.t. some typing system (Δ_0, δ_0) and with an acyclic refined dependency graph. Let $\ell \in \mathcal{L}(P)$. The problem of deciding whether ℓ is reachable in P, i.e. whether*

$$P \overset{\mathsf{tr}}{\Longrightarrow} (\{i : \mathsf{io}^{\ell}(c, u).Q\} \cup \mathcal{P}; \phi; \sigma; i)$$

for some trace tr, and $\mathsf{io} \in \{\mathsf{in}, \mathsf{out}\}$ is decidable.

Proof The proof follows the same lines than the proof of Theorem 1. Theorem 5 ensures the existence of a witness that only involves asap recipes. Then, we rely on Lemma 18 and Lemma 19 to establish the bound leading to the decidability result. □

Theorem 8 *Le P be a simple protocol type-compliant w.r.t. some typing system (Δ_P, δ_P) and with an acyclic refined dependency graph. Let Q be another simple protocol. The problem of deciding whether P is trace included in Q is decidable.*

Proof The proof follows the same lines than the proof of Theorem 2. Theorem 6 ensures the existence of a witness that only involves asap recipes. Then, we rely on Lemma 18 and Lemma 19 to establish the bound leading to the decidability result. Note that Lemma 13 still applies and does not need to be adapted. □

7 Case studies

We have considered several protocols of the literature, to study whether our decidability result applies to them or not. More precisely, we have checked whether their dependency graph is acyclic or not and checked whether they are type-compliant.

7.1 Properties

We have considered two main security properties depending on the protocol. The first one is *strong secrecy*: a nonce or a key n is strongly secret if an attacker cannot learn any information about n. Following the game-based approach used in computational models, this has been modelled [4] as follows. Even if the attacker knows the possible values for n, say a or b, she should not be able to distinguish whether the value a or the value b is exchanged, even if a and b are public. If P models a protocol, this is formally expressed as

$$P(a/n) \approx_t P(b/n).$$

For example, we have modelled strong secrecy of our running example in Example 6.

However, strong secrecy is too strong to define the security of a key k, as soon as the key is used to encrypt. Indeed, imagine that the key k is used to encrypt some message m, that is, the message $\mathsf{senc}(m, k)$ is sent at some point. Then requiring strong secrecy of k would require at least that an attacker cannot distinguish between $\mathsf{senc}(m, a)$ and $\mathsf{senc}(m, b)$ where a and b are public values, that is, it would require that:

$$\{\mathsf{w}_1 \triangleright \mathsf{senc}(m, a)\} \sim_s \{\mathsf{w}_1 \triangleright \mathsf{senc}(m, b)\}.$$

But these frames can be distinguished with the test $\mathsf{senc}(\mathsf{sdec}(\mathsf{w}_1, a), a) = \mathsf{w}_1$.

Therefore, the security of such a key is expressed as *key privacy*. Intuitively, a key k is secure if an attacker cannot learn any information on messages that are encrypted by k. This can be intuitively modelled as:

$$\{\mathsf{w}_1 \triangleright \mathsf{senc}(a, k)\} \sim_s \{\mathsf{w}_1 \triangleright \mathsf{senc}(b, k)\}.$$

Moreover, the key k should be indistinguishable from a fresh key. In particular, the attacker should not be able to detect if the same key is used to encrypt all messages or if a fresh key is used each time. These two properties are encoded in a single equivalence, by requiring trace equivalence of P and Q where P represents the protocol that additionally sends $\mathsf{senc}(a, k)$ and Q represents the protocol that additionally sends $\mathsf{senc}(b, k')$ where k' is a fresh key.

7.2 Protocols and scenarios

We have considered a dozen of protocols of the literature, that use symmetric or asymmetric encryption, and possibly signatures. We do not recall the protocols here since most of them are standard and their description can be found in the literature, except for the passport case (see next section). For each protocol, we have considered a scenario where each role is instantiated by all possible players among 2 honest agents a and b and a dishonest one c. For example, if $\mathsf{RoleInit}(x, y)$ represents one session of the initiator role, where agent x talks to agent y, then we consider $\mathsf{RoleInit}(a, b)$, $\mathsf{RoleInit}(b, a)$, $\mathsf{RoleInit}(a, c)$, and $\mathsf{RoleInit}(b, c)$. We do not consider the cases where a dishonest agent communicates with other agents since this can already be simulated by the attacker. And we proceed similarly for the other roles of the protocols. Public-key protocols typically include two roles (initiator and responder) while symmetric key usually also involve a server. This corresponds to a total of 8 processes in the asymmetric case (4 for RoleInit, 4 for RoleResp, the role of the responder) and 14 processes in the symmetric case (4 for RoleInit, 4 for RoleResp, and 6 for RoleS, the role of the server). Then each of the processes is considered under replication. In other words, we consider the protocols with an unbounded number of sessions of each role, but with only two honest agents and one dishonest agent. In order to consider an unbounded number of agent names, we would need to model a more complex protocol with key generation, which may go beyond our class. An alternative approach is to rely in [18] that shows how to bound the number of agents, at the price of considering slightly more agents (2 honest and 2 dishonest agents) as well as some additional scenarios when an agent talks to itself.

7.3 Passport protocols

We describe the three less standard protocols that we have considered, namely the Basic Access Control (BAC) protocol, the Passive Authentication protocol (PA), and the Active Authentication protocols (AA). The three protocols are part of the protocol suite embedded in the biometric passport [1]. They are run to authenticate the passport (and the passport holder, thanks to biometric data) to the reader. We present here the core protocols, as described in [30].

	tagged	property	public types	precedence	acyclicity
Symmetric protocols					
Denning-Sacco		Kpriv	★		✓
Needham-Schroeder		Kpriv	★	★	✓
Otway-Rees	★	Ssec	★	★	✓
Wide-Mouth-Frog	★	Ssec	★		✓
Kao-Chow (variant)	★	Kpriv	★	★	✓
Yahalom-Paulson	★	Kpriv	★	★	✓
Yahalom-Lowe	★	Kpriv	★		✗
Asymmetric protocols					
Running example		Ssec			✓
Denning-Sacco with signature		Kpriv	★		✓
Needham-Schroeder	★	Ssec	★		✗
Needham-Schroeder-Lowe	★	Ssec	★		✓
Passport					
BAC	★	Unlink			✓
Passive Authentication		Unlink			✓
Active Authentication		Unlink			✓

Ssec: strong secrecy Kpriv: key privacy

Table 1 Acyclicity of the dependancy graphs of protocols of the literature.

Basic Access Control. The BAC protocol aims at exchanging a session key between the Reader (R) and the Passport (P), used in subsequent communications. It assumes that the reader and the passport already share an encryption key k_e and a MAC key k_m, that the reader derives from a seed read optically on the first page of the passport.

$$
\begin{aligned}
&R \to P : \mathsf{getChall}\\
&P \to R : n_P\\
&R \to P : \mathsf{senc}((n_R, n_P, k_R), k_e), \mathsf{hash}(\mathsf{senc}((n_R, n_P, k_R), k_e), k_m)\\
&P \to R : \mathsf{senc}((n_P, n_R, k_P), k_e), \mathsf{hash}(\mathsf{senc}((n_P, n_R, k_P), k_e), k_m)
\end{aligned}
$$

At the end of the exchange, the reader and the passport share two session keys k_R and k_P that are used to derive an encryption session key *ksenc* and a MAC session key *kmac*. Actually, the passport additionally sends error messages when it receives ill-formed messages (for example if the MAC check fails). However, due to the restriction of our model (no else branch), we do not model this part of the protocol.

We analyse *unlinkability* of the BAC protocol. Unlinkability intuitively says that an attacker, having seen a session from Alice and a session from Bob, should then not be able to distinguish Alice from Bob. This is modelled thanks to phases. In a first phase, the attacker interacts with two passports and two readers. In a second phase, the attacker interacts with either the first passport (and a reader) or the second one and she should not be able to distinguish the two cases.

Passive Authentication. The protocol assumes that the reader and the passport have just run BAC and share an encryption key $ksenc$ and a MAC key $kmac$. In this protocol, the passport sends some biometric data dgp to the reader, authenticated by a certificate (the data have been signed by an authority, with signing key $skds$).

$$R \to P : \mathsf{senc}(\mathsf{read}, ksenc), \mathsf{hash}(\mathsf{senc}(\mathsf{read}, ksenc), kmac)$$
$$P \to R : \mathsf{senc}(datap, ksenc), \mathsf{hash}(\mathsf{senc}(datap, ksenc), kmac)$$

where $datap = dgp, \mathsf{sign}(\mathsf{hash}(dgp), skds)$.

As for the BAC protocol, we analyse unlinkability.

Active Authentication. The active authentication protocol works similarly to the passive authentication protocol but prevents cloning the passport by copying a valid certificate. It assumes that the reader knows the verification key $\mathsf{vk}(sk_P)$ of the passport. The protocol authenticates the reader through a challenge-response mechanism where the protocol must sign a challenge r generated by the reader. The nonce n generated by the passport is used to model a randomised signature scheme.

$$R \to P : \mathsf{senc}((\mathsf{init}, r), ksenc), \mathsf{hash}(\mathsf{senc}((\mathsf{init}, r), ksenc), kmac)$$
$$P \to R : \mathsf{senc}(\mathsf{sign}((n, r), sk_P), ksenc), \mathsf{hash}(\mathsf{senc}(\mathsf{sign}((n, r), sk_P), ksenc), kmac)$$

As for the BAC protocol, we analyse unlinkability.

7.4 Outcome

Since building a dependency graph may be cumbersome and error-prone, we have written a small program to compute the dependency graph of a protocol given its specification and a marking provided by the user. The program also checks that the marking complies with our definition and that type-compliance is satisfied. Note that a marking could actually be derived automatically following our two syntactic criteria but for simplicity, we did not implement this feature. The specifications of all protocols considered here, together with their associated dependency graph, can be found at [21].

The results of our study are displayed in Table 1. For each protocol, we indicate whether the resulting graph is acyclic. The fourth column of the table indicates whether we used the public type criterion whereas the fifth column indicates whether we used the precedence criterion. We note that all the obtained graphs are indeed acyclic with two exceptions: the Yahalom-Lowe and Needham-Schroeder protocols. The Needham-Schroeder protocol admits an attack and thes discovered cycle corresponds to the attack (although insecure protocols may also have an acyclic graph). The reasons of the cyclicity of the graph corresponding to Yahalom-Lowe are more subtle. The security of the

protocol partly relies on the secrecy of a nonce N_b that is first sent encrypted under a long-term key K_{bs} for which we have strong secrecy guarantees and then later sent encrypted under the session key K_{ab}. Our type system cannot exclude that N_b gets revealed at this last step, and is maybe reused in an earlier step, hence creating a cycle.

Our result assumes type-compliance: whenever two encrypted subterms can be unified, they have the same type. For protocols that do not enjoy this property, type-compliance can be retrieved by adding a tag (e.g. a number) in each encryption, which is a good design practice as it avoids message confusion. The second column of the table indicates whether we needed to tag the protocol.

Note that in each case where the resulting graph is acyclic, we can compute a bound on the number of sessions that needs to be analysed, hence reducing decidability of equivalence to the bounded case.

8 Conclusion

We have identified a novel decidable fragment of security protocols for both reachability and trace equivalence. Most of standard protocols used as library of examples for automatic tools fall into our class. However, we have considered only relatively simple protocols. As further work, we should explore whether industrial-scale protocols fall into our class like TLS 1.3 or 5G protocols. This is probably not the case due to the fact that we do not handle else branches. To tackle this issue, we would first need to extend [13] to else branches, that is, showing that whenever there is an attack, there is also a well-typed attack in the presence of else branches.

An interesting feature of our approach is that, for each protocol of our class, we can compute an explicit bound on the number of sessions that need to be considered. This bound is still quite high but thanks to the recent progress of tools like DeepSec [12] or SAT-Equiv [20], we can hope that it will be possible to analyse protocols of our class (hence for an unbounded number of sessions) with automatic tools, that decide security for a bounded number of sessions. In particular, it was shown in [20] that about a hundred of sessions can now be analysed automatically with SAT-Equiv, for some relatively small protocols of the literature. We believe that we can further refine the computation on the bound on the number of sessions in order to match a bound that tools can reach. To provide a tighter bound, we could for example distinguish when we need a full session or only the first steps of some role, guided by the dependency graph. This would probably require to further refine the dependency graph.

Also, in order to cover a larger class of protocols, one approach is to show that we can soundly remove some additional arrows of the dependency graph. One first step would be to provide additional syntactic criteria for appropriate markings.

Acknowledgements The research leading to these results has received funding from the European Research Council under the European Union's horizon 2020 research and innovation program (ERC grant agreement n° 714955-POPSTAR and ERC grant agreement n° 645865-SPOOC), as well as from the French National Research Agency (ANR) under the project TECAP.

References

1. PKI for machine readable travel documents offering ICC read-only access. Tech. rep., International Civil Aviation Organization (2004)
2. Exigences techniques et administratives applicables au vote électronique. Chancellerie fédérale ChF (2014). Swiss recommendation on e-voting
3. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proc. 28th ACM Symposium on Principles of Programming Languages, POPL '01, pp. 104–115. ACM (2001). DOI 10.1145/360204.360213. URL http://doi.acm.org/10.1145/360204.360213
4. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Inf. Comput. pp. 1–70 (1999)
5. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Hankes Drielsma, P., Héam, P.C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: The AVISPA Tool for the automated validation of internet security protocols and applications. In: K. Etessami, S. Rajamani (eds.) 17th International Conference on Computer Aided Verification, CAV'2005, *Lecture Notes in Computer Science*, vol. 3576, pp. 281–285. Springer, Edinburgh, Scotland (2005)
6. Baudet, M.: Deciding security of protocols against off-line guessing attacks. In: Proc. 12th ACM Conference on Computer and Communications Security (CCS'05). ACM Press (2005)
7. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proc. 14th Computer Security Foundations Workshop (CSFW'01). IEEE Computer Society Press (2001)
8. Blanchet, B.: Vérification automatique de protocoles cryptographiques : modèle formel et modèle calculatoire. automatic verification of security protocols: formal model and computational model. Mémoire d'habilitation à diriger des recherches, Université Paris-Dauphine (2008). En français avec publications en anglais en annexe. In French with publications in English in appendix.
9. Chadha, R., Ciobâcă, S., Kremer, S.: Automated verification of equivalence properties of cryptographic protocols. In: Programming Languages and Systems —Proceedings of the 21th European Symposium on Programming (ESOP'12), *LNCS*, vol. 7211, pp. 108–127. Springer (2012)
10. Chadha, R., Ciobâcă, Ş.., Kremer, S.: Automated verification of equivalence properties of cryptographic protocols. In: Proc. 21th European Symposium on Programming (ESOP'12), LNCS (2012)
11. Cheval, V.: Apte: an algorithm for proving trace equivalence. In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14), *LNCS*, vol. 8413, pp. 587–592 (2014)
12. Cheval, V., Kremer, S., Rakotonirina, I.: Deepsec: Deciding equivalence properties in security protocols - theory and practice. In: Proc. 39th IEEE Symposium on Security and Privacy (S&P'18), pp. 525–542. IEEE Computer Society Press, San Francisco, CA, USA (2018)
13. Chrétien, R., Cortier, V., Dallon, A., Delaune, S.: Typing messages for free in security protocols. ACM Transactions on Computational Logic **21**(1) (2019)
14. Chrétien, R., Cortier, V., Delaune, S.: From security protocols to pushdown automata. In: Proc. 40th Int. Colloquium on Automata, Languages and Programming (ICALP'13) (2013)

15. Chrétien, R., Cortier, V., Delaune, S.: Typing messages for free in security protocols: the case of equivalence properties. In: Proc. 25th International Conference on Concurrency Theory (CONCUR'14), *Lecture Notes in Computer Science*, vol. 8704, pp. 372–386. Springer (2014)
16. Chrétien, R., Cortier, V., Delaune, S.: Decidability of trace equivalence for protocols with nonces. In: Proc. 28th IEEE Computer Security Foundations Symposium (CSF'15), pp. 170–184. IEEE Computer Society Press (2015)
17. Comon-Lundh, H., Cortier, V.: New decidability results for fragments of first-order logic and application to cryptographic protocols. In: Proc. 14th International Conference on Rewriting Techniques and Applications (RTA'2003), *LNCS*, vol. 2706. Springer (2003)
18. Cortier, V., Dallon, A., Delaune, S.: Bounding the number of agents, for equivalence too. In: F. Piessens, L. Viganó (eds.) Proceedings of the 5th International Conference on Principles of Security and Trust (POST'16), *Lecture Notes in Computer Science*, vol. 9635, pp. 211–232. Springer, Eindhoven, The Netherlands (2016)
19. Cortier, V., Dallon, A., Delaune, S.: Efficiently deciding equivalence for standard primitives and phases. In: Proc. 23rd European Symposium on Research in Computer Security (ESORICS'18), Lecture Notes in Computer Science. Springer, Barcelona, Spain (2018)
20. Cortier, V., Delaune, S., Dallon, A.: SAT-Equiv: an efficient tool for equivalence properties. In: Proc. 30th IEEE Computer Security Foundations Symposium (CSF'17), pp. 481–494. IEEE Computer Society Press (2017)
21. Cortier, V., Delaune, S., Sundararajan, V.: A decidable class of security protocols for both reachability and equivalence properties. Research report, Loria & Inria Grand Est ; Irisa (2020). URL https://hal.inria.fr/hal-02446170. Supplementary material available at https://hal.inria.fr/hal-02446170/file/protocol-files.zip (protocol specification and dependency graph)
22. Cremers, C.: The Scyther Tool: Verification, falsification, and analysis of security protocols. In: Proc. 20th International Conference on Computer Aided Verification (CAV'08), Lecture Notes in Computer Science, pp. 414–418. Springer (2008)
23. Dawson, J., Tiu, A.: Automating open bisimulation checking for the spi-calculus. In: IEEE Computer Security Foundations Symposium (CSF 2010) (2010)
24. Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Protzenko, J., Rastogi, A., Swamy, N., Béguelin, S.Z., Bhargavan, K., Pan, J., Zinzindohoue, J.K.: Implementing and proving the TLS 1.3 record layer. In: 2017 IEEE Symposium on Security and Privacy (S&P 2017), p. 463–482 (2017)
25. Denning, D.E., Sacco, G.M.: Timestamps in key distribution protocols. Commun. ACM **24**(8), 533–536 (1981). DOI 10.1145/358722.358740. URL https://doi.org/10.1145/358722.358740
26. Dougherty, D.J., Guttman, J.D.: Decidability for lightweight Diffie-Hellman protocols. In: Proc. 27th IEEE Symposium on Computer Security Foundations (CSF'14) (2014)
27. Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Undecidability of bounded security protocols. In: Workshop on Formal Methods and Security Protocols. Trento, Italia (1999)
28. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. Theoretical Computer Science **367**(1–2), 162–202 (2006)
29. Fröschle, S.: Leakiness is decidable for well-founded protocols? In: Proc. 4th Conference on Principles of Security and Trust (POST'15), Lecture Notes in Computer Science. Springer (2015)
30. Hirschi, L., Delaune, S.: Description of some case studies. Deliverable VIP 6.1, (ANR-11-JS02-0006) (2013). 14 pages
31. Lowe, G.: Towards a completeness result for model checking of security protocols. In: Proc. of the 11th Computer Security Foundations Workshop (CSFW'98). IEEE Computer Society Press (1998)
32. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In: Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc., *LNCS*, vol. 8044, pp. 696–701. Springer (2013)

33. Ramanujam, R., Suresh, S.P.: Tagging makes secrecy decidable with unbounded nonces as well. In: Proc. 23rd Conference of Foundations of Software Technology and Theoretical Computer Science (FST&TCS'03), LNCS, pp. 363–374. Springer (2003)
34. Rusinowitch, M., Turuani, M.: Protocol Insecurity with Finite Number of Sessions and Composed Keys is NP-complete. Theoretical Computer Science **299**, 451–475 (2003)