Decidability of trace equivalence for protocols with nonces

Rémy Chrétien, Véronique Cortier, Stéphanie Delaune

Decidability of trace equivalence for protocols with nonces

Rémy Chrétien* †, Véronique Cortier*, and Stéphanie Delaune †
*LORIA, CNRS, France
†LSV, CNRS & ENS Cachan, France

***Abstract*—Privacy properties such as anonymity, unlinkability, or vote secrecy are typically expressed as equivalence properties. In this paper, we provide the first decidability result for trace equivalence of security protocols, for an unbounded number of sessions and unlimited fresh nonces. Our class encompasses most symmetric key protocols of the literature, in their tagged variant.**

I. Introduction

Formal analysis of security protocols has received a lot of attention for the last three decades. Many decision procedures have been proposed with now very mature tools such as ProVerif [1], Avispa [2], or Scyther [3]. Two main families of properties are used to express security goals. Trace-based or reachability properties check that some property holds for any execution trace. They are typically used to state secrecy or authentication. Equivalence properties have been introduced more recently in the context of security protocols. They express that an adversary should not be able to distinguish between two scenarios. They are typically used to state privacy properties such as anonymity, unlinkability, or vote secrecy (*e.g.* [4], [5]).

Given a security protocol, does it achieve its security goals? This question is actually undecidable for trace properties as well as equivalence properties, for an unbounded number of sessions [6]. Bounding the number of sessions suffices to retrieve decidability for standard primitives, both for trace properties [7] and equivalence properties [8]. However, analysing a protocol for a fixed (often low) number of sessions does not allow to *prove* security. Even if my favourite security protocol has no flaw when used three times, there is absolutely no guarantee that a flaw will not appear when used a fourth time. How to prove security without limiting the number of sessions? Some tools such as ProVerif [1] or Scyther [3] can actually handle an unbounded number of sessions although they are not guaranteed to terminate. Yet, in practice, these tools work well, at least for trace 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.

Most existing decidability results for an unbounded number of sessions focused at protocols without nonces (see *e.g.* [6], [9] for trace properties and [10], [11] for equivalence). However, protocols do use nonces in practice. Focusing now at decidability results for protocols with nonces, the results are much fewer. For equivalence properties, there is actually no decidability result. For reachability property, and more specially for secrecy, G. Lowe [12] 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 [13], 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 [14] have recently proposed a decidability result dedicated to Diffie-Hellman protocols. Contemporaneously to our work, Sybille Fröschle [15] has proposed a new decidability result for the "leakiness" property and the class of "well-founded protocols". We provide in Section V-C a detailed comparison with our result. In brief, our result are incomparable since [15] considers a larger class of primitives but a less accurate security property and more restriction on the protocols (*e.g.* ciphertext forwarding is again prohibited and as in [12] a typed model is considered). Note that all these four results are dedicated to secrecy (or a variant of it), a particular trace-based property.

Our contribution. We propose the first decidability result for trace equivalence, for an unbounded number of sessions and with nonces. Since even simple reachability properties are undecidable in this context, we make some assumptions.

Simple processes. This notion has been introduced in [16] and used in subsequent works. Intuitively, we assume that each process communicates on a distinct channel. In practice, each machine has its own IP address and each session is characterised by some session identifier. We also assume that each process consists of a sequence of inputs and outputs (with some tests). This models very well standard security protocols (with no else branches).

Type compliant protocols. Intuitively, we assume that ciphertexts cannot be confused. A similar notion has been formally introduced in [17] and was shown to ensure termination of ProVerif (without nonces). This condition is part of the good design practices and is easy to enforce by adding some identifier (a tag) in each ciphertext. Of course the same tags

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° 258865, project ProSecure, and the ANR project JCJC VIP n^o 11 JS02 006 01.

are re-used in all sessions.

Acyclic dependency graph. Considering constructions used in undecidability results, one can notice that the encodings rely on some form of cyclicity. Typically, the last message of the protocol is re-injected at the first step of the protocol, forming an infinite loop. We therefore introduce the notion of dependency graph, with two notions of dependencies:

- sequential dependency: some action can only be taken after some other actions;
- data dependency: some message can only be built once some information is learnt from another message.

This graph can be computed (automatically) from the protocol's specification. To detect data dependencies, we actually consider a particular typed instantiation of the protocol. Therefore, the definition of a dependency graph relies itself on the typing system. Moreover, finer typing systems are more likely to yield acyclic dependency graphs.

Our main contribution is to show that the equivalence between simple and type-compliant protocols with an acyclic dependency graph is decidable, for protocols using symmetric encryption, concatenation, and nonces. Our class encompasses most symmetric key protocols we considered, including Needham-Schroeder with symmetric key, Otway-Rees, Denning-Sacco, or Wide-Mouthed-Frog. For some of these protocols, we had to consider an explicitly tagged version.

As mentioned earlier, equivalence properties are used to express privacy properties. They can also be used to encode standard trace properties such as secrecy. For example, secrecy of nonce n in a protocol P can be encoded as strong secrecy:

$$P(a) \approx P(b)$$

where a and b are two public constants used instead of n.

Strong secrecy is generally stronger than secrecy but the two notions may coincide [18]. Similarly, secrecy of a key k may be encoded as a combination of key usability [19] and "which key-concealing" [20]. Intuitively, an attacker should not be able to distinguish between a situation where the key k is used to encrypt some public message a, and a situation where a fresh key is used to encrypt some other public message b. This can be formalised by the following equivalence:

$$P[\mathsf{out}(\{a\}_k)] \approx P[\mathsf{new}\ k'.\mathsf{out}(\{b\}_{k'})]$$

where a, b are two public constants and $P[Q]$ runs Q once the key k is established in P.

In that sense, decidability for equivalence encompasses decidability for secrecy.

Proof technique. We show decidability in two main steps. First, thanks to the type-compliance assumption, we show that we can apply [11], yielding a bound on the size of the messages: if there is a witness of non-equivalence then there is a well-typed witness, and this induces a strict format for the messages occurring in such a witness. Note that the number of distinct messages remains unbounded due to nonces.

The second step of the proof relies on the dependency graph. We show that any well-typed execution trace complies with the execution order induced by the dependency graph, which allows to split well-typed traces into small independent traces, which in turn yields decidability.

Scope. The scope of our result depends on how often protocols induce an acyclic dependency graph. For the sake of clarity, we first provide a generic definition of a *dependency graph* (Definition 9). However, some interesting protocols such as the Needham-Schroeder symmetric key protocol are cyclic with this definition. In a second step, we provide a criterion that safely allows to remove edges in the dependency graph, yielding acyclicity for most of the protocols we considered. This more flexible notion of dependency graph is called *refined dependency graph* (Definition 13). We believe that our approach provides a good level of flexibility. In case some protocols were found to be cyclic with our current definition of a dependency graph, it should be possible to develop other criteria that soundly remove edges.

II. Model for security protocols

Security protocols are naturally modelled through a process algebra like the applied pi-calculus [21]. We adapt here the process algebra provided in [11] since we wish to use the corresponding simplification result to well-typed traces.

A. Syntax

Term algebra: As usual, messages are modelled by terms. We consider an infinite set of *names* $\mathcal{N}$ and two distinct sets of *variables* $\mathcal{X}$ and $\mathcal{W}$. Names are typically used to represent keys or nonces. Variables in $\mathcal{X}$ typically refer to unknown parts of messages expected by participants while variables in $\mathcal{W}$ are used to store messages learnt by the attacker. We consider the following sets of function symbols:

$$\Sigma_{\mathsf{c}} = \{\mathsf{enc}, \langle\,\rangle\},\ \Sigma_{\mathsf{d}} = \{\mathsf{dec}, \mathsf{proj}_1, \mathsf{proj}_2\}, \text{ and } \Sigma = \Sigma_{\mathsf{c}} \cup \Sigma_{\mathsf{d}}.$$

The symbol enc of arity 2 represents encryption while dec is the corresponding decryption symbol. Concatenation of messages is modelled through the symbol $\langle\ \rangle$ of arity 2, with corresponding projection functions proj_1 and proj_2 of arity 1. We distinguish between *constructor* symbols in Σ_{c} and *destructor* symbols in Σ_{d}.

We consider several sets of terms. Given a set of A of atoms (*i.e.* names, variables, and constants), and a signature $\mathcal{F} \in \{\Sigma_{\mathsf{c}}, \Sigma_{\mathsf{d}}, \Sigma\}$, we denote by $\mathcal{T}(\mathcal{F}, \mathsf{A})$ the set of terms built from $\mathcal{F}$ and A. Constructors terms with atomic encryptions are represented by the set $\mathcal{T}_0(\Sigma_{\mathsf{c}}, \mathsf{A})$, which is the subset of $\mathcal{T}(\Sigma_{\mathsf{c}}, \mathsf{A})$ such that any subterm $\mathsf{enc}(m, k)$ of a term in $\mathcal{T}_0(\Sigma_{\mathsf{c}}, \mathsf{A})$ is such that $k \in \mathsf{A}$. Then $\mathcal{T}_0(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N})$ is the set of *messages*. The *positions* of a term are defined as usual.

An attacker can build any term by applying function symbols. His computation is formally modelled by a term $R \in \mathcal{T}(\Sigma, \Sigma_0 \cup \mathcal{W})$, called *recipe*. Note that a recipe does not contain names.

We denote $vars(u)$ the set of variables that occur in u. The application of a substitution σ to a term u is written $u\sigma$, and we denote $dom(\sigma)$ its *domain*. Two terms u_1 and u_2 are *unifiable* when there exists σ such that $u_1\sigma = u_2\sigma$.

The decryption of an encryption with the right key yields the plaintext. Similarly, the left (or right) projection of a concatenation yields the left (or right) component. These properties are reflected in the three following convergent rewrite rules:

$$\begin{array}{rcll} \mathsf{dec}(\mathsf{enc}(x,y),y) & \to & x & \\ \mathsf{proj}_i(\langle x_1,x_2\rangle) & \to & x_i & \text{with } i\in\{1,2\}. \end{array}$$

The *normal form* of a term $t \in \mathcal{T}(\Sigma, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$ is denoted by $u\!\downarrow$. The reader is referred to [22] for the exact definitions of rewrite rules, convergence, and normal forms.

Example 1: Let $t = \mathsf{enc}(n, k)$ where $n, k \in \mathcal{N}$. This term represents the encryption of n by k. The term $\mathsf{dec}(t, k)$ models the application of the decryption algorithm on t using the key k. Then $\mathsf{dec}(t, k)\!\downarrow = n$.

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 name input and output actions of processes. Protocols are modelled through processes built by the following grammar:

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

where $u \in \mathcal{T}(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$, $n \in \mathcal{N}$, $c, c' \in \mathcal{C}h$, and $\alpha \in \mathcal{L}$.

The process 0 denotes the null process that does nothing. $\alpha : \mathsf{in}(c, u).P$ inputs a message m of the form u on channel c and then behaves like P. Similarly, $\alpha : \mathsf{out}(c, u).P$ outputs u on channel c and behaves like P. The process $(P \mid Q)$ denotes the parallel composition of P and Q while $!P$ denotes an arbitrary number of processes P in parallel. $\mathsf{new}\ n.P$ generates a fresh nonce (or key) n and behaves like P. The process $\mathsf{new}\ c'.\mathsf{out}(c, c').P$ is a special construction for creating new channels: any new channel should be made public immediately. Intuitively, we consider here only public channels. These fresh channel names are used to identify a process, similarly to a session identifier for example.

We may omit the null process for simplicity. We assume that names and variables are bound at most once. For example, in the process $\mathsf{in}(c, x).\mathsf{in}(c, x)$ the variable x is bound once, whereas in $\mathsf{in}(c, x) \mid \mathsf{in}(c, x)$, one occurrence of the variable x should be renamed. The set of *free variables* that occur in P, *i.e.* variables that are not in the scope of an input, is denoted $fv(P)$. The set of labels that occur in P is denoted $\mathcal{L}(P)$.

Definition 1: A *protocol* P is a process such that P is ground (*i.e.* $fv(P) = \emptyset$), P does not use channel names from $\mathcal{C}h^{\mathsf{fresh}}$, and labels occurring in P are distinct.

Example 2: The Denning Sacco protocol [23] (without timestamps) is a key distribution protocol using symmetric encryption and a trusted server. It can be described informally as follows:

1. $A \to S:\ A, B$
2. $S \to A:\ \{B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3. $A \to B:\ \{K_{ab}, A\}_{K_{bs}}$

where $\{m\}_k$ denotes the symmetric encryption of a message m with key k. The agents A and B aim at authenticating each other and establishing a session key K_{ab} through a trusted server S. The key K_{as} (resp. K_{bs}) is a long term key shared between A and S (resp. B and S).

We model the Denning Sacco protocol in our formalism. Below, k_{as}, k_{bs}, k_{ab} are names, whereas a and b are constants from Σ_0. We denote by $\langle x_1, \ldots, x_{n-1}, x_n\rangle$ the term $\langle x_1, \langle \ldots \langle x_{n-1}, x_n\rangle\rangle\rangle$. The protocol is modelled by the parallel composition of three processes P_A, P_B, and P_S, corresponding to the roles of A, B, and S.

$$\begin{array}{rl} P_{\mathsf{DS}} = & !\,\mathsf{new}\ c_1.\mathsf{out}(c_A, c_1).P_A \ \mid !\,\mathsf{new}\ c_2.\mathsf{out}(c_B, c_2).P_B \\ & \mid !\,\mathsf{new}\ c_3.\mathsf{out}(c_S, c_3).P_S \end{array}$$

The processes P_A, P_B, and P_S are given below.

$$\begin{array}{rcl} P_A & = & \alpha_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{b}\rangle). \\ & & \alpha_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{b}, x_{AB}, x_B\rangle, k_{as})). \\ & & \alpha_3 : \mathsf{out}(c_1, x_B) \\ P_B & = & \beta_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y_{AB}, \mathsf{a}\rangle, k_{bs})) \\ P_S & = & \gamma_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{b}\rangle).\ \mathsf{new}\ k_{ab}. \\ & & \gamma_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a}\rangle, k_{bs})\rangle, k_{as})) \end{array}$$

B. Semantics

A configuration is a process together with the current knowledge of the attacker. Formally, a *configuration* is a pair $(\mathcal{P}; \phi)$ where:

- $\mathcal{P}$ is a multiset of ground processes;
- $\phi = \{\mathsf{w}_1 \triangleright m_1, \ldots, \mathsf{w}_n \triangleright m_n\}$ 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, *i.e.* terms in $\mathcal{T}_0(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N})$.

We may write P instead of $(\{P\}; \emptyset)$, and $P \cup \mathcal{P}$ or $P \mid \mathcal{P}$ instead of $\{P\} \cup \mathcal{P}$.

The frame ϕ represents the messages the attacker has learnt so far. He may deduce new messages from his knowledge. This is formalised through the deducibility notion.

Definition 2: A message u is *deducible* from a frame ϕ, denoted $\phi \vdash u$, if there exists a recipe R such that $R\phi\!\downarrow = u$.

The operational semantics of a process is induced by the relation $\xrightarrow{\alpha}$ over configurations defined in Figure 1. The first rule corresponds to the case where an agent expects to receive a message of the form u. The adversary may send any message $R\phi\!\downarrow$ he can build, provided it matches u, that is $u\sigma = R\phi\!\downarrow$ for some σ. Then the process proceeds, instantiated by σ. The second rule reflects the case of an emission of a message, which is stored in ϕ using some new variable w_{i+1}. Note that the term is emitted only if it is a message, otherwise there is no output. When a new channel is generated (third rule), we simply use a fresh channel name. The channel is not added to the attacker's knowledge but the attacker is able

$$(\alpha : \mathsf{in}(c,u).P \cup \mathcal{P}; \phi) \xrightarrow{\mathsf{in}(c,R)} (P\sigma \cup \mathcal{P}; \phi) \quad \text{where } R \text{ is a recipe such that } R\phi\downarrow \text{ is a message and } R\phi\downarrow = u\sigma \text{ for some } \sigma \text{ with } dom(\sigma) = vars(u)$$

$$(\alpha : \mathsf{out}(c,u).P \cup \mathcal{P}; \phi) \xrightarrow{\mathsf{out}(c,\mathsf{w}_{i+1})} (P \cup \mathcal{P}; \phi \cup \{\mathsf{w}_{i+1} \triangleright u\}) \quad \text{where } u \text{ is a message and } i \text{ is the number of elements in } \phi$$

$$(\mathsf{new}\, c'.\mathsf{out}(c,c').P \cup \mathcal{P}; \phi) \xrightarrow{\mathsf{out}(c,ch_i)} (P\{^{ch_i}/_{c'}\} \cup \mathcal{P}; \phi) \quad \text{where } ch_i \text{ is the "next" fresh channel name available in } \mathcal{C}h^{\mathsf{fresh}}$$

$$(\mathsf{new}\, n.P \cup \mathcal{P}; \phi) \xrightarrow{\tau} (P\{^{n'}/_{n}\} \cup \mathcal{P}; \phi) \quad \text{where } n' \text{ is a fresh name in } \mathcal{N}$$

$$(!P \cup \mathcal{P}; \phi) \xrightarrow{\tau} (P \cup !P \cup \mathcal{P}; \phi)$$

Fig. 1. Semantics of the processes

to observe on which channels the processes communicate. Finally, the two last rules are quite standard and correspond respectively to nonce generation and replication. They are not observable (τ action) by an attacker. The relation $\xrightarrow{\alpha_1 \dots \alpha_n}$ between configurations (where $\alpha_1 \dots \alpha_n$ is a sequence of actions) is defined as the transitive closure of $\xrightarrow{\alpha}$.

Given a sequence of observable actions tr, we write $K \overset{\mathsf{tr}}{\Longrightarrow} K'$ when there exists a sequence $\alpha_1 \dots \alpha_n$ such that $K \xrightarrow{\alpha_1 \dots \alpha_n} K'$ and tr is obtained from $\alpha_1 \dots \alpha_n$ by erasing all occurrences of τ. For every protocol P, we define its *set of traces* as follows:

$$\mathsf{trace}(P) = \{(\mathsf{tr}, \phi) \mid P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi) \text{ for some } (\mathcal{P}; \phi)\}.$$

Note that, by definition of $\mathsf{trace}(P)$, $\mathsf{tr}\phi\downarrow$ only contains terms from $\mathcal{T}_0(\Sigma_\mathsf{c}, \Sigma_0 \cup \mathcal{N})$.

Example 3: Consider the following sequence tr:

$$\begin{aligned}\mathsf{tr} = \quad & \mathsf{out}(c_A, ch_1).\mathsf{out}(c_B, ch_2).\mathsf{out}(c_S, ch_3).\\ & \mathsf{out}(ch_1, \mathsf{w}_1).\mathsf{in}(ch_3, \mathsf{w}_1).\mathsf{out}(ch_3, \mathsf{w}_2).\\ & \mathsf{in}(ch_1, \mathsf{w}_2).\mathsf{out}(ch_1, \mathsf{w}_3).\mathsf{in}(ch_2, \mathsf{w}_3)\end{aligned}$$

This sequence tr allows one to reach the frame:

$$\begin{aligned}\phi = \quad & \{\mathsf{w}_1 \triangleright \langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{w}_2 \triangleright \mathsf{enc}(\langle \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a}\rangle, k_{bs})\rangle, k_{as}),\\ & \mathsf{w}_3 \triangleright \mathsf{enc}(\langle k_{ab}, \mathsf{a}\rangle, k_{bs})\}.\end{aligned}$$

We have that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P_{\mathsf{DS}})$. This trace corresponds to a normal execution of the protocol.

C. Trace equivalence

Privacy properties are typically expressed through behavioural equivalences. For example, for anonymity, an attacker should not distinguish between a session from Alice and a session from Bob. Equivalence-based properties are not limited to privacy but may also be used to state strong secrecy or other game-based properties [16]. There exist several variants of equivalences: *e.g.* observational equivalence, may-testing, or trace equivalence. We study trace equivalence and we refer the reader to [24] for a comparison of these notions.

We first start with *static equivalence* which formalises when an attacker cannot distinguish between two sequences of messages.

Definition 3: Two frames ϕ_1 and ϕ_2 are *statically equivalent*, $\phi_1 \sim \phi_2$, if $dom(\phi_1) = dom(\phi_2)$ and:

- for any recipe R, $R\phi_1\downarrow \in \mathcal{T}_0(\Sigma_\mathsf{c}, \Sigma_0 \cup \mathcal{N})$ if, and only if, $R\phi_2\downarrow \in \mathcal{T}_0(\Sigma_\mathsf{c}, \Sigma_0 \cup \mathcal{N})$; and
- for all recipes R_1 and R_2 such that $R_1\phi_1\downarrow, R_2\phi_1\downarrow \in \mathcal{T}_0(\Sigma_\mathsf{c}, \Sigma_0 \cup \mathcal{N})$, we have that $R_1\phi_1\downarrow = R_2\phi_1\downarrow$ if, and only if, $R_1\phi_2\downarrow = R_2\phi_2\downarrow$.

Intuitively, an attacker can see the difference between two frames ϕ_1 and ϕ_2 if, either some computation R fails in ϕ_1 (that is $R\phi_1\downarrow$ is *not* a message) while it does not fail in ϕ_2; or the attacker can build an equality $R_1 = R_2$ that holds in ϕ_1 and not in ϕ_2 (or conversely).

Example 4: Consider $\phi_1 = \phi \cup \{\mathsf{w}_4 \triangleright \mathsf{enc}(\mathsf{m}_1, k_{ab})\}$ and $\phi_2 = \phi \cup \{\mathsf{w}_4 \triangleright \mathsf{enc}(\mathsf{m}_2, k)\}$ where ϕ has been introduced in Example 3. The terms m_1, m_2 are public constants in Σ_0, and k is a name in $\mathcal{N}$. We have that the two frames ϕ_1 and ϕ_2 are statically equivalent. Intuitively, at the end of a normal execution between honest participants, an attacker can not distinguish whether the key used to encrypt a message (here the constants m_1 and m_2) is the session key that has been established or a fresh name k.

In contrast, $\phi_1' = \phi_1 \cup \{\mathsf{w}_5 \triangleright k_{ab}\}$ and $\phi_2' = \phi_2 \cup \{\mathsf{w}_5 \triangleright k_{ab}\}$ are *not* in static equivalence. Indeed, an attacker can observe that the fourth message of ϕ_1 can be decrypted by the fifth message, which is not the case in ϕ_2'. Formally, consider the recipe $R = \mathsf{dec}(\mathsf{w}_4, \mathsf{w}_5)$. Then $R\phi_1'\downarrow \in \mathcal{T}_0(\Sigma_\mathsf{c}, \Sigma_0 \cup \mathcal{N})$ while $R\phi_2'\downarrow \notin \mathcal{T}_0(\Sigma_\mathsf{c}, \Sigma_0 \cup \mathcal{N})$

Trace equivalence is the active counterpart of static equivalence: an attacker is not be able to distinguish between two processes if, however he interacts with them, the resulting sequences of sent messages are in static equivalence.

Definition 4: A protocol P is *trace included* in a protocol Q, written $P \sqsubseteq Q$, if for every $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$, there exists $(\mathsf{tr}', \phi') \in \mathsf{trace}(Q)$ such that $\mathsf{tr} = \mathsf{tr}'$ and $\phi \sim \phi'$. The protocols P and Q are *trace equivalent*, written $P \approx Q$, if $P \sqsubseteq Q$ and $Q \sqsubseteq P$.

Example 5: The process P_{DS} presented in Example 2 models the Denning Sacco protocol. Assume now that we wish to check strong secrecy of the exchanged key, as received by the agent B. As discussed in Introduction, this can be expressed by checking whether $P^1_{\mathsf{DS}} \approx P^2_{\mathsf{DS}}$ where:

- P^1_{DS} is as P_{DS} but we add "$\beta_2 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_1, y_{AB}))$" at the end of the process P_B;

- P^2_{DS} is as the protocol P_{DS} but we add the instruction "$\mathsf{new}\, k.\beta_2 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_2, k))$" at the end of P_B,

The terms m_1 and m_2 are two public constants from Σ_0.

While the key received by B cannot be learnt by an attacker, strong secrecy of this key is not guaranteed. Indeed, due to the lack of freshness, the same key can be sent several times to B, and this can be observed by an attacker. Formally, the attack is as follows. Consider the sequence

$$\mathsf{tr}' = \mathsf{tr}.\mathsf{out}(ch_2, \mathsf{w}_4).\\ \mathsf{out}(c_B, ch_4).\mathsf{in}(ch_4, \mathsf{w}_3).\mathsf{out}(ch_4, \mathsf{w}_5)$$

where tr has been defined in Example 3. The attacker simply replays an old session. The resulting (unique) frames are

- $\phi'_1 = \phi \cup \{\mathsf{w}_4 \triangleright \mathsf{enc}(\mathsf{m}_1, k_{ab}),\ \mathsf{w}_5 \triangleright \mathsf{enc}(\mathsf{m}_1, k_{ab})\}$; and
- $\phi'_2 = \phi \cup \{\mathsf{w}_4 \triangleright \mathsf{enc}(\mathsf{m}_2, k),\ \mathsf{w}_5 \triangleright \mathsf{enc}(\mathsf{m}_2, k')\}$.

Then $(\mathsf{tr}', \phi'_1) \in \mathsf{trace}(P^1_{\mathsf{DS}})$ and $(\mathsf{tr}', \phi'_2) \in \mathsf{trace}(P^2_{\mathsf{DS}})$. However, we have that $\phi'_1 \not\sim \phi'_2$ since $\mathsf{w}_4 = \mathsf{w}_5$ in ϕ'_1 but not in ϕ'_2. Thus P^1_{DS} and P^2_{DS} are *not* in trace equivalence. To avoid this attack, the messages of the Denning-Sacco protocol shall include timestamps.

D. Simple processes

Consider two processes which are both willing to emit some message m on some public channel c: let $A = B = \mathsf{out}(c, m)$. Then consider the two processes in parallel:

$$A \mid B = \mathsf{out}(c, m) \mid \mathsf{out}(c, m)$$

This current practice when modelling protocols actually weakens the attacker power since in that case, he has no information on whether the message is coming from A or from B. Similar issues appear when processes are expecting a message on some public channel. In practice, it is often the case that an attacker may identify processes through *e.g* IP addresses and even sessions using sessions identifiers.

We therefore introduce the class of *simple processes*, similar to the one introduced in [16].

Definition 5: A *simple protocol* P is a protocol 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:

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

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

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

III. A first decidability result

Trace equivalence is undecidable in general for an unbounded number of sessions, inheriting undecidability from the standard secrecy case (see *e.g.* [6]). We present here our two main assumptions for obtaining decidability: *type-compliance* and *acyclic dependency graph*. For the clarity of the presentation, we present in this section a rather coarse definition of dependency graph. In the next section, we provide sound criteria to remove some of its edges.

A. Typing system

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

Definition 6: A *structure-preserving typing system* is a pair $(\mathcal{T}_0, \delta_0)$ where $\mathcal{T}_0$ is a set of elements called *atomic types*, and δ_0 is a function mapping atomic terms in $\Sigma_0 \cup \mathcal{N} \cup \mathcal{X}$ to types τ generated using the following grammar:

$$\begin{array}{rcll} \tau, \tau_1, \tau_2 & = & \tau_0 & \text{with } \tau_0 \in \mathcal{T}_0 \\ & & \mid \langle \tau_1, \tau_2 \rangle & \\ & & \mid \mathsf{enc}(\tau_1, \tau_2) & \end{array}$$

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:

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

Example 6: Going back to our running example, we consider the structure-preserving typing system generated from the set $\mathcal{T}_{\mathsf{DS}} = \{\tau_{\mathsf{a}}, \tau_{\mathsf{b}}, \tau_{\mathsf{m}}, \tau_{kas}, \tau_{kbs}, \tau_{kab}\}$ of atomic types and the function δ_{DS} that associates the expected type to each constant/name, and the following type to variables:

- $\delta_{\mathsf{DS}}(x_{AB}) = \delta_{\mathsf{DS}}(y_{AB}) = \tau_{kab}$; and
- $\delta_{\mathsf{DS}}(x_B) = \mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kbs})$.

B. Type-compliance

Our first main hypothesis on the typing of protocols is that any two unifiable subterms have the same type. The goal of this part is to state this hypothesis formally.

Due to the presence of replication, we need to consider two copies of protocols in order to consider different instances of the variables. Given a protocol P with replication, we define its 2-unfolding $\mathsf{unfold}^2(P)$ to be the protocol such that every occurrence of a process $!R$ in P is replaced by $R \mid R$, and some α-renaming (preserving type) is performed on one copy to avoid variables or names capture. Note that if P is a protocol that does not contain any replication, we have that $\mathsf{unfold}^2(P) = P$.

We write $St(t)$ for the set of *(syntactic) subterms* of a term t, and $ESt(t)$ the set of its *encrypted subterms*, *i.e.*

$$ESt(t) = \{u \in St(t) \mid u \text{ is of the form } \mathsf{enc}(u_1, u_2)\}.$$

We extend this notion to sets/sequences of terms, and to protocols as expected.

A protocol is type-compliant if two unifiable subterms have the same type. Formally, we use the definition of [11], which is similar to the one introduced in [17].

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

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

Example 7: The protocol P^1_{DS} (resp. P^2_{DS}) is type-compliant w.r.t. the typing system given in Example 6. Indeed, the encrypted subterms of $\mathsf{unfold}^2(P^1_{\mathsf{DS}})$ are:

1) $t_A = \mathsf{enc}(\langle \mathsf{b}, x_{AB}, x_B \rangle, k_{as})$;
2) $t_{B1} = \mathsf{enc}(\langle y_{AB}, \mathsf{a} \rangle, k_{bs})$;
3) $t_{B2} = \mathsf{enc}(\mathsf{m}_1, y_{AB})$;
4) $t_{S1} = \mathsf{enc}(\langle \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a} \rangle, k_{bs}) \rangle, k_{as})$; and
5) $t_{S2} = \mathsf{enc}(\langle k_{ab}, \mathsf{a} \rangle, k_{bs})$

as well as the renaming of these terms obtained by replacing k_{ab}, x_{AB}, y_{AB}, and x_B with fresh names/variables of the same type.

It is easy to check that the type-compliance condition is satisfied for any pair of terms. For instance, we have that t_A and t_{S1} are unifiable, and they have indeed the same type:

$$\begin{aligned} \delta_{\mathsf{DS}}(t_A) &= \mathsf{enc}(\langle \tau_{\mathsf{b}}, \tau_{kab}, \mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kbs}) \rangle, \tau_{kas}) \\ &= \delta_{\mathsf{DS}}(t_{S1}). \end{aligned}$$

As shown in the following example, not all protocols are type-compliant w.r.t. a structure-preserving typing system.

Example 8: For instance, consider the following protocol:

$$\begin{aligned} P \;=\; !\,\mathsf{new}\, c.\mathsf{out}(c, c').\ &\alpha_1 : \mathsf{in}(c', \mathsf{enc}(x, k)).\\ &\alpha_2 : \mathsf{out}(c', \mathsf{enc}(\langle x, x \rangle, k). \end{aligned}$$

We have that $t_1 = \mathsf{enc}(x, k)$ and $t_2 = \mathsf{enc}(\langle x, x \rangle, k)$ are both in $ESt(\mathsf{unfold}^2(P))$ as well as the terms t'_1 and t'_2 obtained from t_1 and t_2 by simply renaming x with another variable, say x', having the same type as x. The two terms t_1 and t'_2 are unifiable, and thus should receive the same type, but this would imply that $\delta(x) = \langle \delta(x'), \delta(x') \rangle$, and thus x can not receive the same type as x'.

C. Dependency graph

We consider a protocol P which is type-compliant w.r.t. a structure-preserving typing system $(\mathcal{T}_P, \delta_P)$. To define our dependency graph, we first define public and honest types. Terms of public type are always deducible for an adversary, whereas terms of honest type will be guaranteed to be secret (except those built using public constants only).

A type τ_p is *public* if $\delta_P(n) \notin St(\tau_p)$ for any name n occurring in P. Intuitively, in a well-typed execution, a term having a public type is a term built using public constant only, and is thus deducible from the beginning of any execution.

An atomic type τ_h is *honest* if *(i)* τ_h does not appear in plaintext position in $u\delta_P$ for any term u occurring in P; *(ii)* $\tau_h \neq \delta_P(a)$ for any constant/variable a occurring in P. Intuitively, this ensures that, in a well-typed execution, terms of type τ_h will never occur in plaintext position, and no public constant of this type will be used in key position.

Example 9: Going back to our running example, we have that τ_{a}, τ_{b}, τ_{m} are public types while τ_{kas} and τ_{kbs} are honest types. In contrast, τ_{kab} is neither a public nor an honest type. Indeed, $\tau_{kab} = \delta_P(y_{AB})$ and the variable y_{AB} occurs in P.

We define inductively a function ρ_{io} that inspects a type τ and returns its set of deducible subterms (where τ is viewed as a term) together with the set of keys needed to access each subterm. For this, we introduce a new syntactic symbol $\#$.

Definition 8: Given a type τ, a position p and a set S of types, the function ρ_{io} is inductively defined as follows:

- $\rho_{\mathsf{io}}(\tau_0, p, S) = \{(\tau_0, p)\#S\}$ for any atomic type τ_0;
- $\rho_{\mathsf{io}}(\langle \tau_1, \tau_2 \rangle, p, S) = \rho_{\mathsf{io}}(\tau_1, p.1, S) \cup \rho_{\mathsf{io}}(\tau_2, p.2, S)$;
- $\rho_{\mathsf{io}}(\mathsf{enc}(\tau_1, \tau_2), p, S) =$
  $\begin{cases} \{(\mathsf{enc}(\tau_1, \tau_2), p)\#S\} \text{ if } \tau_2 \text{ is an honest type;} \\ \{(\mathsf{enc}(\tau_1, \tau_2), p)\#S\} \cup \rho_{\mathsf{io}}(\tau_1, p.1, S \cup \{\tau_2\}) \text{ otherwise.} \end{cases}$

Given a type τ, the function $\rho_{\mathsf{io}}(\tau, \epsilon, \emptyset)$ computes a set of elements of the form $(\tau_i, p_i)\#S_i$. Intuitively, it means that the term of type τ_i at position p_i in τ is accessible from the term τ after some decryptions using keys occurring in the set S_i.

We also define two functions ρ_{out} and ρ_{in} that help us to define the flows that may happen during a protocol execution.

$$\rho_{\mathsf{out}}(\tau') = \{(\tau, p) \mid (\tau, p)\#S \in \rho_{\mathsf{io}}(\tau', \epsilon, \emptyset)\}; \text{ and}$$
$$\rho_{\mathsf{in}}(\tau') = \{\tau, \tau_1, \ldots, \tau_k \mid (\tau, p)\#\{\tau_1, \ldots, \tau_k\} \in \rho_{\mathsf{io}}(\tau', \epsilon, \emptyset)\}.$$

Intuitively, $\rho_{\mathsf{out}}(\tau')$ returns the types of the terms that may be deducible by the attacker once a term of type τ' is outputted, whereas $\rho_{\mathsf{in}}(\tau')$ returns all the types that may be used by the attacker to fill an input of type τ'. In case of an output, we also return the position at which the type occurred. This information will be added in our dependency graph, and used in Section IV to present our refined dependency graph.

Example 10: Continuing our running example, we have that:

$\rho_{\mathsf{out}}(\langle \tau_{\mathsf{a}}, \tau_{\mathsf{b}} \rangle) = \{(\tau_{\mathsf{a}}, 1), (\tau_{\mathsf{b}}, 2)\}$ and $\rho_{\mathsf{in}}(\langle \tau_{\mathsf{a}}, \tau_{\mathsf{b}} \rangle) = \{\tau_{\mathsf{a}}, \tau_{\mathsf{b}}\}$;
$\rho_{\mathsf{out}}(\mathsf{enc}(\langle y_{AB}\delta_P, \tau_{\mathsf{a}} \rangle, \tau_{kbs})) = \{(\mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kbs}), \epsilon)\}$;
$\rho_{\mathsf{in}}(\mathsf{enc}(\langle y_{AB}\delta_P, \tau_{\mathsf{a}} \rangle, \tau_{kbs})) = \{\mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kbs})\}$.
since τ_{kbs} is an honest type.

We are now ready to define the dependency graph. It captures two main sources of dependencies: sequential dependencies, when an action may only occur after another one, and data dependencies, when the production of a term depends on other sent terms.

Definition 9: The *dependency graph* associated to a type-compliant, simple protocol P (w.r.t. a structure-preserving typing system $(\mathcal{T}_P, \delta_P)$) is a graph having $\mathcal{L}(P)$ as vertices and that are connected as follows:

1) for every action with label α in P that directly follows an action with label β in P, there is an edge $\alpha \to \beta$;

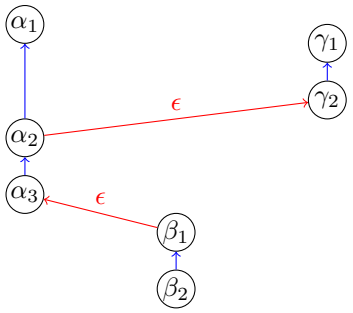

Fig. 2. Dependency graph for P^1_{DS} w.r.t. $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$

2) for every "$\alpha : \mathsf{in}(c, u)$" and "$\beta : \mathsf{out}(d, v)$" in P, there is an edge $\alpha \rightarrow^p \beta$ if there exists $\tau \in \rho_{\mathsf{in}}(u\delta_P)$ such that
$$(\tau, p) \in \rho_{\mathsf{out}}(v\delta_P)$$
unless τ is a public type.
3) for every "$\alpha : \mathsf{out}(c, u)$" and "$\beta : \mathsf{out}(d, v)$" in P, there is an edge $\alpha \rightarrow^p \beta$ if $(\tau, q)\#(S \cup \{\tau_k\}) \in \rho_{\mathsf{io}}(u\delta_P, \epsilon, \emptyset)$ for some τ, q, S, and τ_k such that
$$(\tau_k, p) \in \rho_{\mathsf{out}}(v\delta_P)$$
unless τ_k is a public type.

Intuitively, if there exists an edge from α to β, it implies that the action α *may* depend on action β. More precisely, item 1 captures sequential dependencies, whereas items 2 and 3 are about data dependencies. Item 2 captures dependencies that occur due to the fact that the attacker need to produce a term to comply with the given input. For this, all the needed pieces may come from different outputs (but at a plaintext position). Now, item 3 is needed because, when such a piece occurs at a plaintext position (for instance under an encryption with k), it may be important for the attacker to learn the key k, and this generates new dependencies.

Example 11: The dependency graph for the protocol P^1_{DS} defined in Example 5 w.r.t. the typing system $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$ given in Example 6 is depicted in Figure 2. The vertical arrows correspond to sequential dependencies (item 1) whereas all the other arrows are actually due to item 2. In this example, item 3 does not produce any arrow.

Intuitively, these arrows mean that the input α_2 (resp. β_1) may depend on the output γ_2 (resp. α_3). In other words the outputted term may be (partially) used to fill the input. The relevant parts of the output are indicated by the position p on top of the arrows.

Note that for P^2_{DS} (also defined in Example 5), we can introduce an additional atomic type τ_k for name k (or reuse the atomic type τ_{kab}. In both cases, the dependency graph of P^2_{DS} will be exactly the same as the one obtained for P^1_{DS}.

Example 12: Let P be the protocol

$$P = \alpha : \mathsf{in}(c_1, k_1) | \beta : \mathsf{out}(c_2, \mathsf{enc}(k_1, k_2)) | \gamma : \mathsf{out}(c_3, k_2)$$

Consider the typing system $(\mathcal{T}, \delta)$ where $\mathcal{T} = \{\tau_1, \tau_2\}$; $\delta(k_1) = \tau_1$ and $\delta(k_2) = \tau_2$. Its typing graph is shown in Figure 3. The edge from β to γ is an edge introduced by the third item in Definition 9. Indeed, $\rho_{\mathsf{io}}(\mathsf{enc}(\tau_1, \tau_2), \epsilon, \emptyset) = \{(\mathsf{enc}(\tau_1, \tau_2), \epsilon)\#\emptyset, (\tau_1, 1)\#\{\tau_2\}\}$ and $\rho_{\mathsf{out}}(\tau_2) = \{(\tau_2, \epsilon)\}$.

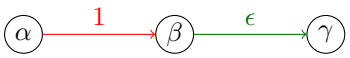

Fig. 3. Dependency graph for P w.r.t. $(\mathcal{T}, \delta)$

D. Our result

Trace equivalence is decidable for simple, type-compliant, acyclic protocols.

Theorem 1: Let P and Q be two simple protocols type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$, and with acyclic dependency graphs. The problem of deciding whether P and Q are in trace equivalence (*i.e.* $P \approx Q$) is decidable.

Example 13: The protocols P^1_{DS} and P^2_{DS} given in Example 5 are simple, type-compliant w.r.t. $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$ and their respective dependency graph is acyclic. Thus this protocol falls into our decidable class.

IV. An improved version of our decidability result

In the previous section we have presented a first decidability result for trace equivalence of simple, type-compliant, acyclic protocols. The Denning-Sacco protocol satisfies these hypotheses. However, some reasonable protocols do not fall in our class. In the next paragraph, we explain why the Needham-Schroeder protocol induces a cyclic dependency graph. However, in that case, the cycle is created by a false dependency. Therefore, in the subsequent paragraphs, we devise criteria to remove some edges of the dependency graph.

A. Motivating example

We consider the well-known Needham Schroeder key establishment protocol [25]. It can be described informally as follows:

1. $A \rightarrow S : \ A, B, N_a$
2. $S \rightarrow A : \ \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3. $A \rightarrow B : \ \{K_{ab}, A\}_{K_{bs}}$
4. $B \rightarrow A : \ \{\mathsf{req}, N_b\}_{K_{ab}}$
5. $A \rightarrow B : \ \{\mathsf{rep}, N_b\}_{K_{ab}}$

We propose a modelling of this protocol in our formalism. Below, k_{as}, k_{bs}, k_{ab}, n_a, n_b, and k are names, whereas a, b, req, rep, m_1 and m_2 are constants from Σ_0.

$$P_{\mathsf{NS}} = \ !\,\mathsf{new}\ c_1.\mathsf{out}(c_A, c_1).P_A \ | \ !\,\mathsf{new}\ c_2.\mathsf{out}(c_B, c_2).P_B \ | \ !\,\mathsf{new}\ c_3.\mathsf{out}(c_S, c_3).P_S$$

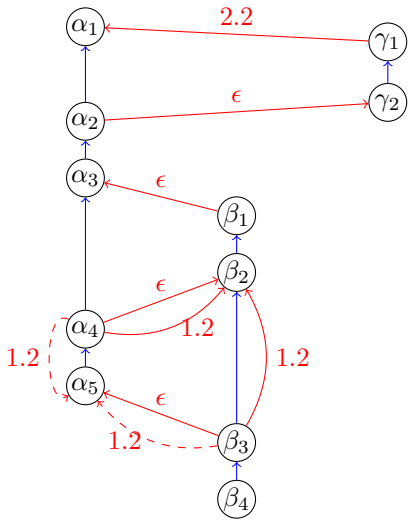

Fig. 4. Dependency graph for P^i_{NS} w.r.t. $(\mathcal{T}_{\mathsf{NS}}, \delta_{\mathsf{NS}})$

where the processes P_A, P_B, and P_S are given below.

$$\begin{aligned} P_A = \mathsf{new}\ n_a.\ & \\ & \alpha_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{b}, n_a \rangle). \\ & \alpha_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle n_a, \mathsf{b}, x_{AB}, x_B \rangle, k_{as})). \\ & \alpha_3 : \mathsf{out}(c_1, x_B). \\ & \alpha_4 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{req}, x_{NB} \rangle, x_{AB})). \\ & \alpha_5 : \mathsf{out}(c_1, \mathsf{enc}(\langle \mathsf{rep}, x_{NB} \rangle, x_{AB})) \end{aligned}$$

$$\begin{aligned} P_B = \beta_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y_{AB}, \mathsf{a} \rangle, k_{bs})).\mathsf{new}\ n_b. & \\ \beta_2 : \mathsf{out}(c_2, \mathsf{enc}(\langle \mathsf{req}, n_b \rangle, y_{AB})). & \\ \beta_3 : \mathsf{in}(c_2, \mathsf{enc}(\langle \mathsf{rep}, n_b \rangle, y_{AB})) & \end{aligned}$$

$$\begin{aligned} P_S = \gamma_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{b}, z_{NA} \rangle).\mathsf{new}\ k_{ab}. & \\ \gamma_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle z_{NA}, \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a} \rangle, k_{bs}) \rangle, k_{as})) & \end{aligned}$$

As in Example 5, we model the security of the exchanged key by requiring that $P^1_{\mathsf{NS}} \approx P^2_{\mathsf{NS}}$ whereP^1_{NS} and P^2_{NS} are defined as follows:

- P^1_{NS} is as the protocol P_{NS} but we add the instruction "$\beta_4 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_1, y_{AB}))$" at the end of P_B;
- P^2_{NS} is as the protocol P_{NS} but we add the instruction "$\mathsf{new}\ k.\beta_4 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_2, k))$" at the end of P_B.

As for the Denning Sacco protocol (see Example 7), type-compliance is satisfied. We only have to introduce some new atomic types: τ_{na}, τ_{nb}, τ_{req}, τ_{rep} and we type each constant (resp. name and variable) as expected. We denote $(\mathcal{T}_{\mathsf{NS}}, \delta_{\mathsf{NS}})$ the resulting structure-preserving typing system.

The resulting dependency graph is depicted in Figure 4 (and the dashed arrow is part of the dependency graph). There is no arrow due to item 3 (dependencies between outputs). As in the Denning Sacco protocol, τ_{kas} and τ_{kbs} are honest type whereas τ_{a}, τ_{b}, τ_{m}, τ_{req}, and τ_{rep} are public types. Due to the fact that τ_{kab} can not be considered as an honest type, many arrows (those that are labelled with 1.2) are added to the dependency graph. This reflects executions that will build ciphertexts with such a key.

The dependency graph is cyclic. Intuitively, this is due to the fact that the subterm at position 1.2 (the nonce N_b) outputted in α_5 may be used in input at α_4. However, if the attacker is able to access this subterm at position 1.2 in α_5, it necessarily means that he already knew this subterm already. Thus, intuitively, this dependency is not necessary. We formalise this notion in the next section. This will lead to a notion of refined dependency graph which is a dependency graph in which some arrows have been removed.

B. Appropriate marking

We first devise a general (semantic) criterion to remove some of the edges of the dependency graph. We proceed by *marking* some of the positions of the graph.

Definition 10: A *marked position* of a protocol P is a pair (α, p) where $\alpha : \mathsf{out}(c, u)$ is an output action occurring in P, and p is a position of the term u. A *marking* of a protocol P is a set of marked positions of P.

This notion of marking is very general. We consider that a marking strategy is *appropriate* for our dependency graph if it indicates subterms that, if deducible, must be deducible earlier in any execution.

Definition 11: A marked position (α, p) of a protocol P is *appropriate* if for any trace $(\mathsf{tr}', \phi') \in \mathsf{trace}(P)$ such that:

- $(\mathsf{tr}', \phi') = (\mathsf{tr}.\mathsf{out}(c, \mathsf{w}), \phi \uplus \{\mathsf{w} \triangleright t\})$; and
- $\mathsf{out}(c, \mathsf{w})$ corresponds to an action labelled by α;

we have that $\phi' \vdash t|_p$ implies $\phi \vdash t|_p$ [1].

A marking $\mathcal{M}$ of P is *appropriate* if all the pairs in $\mathcal{M}$ are appropriate marked positions.

Example 14: We pursue our example started in Section IV-A. We may set $(\alpha_5, 1.2)$ to be a marked position of P^1_{NS} (resp. P^2_{NS}). Intuitively, it is an appropriate marked position since the message $x_{NB}\sigma$ sent by the process P_A cannot be learnt at the step α_5: either $x_{NB}\sigma$ remains secret or it was learnt earlier. We formally show that it is an appropriate marked position in the next section.

Deciding whether a subterm can be marked appropriately is not an easy task. We therefore provide a syntactic (sound) criterion that removes cycles in most cases.

C. A syntactic criterion

We define a function ρ_{io} on terms very similar to the function ρ_{io} defined on types in Section III.

Definition 12: Given a term t, a position p and a set of terms S, the function ρ_{io} is inductively defined as follows:

- $\rho_{\mathsf{io}}(t_0, p, S) = \{(t_0, p)\#S\}$ for any atomic term t_0 (name, constant or variable);
- $\rho_{\mathsf{io}}(\langle t_1, t_2 \rangle, p, S) = \rho_{\mathsf{io}}(t_1, p.1, S) \cup \rho_{\mathsf{io}}(t_2, p.2, S)$;
- $\rho_{\mathsf{io}}(\mathsf{enc}(t_1, t_2), p, S) =$

[1] Note that p is indeed a position of t. Indeed, t must be of the form $t = u\sigma$ where the action labelled by α is $\alpha : \mathsf{out}(c, u)$. Since p is a position of u, it is also a position of $t = u\sigma$.

$$\begin{cases}\{(\mathsf{enc}(t_1,t_2),p)\#S\} \text{ if } \delta_0(t_2) \text{ is an honest type;}\\ \{(\mathsf{enc}(t_1,t_2),p)\#S\}\cup\rho_{\mathsf{io}}(t_1,p.1,S\cup\{t_2\}) \text{ otherwise.}\end{cases}$$

Intuitively, $(u_0,p)\#S \in \rho_{\mathsf{io}}(u,\epsilon,\emptyset)$ if the set S of keys suffices to access the subterm $u_0 = u|_p$.

We show that it is always appropriate to mark a position if the corresponding subterm appears earlier in the protocol, protected by a smaller set of keys.

Lemma 1: Let (α,p) be a marked position in P and assume that there exists an input action $\beta : \mathsf{in}(d,v)$ (or an output action $\beta : \mathsf{out}(d,v)$) in P such that:

1) $\alpha : \mathsf{out}(c,u)$ follows β in P;
2) $(u_0,p)\#S \in \rho_{\mathsf{io}}(u,\epsilon,\emptyset)$ for some u_0, and some S; and
3) $(u_0,q)\#S' \in \rho_{\mathsf{io}}(v,\epsilon,\emptyset)$ for some q, and some $S' \subseteq S$.

Then (α,p) is an appropriate marked position for P.

Example 15: Lemma 1 allows us to formally establish that $(\alpha_5, 1.2)$ is an appropriate marked position for P^1_{NS} (resp. P^2_{NS}). Indeed, we have that:

1) $\alpha_5 : \mathsf{out}(c_1, \mathsf{enc}(\langle \mathsf{rep}, x_{NB}\rangle, x_{AB}))$ follows α_4 in P^1_{NS};
2) $(x_{NB}, 1.2)\#\{x_{AB}\} \in \rho_{\mathsf{io}}(\mathsf{enc}(\langle \mathsf{rep}, x_{NB}\rangle, x_{AB}))$; and
3) $(x_{NB}, 1.2)\#\{x_{AB}\} \in \rho_{\mathsf{io}}(\mathsf{enc}(\langle \mathsf{req}, x_{NB}\rangle, x_{AB}))$.

D. Refined dependency graph

We refine our dependency graph by simply removing any arrow that points towards an appropriate marked position.

Definition 13: Let P be a type-compliant protocol P (w.r.t. a structure-preserving typing system $(\mathcal{T}_P, \delta_P)$) and $\mathcal{M}$ be a marking of P. 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 $\alpha \to^p \beta$ for which

$$(\beta, q) \in \mathcal{M} \text{ and } q \text{ is a prefix of } p.$$

Example 16: The refined dependency graph associated to P^1_{NS} (resp. P^2_{NS}) and $\mathcal{M} = \{(\alpha_5, 1.2)\}$ is the graph depicted in Figure 4, when removing the dashed arrow. This dashed arrow is removed thanks to the (appropriate) marking.

Then trace equivalence is decidable for simple, type-compliant protocols, as soon as their corresponding refined dependency graph is acyclic.

Theorem 2: The problem of deciding whether two simple protocols P and Q, type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$, and with acyclic refined dependency graphs obtained relying on appropriate markings $\mathcal{M}_P$ and $\mathcal{M}_Q$ are trace equivalence (*i.e.* $P \approx Q$) is decidable.

Lemma 1 provides a simple criterion for marking a dependency graph appropriately. We show in the next section that this criterion suffices to cover most cases in practice. It is however possible to devise some other sound criteria.

V. Results

We review several protocols of the literature and identify whether they fall in our decidable class. We first discuss which corruption scenario is considered.

A. Scenario with corruption

The scenario we considered so far for the Denning-Sacco protocol (as well as the Needham-Schroeder protocol) is quite simple. We only consider sessions between two honest agents a and b. Such a scenario is known to be too simplistic and some attacks may be missed, such as the well-known man-in-the-middle attack on the Needham-Schroeder public key protocol [26].

We therefore consider a scenario where honest agents are also willing to engage communications with a dishonest agent c. Let us develop this corruption scenario on the Denning-Sacco protocol. Formally, we consider P^{i+}_{DS} obtained from P^i_{DS} by adding P'_{DS} as well as P''_{DS} in parallel. The purpose of P'_{DS} is to consider that the agent a may be involved in some other sessions with a corrupted agent c, and the server S is ready to answer requests coming from them. Similarly P''_{DS} models the fact that the agent b may be involved in some sessions where the role of A is played by the corrupted agent c. Thus, we consider

$$P'_{\mathsf{DS}} = !\ \mathsf{new}\, c_1.\mathsf{out}(c'_A, c_1).P'_A \ \mid !\ \mathsf{new}\, c_3.\mathsf{out}(c'_S, c_3).P'_S$$

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

$$\begin{aligned} P'_A &= \alpha'_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{c}\rangle).\\ &\quad\ \alpha'_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{b}, x'_{AB}, x'_B\rangle, k_{as})).\\ &\quad\ \alpha'_3 : \mathsf{out}(c_1, x'_B)\\ P'_S &= \gamma'_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{c}\rangle).\ \mathsf{new}\ k'_{ab}.\\ &\quad\ \gamma'_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle \mathsf{c}, k'_{ab}, \mathsf{enc}(\langle k'_{ab}, \mathsf{a}\rangle, k_{cs})\rangle, k_{as})) \end{aligned}$$

We consider also:

$$P''_{\mathsf{DS}} = !\ \mathsf{new}\, c_2.\mathsf{out}(c''_B, c_2).P''_B \ \mid !\ \mathsf{new}\, c_3.\mathsf{out}(c''_S, c_3).P''_S$$

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

$$\begin{aligned} P''_B &= \beta''_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y''_{AB}, \mathsf{c}\rangle, k_{bs}))\\ P''_S &= \gamma''_1 : \mathsf{in}(c_3, \langle \mathsf{c}, \mathsf{b}\rangle).\ \mathsf{new}\ k''_{ab}.\\ &\quad\ \gamma''_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle \mathsf{b}, k''_{ab}, \mathsf{enc}(\langle k''_{ab}, \mathsf{c}\rangle, k_{bs})\rangle, k_{cs})) \end{aligned}$$

The resulting protocols P^{1+}_{DS} and P^{2+}_{DS} are simple protocols. They are also type-compliant w.r.t. $(\mathcal{T}^+_{\mathsf{DS}}, \delta^+_{\mathsf{DS}})$ where $\mathcal{T}^+_{\mathsf{DS}}$ is an enriched version of $\mathcal{T}_{\mathsf{DS}}$ with new atomic types: τ_{c}, τ_{kcs}, $\tau_{kab'}$, and $\tau_{kab''}$. In particular, we have type-compliance for a notion of type that gives different types to k_{ab}, k'_{ab}, and k''_{ab}. The type τ_{c} is public.

The resulting dependency graph remains acyclic and is depicted in Figure 5. Note that there is an arrow from β''_1 to γ''_2 for the following reason. We have that

$$(\mathsf{enc}(\langle \tau_{kab''}, \tau_{\mathsf{c}}\rangle, \tau_{kbs}), 1.2.2) \in \rho_{\mathsf{out}}(u''_2 \delta^+_{\mathsf{DS}})$$

where u''_2 is the term occurring in the action labelled γ''_2. Intuitively, this is because the output labelled γ''_2 is an encryption with a compromised key k_{cs}, and thus the attacker could analyse this term and learn a term of type $\mathsf{enc}(\langle \tau_{kab''}, \tau_{\mathsf{c}}\rangle, \tau_{kbs})$. A term of such a type could be used to fill the input β''_1. This possible dependency is represented by an arrow from β''_1

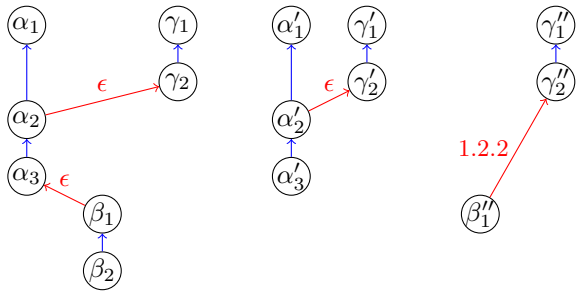

Fig. 5. Dependency graph for P^+_{DS} w.r.t. $(\mathcal{T}^+_{\mathsf{DS}}, \delta^+_{\mathsf{DS}})$

to γ''_2. The label indicates the position at which such a term is available in the output.

The resulting dependency graph is composed of three components that are completely disconnected. This reflects the fact that the protocol ensures that there is no interaction between a session involving honest participants, and sessions that may involve some dishonest participants. Also there is no confusion between messages used at the different stages of the protocol.

For a complete corruption scenario, we then need to consider the cases where the role A is played by the agent b and the role B is played by the agent a. The resulting dependency graph is obtained by symmetry from the one displayed in Figure 5. It remains acyclic and is composed of six disjoint components.

In the remaining of the section, we study several symmetric key protocols of the literature and discuss whether they fall in our decidable class. For all of them, we consider the complete corruption scenario as described above on the Denning-Sacco protocol.

B. Review of symmetric key protocols

Most of the protocols we considered from [23] actually fall in our decidable class. We sometimes need them to include some explicit tags. For some of them, we need to consider a refined version of our typing graph, and we consider the one obtained using our simple syntactic criterion to mark position appropriately. Our findings are summarised in Figure 6. We discuss below each protocol individually.

	Dependency graph		In our
	Normal	Refined	class
Denning-Sacco	✓	✓	yes
Needham-Schroeder		✓	yes
Otway-Rees		✓	yes
Yahalom (Paulson)		✓	yes
Wide-Mouthed-Frog	✓	✓	yes
Yahalom			no
Kao-Chow (modified)		✓	yes

Fig. 6. A ✓ means that the corresponding dependency graph is acyclic.

Denning-Sacco: This protocol forms our running example and its dependency graph is acyclic, without any tagging.

Needham-Schroeder: As discussed in Section IV-A, its dependency graph is not acyclic but its refined dependency graph is, even when considering the complete corruption scenario. We do not need to add explicit tags. However, contrary to what happens in the Denning-Sacco protocol, the resulting dependency graph is more complex. It is composed of six components connected with several arrows. This is due to the fact that to get type-compliance we have to give the same type to all the names that play the role of the key K_{ab} (resp. N_b) with no distinction between those involved in an honest or a dishonest session.

Otway-Rees: The tagged version of the Otway-Rees protocol can be informally described as follows.

$$A \to B : M, A, B, \{1, N_a, M, A, B\}_{K_{as}}$$
$$B \to S : M, A, B, \{1, N_a, M, A, B\}_{K_{as}}, \{2, N_b, M, A, B\}_{K_{bs}}$$
$$S \to B : M, \{3, N_a, K_{ab}\}_{K_{as}}, \{4, N_b, K_{ab}\}_{K_{bs}}$$
$$B \to A : M, \{3, N_a, K_{ab}\}_{K_{as}}$$

Note that, considering a scenario with no corruption, its untagged version can be shown to be simple and type-compliant by typing k_{ab} with the same type as $\langle m, \mathsf{a}, \mathsf{b}\rangle$. However, its dependency graph would be cyclic (a cycle will appear between the two actions of the role S). We therefore consider the tagged version of the Otway-Rees protocol. Its dependency graph is still cyclic but becomes acyclic when marking several positions. In particular, we have to mark

1) all the positions (in roles B and S) where M appears in cleartext;
2) the positions (in roles A and B) at which the variables modelling ciphertext forwarding occur; and
3) the positions in roles S (only those that involve the dishonest agent c) that correspond to N_a (in case c plays the role A) and N_b (in case c plays the role B).

The fact that this marking strategy is appropriate is a direct consequence of our syntactic criterion given in Lemma 1.

Similarly, the *Wide-Mouthed-Frog* and the *Yahalom (Paulson version)* protocols need to be explicitly tagged. Their refined dependency graphs are acyclic. Actually the normal dependency graph of the Wide-Mouthed-Frog protocol is already acyclic.

Yahalom: Consider now the original Yahalom protocol. Its tagged version can be informally described as follows.

1. $A \to B : A, N_a$
2. $B \to S : B, \{1, A, N_a, N_b\}_{K_{bs}}$
3. $S \to A : \{2, B, K_{ab}, N_a, N_b\}_{K_{as}}, \{3, A, K_{ab}\}_{K_{bs}}$
4. $A \to B : \{3, A, K_{ab}\}_{K_{bs}}, \{4, N_b\}_{K_{ab}}$

When considering only honest sessions, the corresponding dependency graph is acyclic. However, cycles appear if sessions with dishonest agents are considered. Intuitively, this is due to the fact that the nonce N_b sent at the final step is encrypted under the key K_{ab} which secrecy cannot be statically guaranteed. Therefore, the nonce N_b could potentially be learnt at step 4 and be reused earlier (at Step 2 for example). Note that such a protocol would be declared leaky in [15].

Kao-Chow: Again, it is necessary to tag this protocol to obtain type-compliance w.r.t. a relatively fine-grained typing system, and to avoid some cycles in the dependency graph.

1. $A \to S:\ A, B, N_a$
2. $S \to B:\ \{1, A, B, N_a, K_{ab}\}_{K_{as}}, \{2, A, B, N_a, K_{ab}\}_{K_{bs}}$
3. $B \to A:\ \{1, A, B, N_a, K_{ab}\}_{K_{as}}, \{3, N_a\}_{K_{ab}}, N_b$
4. $A \to B:\ \{4, N_b\}_{K_{ab}}$

Even with explicit tags, the resulting dependency graph contains a cycle due to the fact that, at the third step, B sends the nonce N_a under K_{ab}, which security cannot be statically asserted. However, N_a is intuitively public. We therefore slightly modify this protocol, assuming that S also sends N_a in clear at the second step, which suffices to obtained an acyclic (refined) dependency graph using our syntactic criterion stated in Lemma 1. This is a typical example where another marking strategy could be applied to (soundly) obtain an acyclic graph.

C. Detailed comparison with [15]

Sibylle Fröschle has recently obtained [15] a decidability result for an unbounded number of sessions, for the property of "leakiness". Our decidability result differs from Fröschle's result on several general points, in particular in terms of primitives and properties that can be handled. Strictly speaking the two results are incomparable since we study trace equivalence while [15] analyses "leakiness", a specially designed property that implies secrecy. In this section, we highlight the main similarities and differences of the two approaches.

Primitives. Fröschle's result applies to all standard cryptographic primitives (concatenation, symmetric and asymmetric encryption, hash, and signatures) while we only consider concatenation and symmetric encryption. This is due to the fact that our decidability result builds upon [11] (to limit ourselves to well-typed traces), which scope is limited to concatenation and symmetric encryption.

Properties. We consider more general security properties since we can decide any equivalence-based property (provided the processes fall into our class), while [15] only applies to the particular leakiness property. Leakiness enforces that a data (a nonce or a key) is either immediately deducible or secret. Note that leakiness is strictly stronger than secrecy. It disallows protocols with temporary secrets but it also discards some very reasonable protocols such as the Needham-Schroeder symmetric key protocol, due to parallel sessions between honest and dishonest agents. Indeed, assume A initiates (honestly) a session with C. Then the key K_{ac} generated by the server is not immediately deducible to the attacker since it is protected by K_{as} but will be deducible as soon as A forwards it to C under the key K_{cs}. The protocol will be declared leaky although no one cares about the secrecy of K_{cs}.

Dependency graph. One important common point between [15] and our result is the notion of dependency graph that should be acyclic. The graph defined in [15] reflects sequential dependencies similarly to our dependency graph. Regarding data dependencies, there is an edge between two actions as soon as their corresponding terms can be instantiated such that they share a common ciphertext as subterm. As a consequence, acyclicity can only be satisfied in a typed model. Therefore, [15] assumes that agents can recognise the type of a data, *e.g.* do not confuse a nonce with a ciphertext or a pair of nonces. Fröschle's result cannot consider protocols with ciphertext forwarding. In some cases where ciphertexts are just appended to the rest of the message, [15] devises a simple transformation. However, this transformation does not apply to protocols including more involved ciphertext forwarding such as the Denning-Sacco and the Needham-Schroeder protocols.

The graph considered in [15] is somewhat simpler (*i.e.* contains less arrows). In particular, this graph does not consider key dependencies (item 3 of Definition 9). This is due to the leakiness property: there is no temporary secret thus a key is either secret or public.

VI. Sketch of proof of our decidability results

We prove our decidability results (Theorems 1 and 2) in three main steps. In both cases, we bound the length of a witness of non-equivalence, and then conclude by invoking a decidability result for a bounded number of sessions (*e.g.* [11]).

Given two simple protocols P and Q, a *witness of non-inclusion* for $P \not\sqsubseteq Q$ is a trace tr for which there exists ϕ such that $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ and:

- either there does not exist ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$,
- or such a ψ exists and $\phi \not\sim \psi$.

A *witness of non-equivalence* is a trace tr that is a witness for $P \not\sqsubseteq Q$ or $Q \not\sqsubseteq P$.

Note that for a simple protocol, once the sequence tr is fixed, all the frames reachable through tr are actually in static equivalence, which ensures the unicity of ψ, if it exists, up-to static equivalence.

The three main steps of our proof can be summarised as follows:

1) We first rely on our type-compliance assumption. We show that we can restrict our attention to witnesses that are well-typed and we further show that each message occurring in such a trace can be computed *as soon as possible* (asap). Intuitively, recipes should refer to messages that occur as early as possible.
2) Then, we show that all the dependencies occurring in such a well-typed and asap trace comply with the dependency graph. Hence, we bound the *width* as well as the *depth* of such a witness exploiting the acyclicity of our dependency graph.
3) Lastly, we explain how to bound the length of a minimal witness.

A. Reducing equivalence

We first use a result from [11]. We recall it below in a version that fits the setting used in the present paper.

Theorem 3 ([11]): Let P and Q be two simple protocols type-compliant w.r.t. some structure-preserving typing systems

$(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$. We have that $P \not\approx Q$ if, and only if, there exists a witness of non-equivalence tr such that:

- either $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ for some ϕ and (tr, ϕ) is pseudo-well-typed w.r.t. $(\mathcal{T}_P, \delta_P)$.
- or $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ for some ψ and (tr, ψ) is pseudo-well-typed w.r.t. $(\mathcal{T}_Q, \delta_Q)$.

In the result stated above, (pseudo)-well-typed means that every variable of type τ has to be instantiated with a term of the same type. Since we consider atomic keys, some execution may fail when a protocol is about to output an encryption with a non atomic key. To detect this kind of behaviours, it is important to consider slightly ill-typed traces, *i.e.* well-typed traces where one special constant may be replaced by a special composed term: $\langle \omega, \omega \rangle$. For these reasons, we consider pseudo-well-typed traces.

Example 17: The traces (tr, ϕ), (tr', ϕ'_1), and (tr', ϕ'_2) given in Example 3 and 5 are well-typed. Indeed, even if in tr' the attacker replays a message (the one stored in w_3) coming from an old session, the message has the expected type, namely $\mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kbs})$, and thus the resulting trace is well-typed.

To bound the length of a witness, it is important to avoid some unnecessary detours, and this is the purpose of computing messages as soon as possible.

Definition 14: Given a frame $\phi = \{\mathsf{w}_1 \triangleright m_1, \ldots, \mathsf{w}_n \triangleright m_n\}$, and a message m such that $\phi \vdash m$. We say that R is an *asap recipe* of m if:

- R is a recipe of m, *i.e.* $R\phi\downarrow = m$, and
- for any $i \in \{0, \ldots, n\}$ such that $\phi_i \vdash m$, we have that $vars(R) \subseteq dom(\phi_i)$

where $\phi_i = \{\mathsf{w}_1 \triangleright m_1, \ldots, \mathsf{w}_i \triangleright m_i\}$ and ϕ_0 is the empty frame.

We say that a trace (tr, ϕ) of a protocol P is an *asap trace* if for any input recipe R occurring in tr, we have that R is an asap recipe of $R\phi\downarrow$ w.r.t. ϕ.

Example 18: The trace (tr, ϕ) given in Example 3 is not an asap trace. Indeed, $\mathsf{in}(ch_3, \mathsf{w}_1)$ occurs in tr and $\mathsf{w}_1\phi\downarrow = \langle \mathsf{a}, \mathsf{b} \rangle$. Thus w_1 is a recipe of $\langle \mathsf{a}, \mathsf{b} \rangle$. However, it is not an asap recipe since $\langle \mathsf{a}, \mathsf{b} \rangle$ is deducible from the empty frame (remember that a and b are public constants from Σ_0). For the same reason, (tr', ϕ'_1) and (tr', ϕ'_2) are not asap traces.

We are then able to show that, when looking for an attack, *i.e.* a witness of non-equivalence, we can further restrict our attention to consider pseudo-well-typed witnesses that are also asap.

B. Exploiting the dependency graph

Given an asap and pseudo-well-typed execution trace (tr, ϕ) of a simple protocol P, we can see it as a dag D (directed acyclic graph) whose vertices are actions of tr, and edges represent sequential dependencies and data dependencies. Note that such a dag can be computed simply from tr since sequential dependencies may be inferred from the channel names occurring in tr, and data dependencies are inferred from input recipes that occur in tr.

Our ultimate goal is to bound the length of tr, and thus the number of vertices in D. We first show that we are able to bound its depth and its width.

1) Bounding the depth of D: Intuitively, any sequential and data dependency occurring in a well-typed and asap trace is already present in the dependency graph. This is obvious regarding sequential dependencies since all these dependencies have been added in the dependency graph. However, regarding data dependencies, this result strongly relies on the fact that we consider well-typed and asap traces.

Proposition 1: Let P be a simple protocol type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$. Let $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ be an asap and pseudo-well-typed trace w.r.t. $(\mathcal{T}_P, \delta_P)$.

For any pair of actions $\mathsf{in}(d, R) \,/\, \mathsf{out}(c, \mathsf{w})$ occurring in tr with $\mathsf{w} \in vars(R)$, we have that $\alpha \rightarrow_G^+ \beta$ where:

- $\alpha, \beta \in \mathcal{L}(P)$ are the labels associated to the actions $\mathsf{in}(d, R)$ and $\mathsf{out}(c, \mathsf{w})$ respectively;
- $\rightarrow_G^+$ is the transitive closure of the relation $\rightarrow$ in the dependency graph G associated to P.

Note that, in the class of simple process, once the trace tr is fixed, the label $\alpha \in \mathcal{L}(P)$ of an action occurring in tr is uniquely defined.

This proposition is one of our key results. It requires to control data dependencies and in particular data dependencies that may occur due to keys: it may be necessary to decrypt to obtain a new key that in turn will be used to learn another key and so on. We show that our dependency graph actually captures all dependencies. When refining the dependency graph with appropriate marking, the main idea is that edges that are removed correspond to dependencies that can not happen in any asap trace.

Corollary 1: Let P be a simple protocol type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and with an acyclic (possibly refined) dependency graph G. Let $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ be an asap and pseudo-well-typed trace w.r.t. $(\mathcal{T}_P, \delta_P)$, and D its corresponding execution graph. We have that:

$$\mathsf{depth}(D) \leq \mathsf{depth}(G) + 1.$$

2) Bounding the width of D: The width of D is the maximal number of outgoing edges of any vertex of D. Actually, any recipe involved in an asap and pseudo-well-typed trace is of the form $C[R_1, \ldots, R_n]$ where C contains only constructors and each R_i contains only destructors. Since messages stored in the frame are well-typed, we can not stack more than $\|\mathsf{out}_P\|$ (maximal size of an outputted term in a well-typed trace) destructors in such a recipe. Note that some key chains may be needed to deduce a message, but the length of such a chain is bounded by $\mathsf{depth}(G)$. Hence, we have that each R_i involves no more than $(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$ recipe variables, and we have also that $n \leq \|\mathsf{in}_P\|$ where $\|\mathsf{in}_P\|$ denotes the maximal size of any input term in a well-typed trace. Thus, we have that:

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

C. Bounding the length of a minimal witness

To conclude, we need to bound the length of a minimal witness of non-equivalence. We have already seen that we can consider a witness that is asap and pseudo-well-typed, and we have bound the depth and the width of its associated dag. Still, the length of such a trace may be arbitrary long. We now show that we can bound the number of roots (vertices with no ingoing edge) in a minimal witness of non-equivalence.

Lemma 2: Let P and Q be two simple protocols type-compliant w.r.t. some structure-preserving typing systems $(\mathcal{T}_P, \delta_P)$ and $(\mathcal{T}_Q, \delta_Q)$, and such that $P \not\sqsubseteq Q$. Let (tr, ϕ) be a witness of non-inclusion which is asap, pseudo-well-typed and with minimal length, and D be its corresponding execution graph. We have that:

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

There are two main reasons of non inclusion.

- Either Q is not able to mimic the last action of the trace tr. In that case, we prune D by selecting only the last action of tr and its (successive) sons.
- Or the resulting frames are not in static equivalence. Consider an equality test $R_1 = R_2$ that witnesses non static equivalence. We show that R_1 and R_2 can be chosen to be destructor-only, and we bound the number of recipe variables involved in R_1 and R_2 by $(1 + \|\mathsf{out}_P\|)^{\mathsf{depth}(G)+1}$. Thus, in this case, we prune D by selecting the actions producing these variables and their successive sons.

In conclusion, we have bound the (minimal) length of a witness of non equivalence. This, in turn, bounds the number of sessions. We then conclude using *e.g.* [11] since trace equivalence is decidable for a bounded number of sessions. Since trace equivalence is NP for a bounded number of sessions [8], we deduce decidability in triple exponential time, in the size of the protocols.

VII. Conclusion

We have obtained the first decidability result for trace equivalence, for an unbounded number of sessions and unrestricted nonces.

Generating a structure-preserving typing system (actually the more fine-grained one) for which type-compliance is satisfied, and checking acyclicity of the resulting dependency graph is not difficult but rather cumbersome. We plan to devise a script to perform these steps automatically. We also plan to study how to relax some of our assumptions. First, we think that the "simple protocols" assumption could be relaxed to consider action-determinate protocols. Second, we plan to investigate other criteria to soundly remove edges in the dependency graph, in order to get rid of meaningless cycles. Lastly, our result applies only to protocols with concatenation and symmetric encryption. We inherit this restriction from [11]. We believe that once [11] will be extended to all standard primitives then our decidability result will extend as well.

The current complexity of our result is too high to use existing tools that decide trace equivalence for a bounded number of sessions (they typically handle up to 2-3 sessions). However, since our result bounds quite precisely the form of a minimal attack, it seems possible to improve its complexity and to use model-checkers instead.

Our decidability result intuitively encompasses decidability of secrecy, expressed as a trace property, since secrecy can be encoded using trace equivalence. We believe that our proof technique could be applied to decide authentication properties as well, for which we are not aware of any decidability result. The main difficulty induced by authentication properties is that authentication implicitly introduces disequalities (there might be an attack because agent B received a message *different* from the one sent by agent A). However, deciding trace equivalence also requires a careful treatment of disequalities. As future work, we plan to formally apply our technique to obtain decidability of a fragment of trace properties that encompasses secrecy and authentication.

References

[1] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *14th Computer Security Foundations Workshop (CSFW'01)*. IEEE Computer Society Press, 2001.

[2] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "The AVISPA Tool for the automated validation of internet security protocols and applications," in *17th International Conference on Computer Aided Verification, CAV'2005*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds., vol. 3576. Edinburgh, Scotland: Springer, 2005, pp. 281–285.

[3] C. Cremers, "The Scyther Tool: Verification, falsification, and analysis of security protocols," in *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.*, ser. Lecture Notes in Computer Science, vol. 5123/2008. Springer, 2008, pp. 414–418.

[4] M. Bruso, K. Chatzikokolakis, and J. den Hartog, "Formal verification of privacy for RFID systems," in *23rd Computer Security Foundations Symposium (CSF'10)*, 2010.

[5] M. Backes, C. Hritcu, and M. Maffei, "Automated verification of remote electronic voting protocols in the applied pi-calculus," in *21st IEEE Computer Security Foundations Symposium (CSF'08)*. IEEE Computer Society, 2008, pp. 195–209.

[6] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, "Undecidability of bounded security protocols," in *Workshop on Formal Methods and Security Protocols*, Trento, Italia, 1999.

[7] M. Rusinowitch and M. Turuani, "Protocol Insecurity with Finite Number of Sessions and Composed Keys is NP-complete," *Theoretical Computer Science*, vol. 299, pp. 451–475, April 2003.

[8] M. Baudet, "Deciding security of protocols against off-line guessing attacks," in *12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM Press, 2005.

[9] H. Comon-Lundh and V. Cortier, "New decidability results for fragments of first-order logic and application to cryptographic protocols," in *14th International Conference on Rewriting Techniques and Applications (RTA'2003)*, ser. LNCS, vol. 2706. Springer, 2003.

[10] R. Chrétien, V. Cortier, and S. Delaune, "From security protocols to pushdown automata," in *40th Int. Colloquium on Automata, Languages and Programming (ICALP'13)*, 2013.

[11] R. Chrétien, V. Cortier, and S. Delaune, "Typing messages for free in security protocols: the case of equivalence properties," in *Proceedings of the 25th International Conference on Concurrency Theory (CONCUR'14)*, ser. LNCS, vol. 8704. Rome, Italy: Springer, Sep. 2014, pp. 372–386.

[12] G. Lowe, "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.

[13] R. Ramanujam and S. P. Suresh, "Tagging makes secrecy decidable with unbounded nonces as well," in *23rd Conference of Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, ser. LNCS. Springer, 2003, pp. 363–374.
[14] D. J. Dougherty and J. D. Guttman, "Decidability for lightweight Diffie-Hellman protocols," in *IEEE Symposium on Computer Security Foundations (CSF'14)*, 2014.
[15] S. Fröschle, "Leakiness is decidable for well-founded protocols?" in *Proceedings of the 4th Conference on Principles of Security and Trust (POST'15)*, ser. Lecture Notes in Computer Science. London, UK: Springer, Apr. 2015.
[16] H. Comon-Lundh and V. Cortier, "Computational soundness of observational equivalence," in *15th ACM Conference on Computer and Communications Security (CCS'08)*. ACM Press, 2008.
[17] B. Blanchet and A. Podelski, "Verification of cryptographic protocols: Tagging enforces termination," in *Foundations of Software Science and Computation Structures (FoSSaCS'03)*.
[18] V. Cortier, M. Rusinowitch, and E. Zalinescu, "Relating two standard notions of secrecy," *Logical Methods in Computer Science*, vol. 3, no. 3, July 2007.
[19] A. Datta, A. Derek, J. Mitchell, and B. Warinschi, "Computationally sound compositional logic for key exchange protocols," in *Proceedings of 19th IEEE Computer Security Foundations Workshop (CSF'2006)*, 2006, pp. 321–334.
[20] M. Abadi and P. Rogaway, "Reconciling two views of cryptography (the computational soundness of formal encryption)," *Journal of Cryptology*, vol. 2, pp. 103–127, 2002.
[21] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *28th Symposium on Principles of Programming Languages (POPL'01)*. ACM Press, 2001.
[22] N. Dershowitz and J.-P. Jouannaud, "Rewrite systems," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier, 1990.
[23] J. Clark and J. Jacob, "A survey of authentication protocol literature: Version 1.0," 1997.
[24] V. Cheval, V. Cortier, and S. Delaune, "Deciding equivalence-based properties using constraint solving," *Theoretical Computer Science*, vol. 492, pp. 1–39, Jun. 2013.
[25] R. Needham and M. Schroeder, "Using encryption for authentification in large networks of computers," *Communications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.
[26] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, ser. LNCS, T. Margaria and B. Steffen, Eds., vol. 1055. Springer-Verlag, march 1996, pp. 147–166.

APPENDIX

Needham Schroeder protocol

We propose a modelling of the Needham Schroeder protocol for the semi-complete scenario in our formalism. Here, agents a and b are both willing to start sessions with a malicious agent c. Below, k_{as}, k_{bs}, k_{cs}, k_{ab}, k'_{ab}, k''_{ab}, n_a, n'_a, n_b, n''_b, and k are names, whereas a, b, c, req, rep, m_1 and m_2 are constants from Σ_0. As in Example 5, we model the security of the exchanged key by requiring that $P^{1+}_{\mathsf{NS}} \approx P^{2+}_{\mathsf{NS}}$ where P^{1+}_{NS} and P^{2+}_{NS} are defined as follows:

- P^{1+}_{NS} is as the protocol P^{+}_{NS} (see below) but we add the instruction "$\beta_4 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_1, y_{AB}))$" at the end of P_B;
- P^{2+}_{NS} is as the protocol P^{+}_{NS} but we add the instruction "$\mathsf{new}\, k.\beta_4 : \mathsf{out}(c_2, \mathsf{enc}(\mathsf{m}_2, k))$" at the end of P_B.

$$
\begin{aligned}
P^{+}_{\mathsf{NS}} = \;& !\,\mathsf{new}\; c_1.\mathsf{out}(c_A, c_1).P_A \mid !\,\mathsf{new}\; c_2.\mathsf{out}(c_B, c_2).P_B \\
& \mid !\,\mathsf{new}\; c_3.\mathsf{out}(c_S, c_3).P_S \\
& \mid !\,\mathsf{new}\; c_1.\mathsf{out}(c'_A, c_1).P'_A \mid !\,\mathsf{new}\; c_3.\mathsf{out}(c'_S, c_3).P'_S \\
& \mid !\,\mathsf{new}\; c_2.\mathsf{out}(c''_B, c_2).P''_B \mid !\,\mathsf{new}\; c_3.\mathsf{out}(c''_S, c_3).P''_S
\end{aligned}
$$

where processes P_A, P'_A, P_B, P''_B, P_S, P'_S, P''_S are as follows.

$$
\begin{aligned}
P_A = \;& \mathsf{new}\; n_a. \\
& \alpha_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{b}, n_a \rangle). \\
& \alpha_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle n_a, \mathsf{b}, x_{AB}, x_B \rangle, k_{as})). \\
& \alpha_3 : \mathsf{out}(c_1, x_B). \\
& \alpha_4 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{req}, x_{NB} \rangle, x_{AB})). \\
& \alpha_5 : \mathsf{out}(c_1, \mathsf{enc}(\langle \mathsf{rep}, x_{NB} \rangle, x_{AB}))
\end{aligned}
$$

$$
\begin{aligned}
P'_A = \;& \mathsf{new}\; n'_a. \\
& \alpha'_1 : \mathsf{out}(c_1, \langle \mathsf{a}, \mathsf{c}, n'_a \rangle). \\
& \alpha'_2 : \mathsf{in}(c_1, \mathsf{enc}(\langle n'_a, \mathsf{c}, x'_{AB}, x'_B \rangle, k_{as})). \\
& \alpha'_3 : \mathsf{out}(c_1, x'_B). \\
& \alpha'_4 : \mathsf{in}(c_1, \mathsf{enc}(\langle \mathsf{req}, x'_{NB} \rangle, x'_{AB})). \\
& \alpha'_5 : \mathsf{out}(c_1, \mathsf{enc}(\langle \mathsf{rep}, x'_{NB} \rangle, x'_{AB}))
\end{aligned}
$$

$$
\begin{aligned}
P_B = \;& \beta_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y_{AB}, \mathsf{a} \rangle, k_{bs})).\mathsf{new}\; n_b. \\
& \beta_2 : \mathsf{out}(c_2, \mathsf{enc}(\langle \mathsf{req}, n_b \rangle, y_{AB})). \\
& \beta_3 : \mathsf{in}(c_2, \mathsf{enc}(\langle \mathsf{rep}, n_b \rangle, y_{AB}))
\end{aligned}
$$

$$
\begin{aligned}
P''_B = \;& \beta''_1 : \mathsf{in}(c_2, \mathsf{enc}(\langle y''_{AB}, \mathsf{c} \rangle, k_{bs})).\mathsf{new}\; n''_b. \\
& \beta''_2 : \mathsf{out}(c_2, \mathsf{enc}(\langle \mathsf{req}, n''_b \rangle, y''_{AB})). \\
& \beta''_3 : \mathsf{in}(c_2, \mathsf{enc}(\langle \mathsf{rep}, n''_b \rangle, y''_{AB}))
\end{aligned}
$$

$$
\begin{aligned}
P_S = \;& \gamma_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{b}, z_{NA} \rangle).\mathsf{new}\; k_{ab}. \\
& \gamma_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle z_{NA}, \mathsf{b}, k_{ab}, \mathsf{enc}(\langle k_{ab}, \mathsf{a} \rangle, k_{bs}) \rangle, k_{as}))
\end{aligned}
$$

$$
\begin{aligned}
P'_S = \;& \gamma'_1 : \mathsf{in}(c_3, \langle \mathsf{a}, \mathsf{c}, z'_{NA} \rangle).\mathsf{new}\; k'_{ab}. \\
& \gamma'_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle z'_{NA}, \mathsf{c}, k'_{ab}, \mathsf{enc}(\langle k'_{ab}, \mathsf{a} \rangle, k_{cs}) \rangle, k_{as}))
\end{aligned}
$$

$$
\begin{aligned}
P''_S = \;& \gamma''_1 : \mathsf{in}(c_3, \langle \mathsf{c}, \mathsf{b}, z''_{NA} \rangle).\mathsf{new}\; k''_{ab}. \\
& \gamma''_2 : \mathsf{out}(c_3, \mathsf{enc}(\langle z''_{NA}, \mathsf{b}, k''_{ab}, \mathsf{enc}(\langle k''_{ab}, \mathsf{c} \rangle, k_{bs}) \rangle, k_{cs}))
\end{aligned}
$$

To ensure the protocol is type-compliant, we need to consider a typing system $(\mathcal{T}^{+}_{\mathsf{NS}}, \delta^{+}_{\mathsf{NS}})$ such that x_{NB}, x'_{NB}, n_b and n''_b share the same type, τ_{nb}; x_{AB}, x'_{AB}, y_{AB}, y''_{AB}, k_{ab}, k'_{ab}, k''_{ab}, k have all type τ_{kab}; and both n_a and z_{NA} (resp. n'_a and z'_{NA}) have type τ_{na}. Which implies that $\delta^{+}_{\mathsf{NS}}(x_B) = \mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kbs})$, $\delta^{+}_{\mathsf{NS}}(x'_B) = \mathsf{enc}(\langle \tau_{kab}, \tau_{\mathsf{a}} \rangle, \tau_{kcs})$ and both m_1 and m_2 must have type τ_{m}. Moreover, τ_{kas} and τ_{kbs} are honest types whereas τ_{a}, τ_{b}, τ_{c}, τ_{m}, τ_{req}, and τ_{rep} are public types.

The dependency graph for P^{i+}_{NS} (for any $i \in \{1, 2\}$) w.r.t. this scenario is then given in Figure 7, split into three graphs displaying edges of type 1, 2 and 3 respectively. Applying the syntactic criterion from Lemma 1 enables us to consider $(\alpha_5, 1.2)$ and $(\alpha'_5, 1.2)$ as appropriate marked position in P^{1+}_{NS} and P^{2+}_{NS}, which allows us to discard type 2 arrows from α_4, β_3, α'_4 and β''_3 towards α_5 and α'_5 with label 1.2.

Edges towards α'_3 (resp. γ''_2) are all with label 1.1 (resp. 1.2.2.1) and exist because both labels correspond to outputs of keys known to the agent c (encrypted by k_{cs}). As all keys generated by the server share the same type τ_{kab}, any use of such a key create an edge towards those two nodes. Similarly, as nonces created by b and outputted at labels β_2 and β''_2 share the same type τ_{nb} and are encrypted by keys of (non-honest) type τ_{kab}, inputs using such nonces all point towards these nodes. Finally, arrows with label ϵ correspond to regular executions of the protocol, albeit with some collision between messages because of the previously detailed equal types.

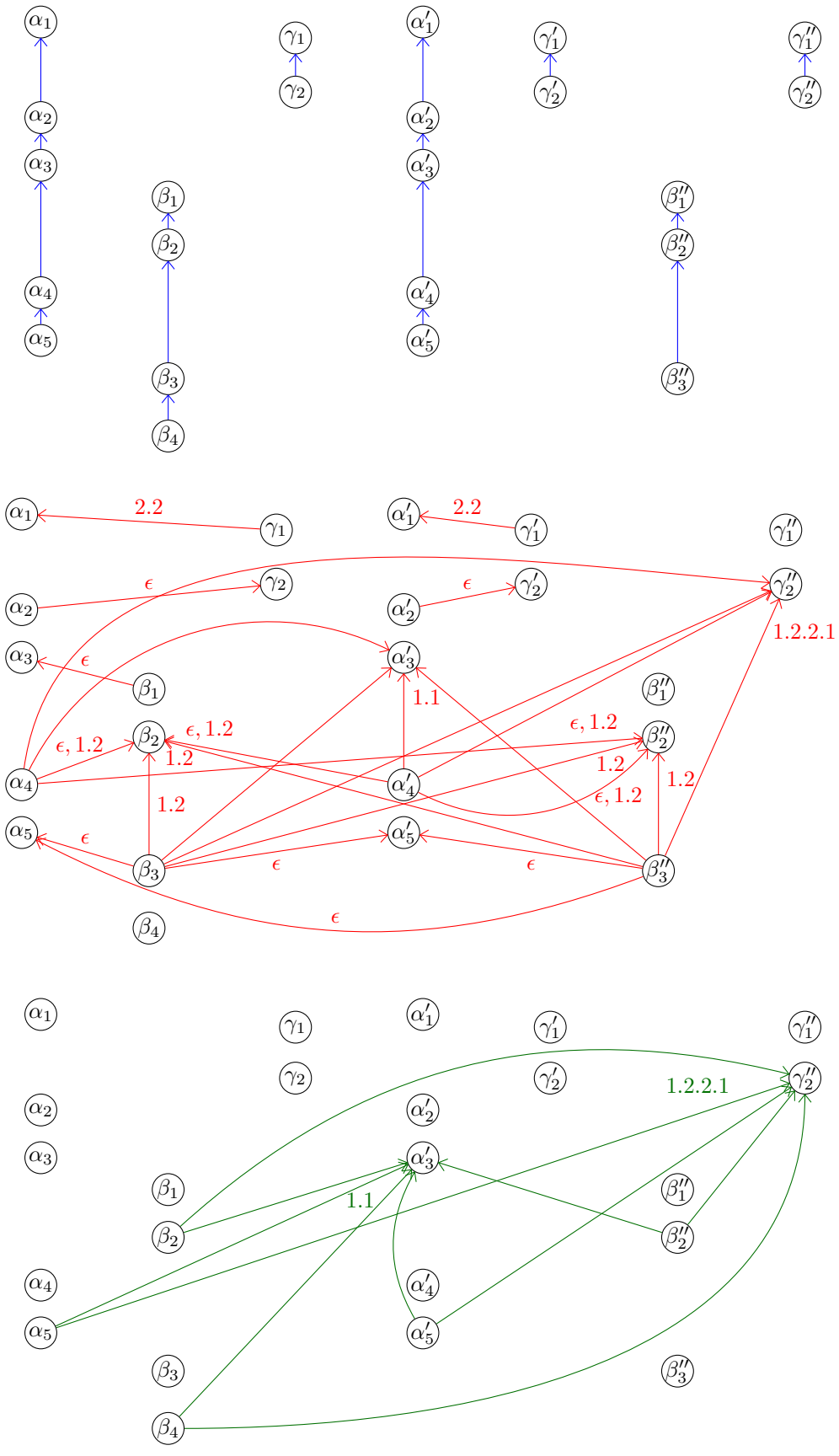

Fig. 7. Dependency graph for P^{i+}_{NS} w.r.t. $(\mathcal{T}^{+}_{\mathsf{NS}}, \delta^{+}_{\mathsf{NS}})$