A small bound on the number of sessions for security protocols

Véronique Cortier, Antoine Dallon, Stéphanie Delaune

▶ **To cite this version:**

Véronique Cortier, Antoine Dallon, Stéphanie Delaune. A small bound on the number of sessions for security protocols. CSF 2022 - 35th IEEE Computer Security Foundations Symposium, Aug 2022, Haifa, Israel. hal-03473179

HAL Id: hal-03473179
https://hal.inria.fr/hal-03473179

Submitted on 25 Jan 2022

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

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

A small bound on the number of sessions for security protocols

Véronique Cortier
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Antoine Dallon
DGA MI, Bruz, France

Stéphanie Delaune
Univ Rennes, CNRS, IRISA, France

***Abstract*—Bounding the number of sessions is a long-standing problem in the context of security protocols. It is well known that even simple properties like secrecy are undecidable when an unbounded number of sessions is considered. Yet, attacks on existing protocols only require a few sessions.**

In this paper, we propose a sound algorithm that computes a sufficient set of scenarios that need to be considered to detect an attack. Our approach can be applied for both reachability and equivalence properties, for protocols with standard primitives that are type-compliant (unifiable messages have the same type). Moreover, when equivalence properties are considered, else branches are disallowed, and protocols are supposed to be simple (an attacker knows from which role and session a message comes from). Since this class remains undecidable, our algorithm may return an infinite set. However, our experiments show that on most basic protocols of the literature, our algorithm computes a small number of sessions (a dozen). As a consequence, tools for a bounded number of sessions like DeepSec can then be used to conclude that a protocol is secure for an unbounded number of sessions.

I. Introduction

For several decades, decision procedures have been developed for the automatic analysis of security protocols. Various security properties can be considered. Secrecy and authentication are usually formalized as reachability properties, while anonymity, untraceability, and other privacy properties are expressed as equivalence properties. Such properties are known to be undecidable in general [25]. However, if a bounded number of sessions is considered, then reachability properties as well as trace equivalence are decidable, see e.g. [32], [31], [5], [10], [22].

In practice, attacks exploit only a few sessions. Let us first clarify what we call sessions. A protocol defines several roles (client, server, certificate authority, etc.). Each role is a program that may be run several times, each run corresponds to a *session* of the role. So the number of sessions will be the total number of programs that can be run (once). Written in terms of processes, this corresponds to the total number of processes in parallel, with no replication. Up to our knowledge, extreme cases are when 5-6 sessions are needed for an attack. For example, the Triple Handshakes Attack on TLS [6] requires an honest client C and an honest server S that each runs 3 subprograms, yielding 6 sessions (or even less, depending on how programs are divided). The traceability attack on electronic passports [13] requires one honest run between the reader and the passport and then a replay against a passport, thus 3 sessions. Hence a tempting heuristic is to conclude that either an attack can be found within a few sessions, or the protocol is secure. Unfortunately, there is absolutely no formal guarantee that this is indeed the case. It is possible to construct protocols for which an arbitrary number of sessions can be needed for attacks. Hence, bounding the number of sessions is a long-standing problem. The goal is to identify criteria, achieved in practice, such that if there is an attack, then there is an attack within an *a priori* bounded number of sessions.

Related work. Several results have studied this question.

- Sybille Fröschle [27] proposes a decidability result for the "leakiness" property, that guarantees that all data are either public or secret. This excludes protocols with temporary secrets. [27] holds for typed protocols: an agent expecting a nonce or a key cannot accept a ciphertext. The considered primitives are encryption and concatenation only.
- In [16], [21], the notion of typed protocols is relaxed to consider type-compliance, that intuitively requires that unifiable messages have the same type. While protocol agents may receive arbitrary messages, type-compliance ensures the existence of a well-typed witness when an attack exists. In [16], [21], the notion of dependency graph is introduced with the aim to characterize how actions depend from the other ones. For protocols with an *acyclic* dependency graph, the number of sessions can be bounded and hence reachability and equivalence properties are decidable. This result assumes protocols to be simple (actions can be precisely identified) and else branches are disallowed.
- [23] defines the notion of "depth-boundedness" that restricts intuitively the number of nested encryptions. It is shown that secrecy is decidable for depth-bounded protocols. As for [27], this does not cover equivalence properties. [24] generalizes the notion of depth-boundedness to support a wider variety of cryptographic primitives.

All these results provide new decidability results: they identify classes of protocols for which secrecy (and some-

The research leading to these results has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 714955-POPSTAR), as well as from the French National Research Agency (ANR) under the project TECAP, and the Chaire IA ASAP.

times equivalence) is decidable, for an unbounded number of sessions and fresh nonces. In passing, they bound the number of sessions. However, they do not provide explicit bounds or the bounds are not practical. For example, [21] gives a bound of 10^{19} sessions for the simple Denning-Sacco protocol and the bound is even larger for more complex protocols.

Many other works have studied the analysis of security protocols in a symbolic setting, aiming at reducing the search space. Some approaches show that it is sufficient to consider *typed* attack traces, where messages follow some format [3], [15], [29]. Other results reason on the origination of messages (see e.g. [28]).

Contribution. Our main contribution is to show that, under assumptions similar to [21], it is possible to efficiently bound the number of sessions. Our result holds both for reachability and equivalence properties and for a generic class of equational theories that encompasses standard cryptographic primitives (symmetric and asymmetric encryption, signatures, hash). We consider the class of protocols that are type-compliant, which is intuitively guaranteed as soon as two encrypted messages of the protocol cannot be confused. Type-compliance can often be achieved by simply adding a tag, that indicates at which step the message has been created. Such a tagging scheme is usually considered as a good practice since it avoids attacks and is shown to ensure type-compliance for protocols with symmetric encryption and pairing [15]. Moreover, when equivalence properties are considered, else branches are disallowed, and protocols must also be simple, i.e. an attacker can identify from which participant and which session a message originates from.

Under these assumptions, we provide an algorithm that explores the actions of a protocol and computes a bound on the number of sessions needed for an attack. It is important to note that our assumptions do not yield a decidable class, hence our algorithm can also return that no bound could be found. On the other hand, we do compute a bound for any protocol with an acyclic graph as defined in [21], hence our algorithm covers the decidable class introduced in [21]. Compared to [21], the major difference is that we provide a small, usable, bound where [21] is impractical. We also provide a bound in slightly more cases but these additional cases cover contrived examples only. Actually, instead of just a bound on the number of sessions, our algorithm provides a list of scenarios that need to be considered. Each scenario corresponds to a precise number of replications of each role of the protocol, possibly truncated. For example, maybe an attack needs two replications of role B and only twice the two first steps of role A. This more precise information can be helpful when running tools for the protocol analysis.

We prove that our algorithm is correct: if there is an attack, then there is an attack covered by one of the returned scenario. The proof involves two main steps. First, we use the fact that if there is an attack, then there is a well-typed attack, that is, an attack where all the messages comply with the expected format in the protocol. This result heavily relies on a previous typing result [14], extended to else branches for reachability properties. Then, in a second step, we need to show that our algorithm explores all possible scenarios, that is, we show that a well-typed attack of minimal size (minimal length and small attacker steps) is necessarily covered by at least one of our scenarios.

Implementation. We have implemented our procedure in a tool HowMany that *(i)* first checks whether our assumptions, in particular type-compliance, are satisfied, *(ii)* then recursively computes, for each action of the protocol, how this action can be reached from other steps of the protocol. This yields the set of scenarios that need to be considered. If a loop is detected, then no bound can be found. We experimented our tool on several protocols of the literature. As already noticed in [21], they all satisfy our assumptions, possibly after tagging messages. For most protocols, HowMany can find a bound of size 3 to 55 sessions, in the worst case. For example, HowMany computes a bound of 3 sessions only for the Denning-Sacco protocol, to be compared with the previous 10^{19} bound. Even in the case where 55 sessions may be needed (for the Kao-Chow protocol), HowMany actually provides finer grain information: one can either consider one scenario with 55 sessions in parallel or, instead, 385 different scenarios of at most 29 sessions each, and for which the analysis can be done independently.

Discussion. Although they do not come with termination guarantees, tools like ProVerif [7] or Tamarin [30], dedicated to the analysis of protocols for an unbounded number of sessions, can already analyze efficiently the protocols considered in our experiments. Hence, our first and main contribution is theoretical: we hope to contribute to a better understanding on the interplay between sessions and attacks. In particular, our results show that yes, for standard protocols, it is possible to find a reasonable bound on the number of sessions. Moreover, our approach allows to extend the scope of existing tools developed for a bounded number of sessions such as Avispa [4], Maude-NPA [26], DeepSec [12], or SAT-Equiv [19]. Despite the state-explosion issue due to the intrinsic complexity (NP-complete) of the bounded case, a major improvement has been seen in the number of sessions that can be covered by these tools. For example, DeepSec can analyze up to 10-50 sessions for standard protocols of the literature. SAT-Equiv can even analyze up to 60-400 sessions but covers a much smaller fragment of protocols. Hence HowMany can be used as a small add-on to these tools to conclude to the security of an unbounded number of sessions when HowMany successfully computes a bound. In our experiments, SAT-Equiv was able to prove security for an unbounded number of sessions in all cases, while DeepSec faces a time out (set to 24h) in about 15% of the cases and succeeds otherwise.

All files related to the implementation and case studies are available as supplementary material in [20].

II. Model

A. Messages

As usual, messages are modeled with a term algebra. Private data, such as private keys, private randomness and nonces, are represented by a set of *names* $\mathcal{N}$. Security protocols may also use public data, like tags, or error messages, that are represented by (public) *constants* of Σ_0. Constants also model other data known from the attacker, like corrupted private keys, nonces generated by the attacker or her keys. For technical reasons (related to the typing result [14]), the attacker is also given non-atomic constants. We consider two sorts, atom and bitstring. Any name in $\mathcal{N}$ is of the sort atom, while Σ_0 is split as $\Sigma_0 = \Sigma_0^{\mathsf{atom}} \uplus \Sigma_0^{\mathsf{bitstring}}$, where the constants in Σ_0^{atom} are of sort atom and the constants in $\Sigma_0^{\mathsf{bitstring}}$ are of sort bitstring. We assume an infinite number of names in $\mathcal{N}$, constants in Σ_0^{atom} and constants in $\Sigma_0^{\mathsf{bitstring}}$, such that protocol agents can always generate fresh nonces, and the attacker can always generate new keys and create new messages. Messages (or an unknown part of them) expected by a party of a protocol are represented through *variables* in $\mathcal{X}$. Another set $\mathcal{W}$ of variables is used to refer to messages learnt by the attacker. Most often, those messages result from an output. Typically, variables in $\mathcal{X}$ are denoted x, y, z, whereas variables in $\mathcal{W}$ are denoted $\mathsf{w}_1, \mathsf{w}_2, \ldots$. Names in $\mathcal{N}$ are denoted n, m and k or sk when they are keys, while constants are denoted a, b, c. We refer to variables, names and constants through the generic term of *data*, while the word atomic or atom shall be used only for data of sort atom.

We also need to model cryptographic operations, like encryption, decryption, mac, hash function,... These operations are represented by function symbols. A *signature* Σ is a set of function symbols with their arity. We distinguish between three types of symbols. We consider *constructor* symbols like encryption, in Σ_{c}, *destructor* symbols, like decryption, in Σ_{d}, and *test* symbols, in Σ_{test}, i.e. $\Sigma = \Sigma_{\mathsf{c}} \uplus \Sigma_{\mathsf{d}} \uplus \Sigma_{\mathsf{test}}$. The sets Σ_{d} and Σ_{test} do not contain constant symbols, i.e. function symbols of arity 0. Given a signature Σ and a set of data D, the set of *terms* built from Σ and D is denoted $\mathcal{T}(\Sigma, D)$. *Constructor terms* on D are terms in $\mathcal{T}(\Sigma_{\mathsf{c}}, D)$. We denote $vars(u)$ the set of variables that occur in a term u. A term is *ground* if it contains no variable. Given a substitution σ, we denote $dom(\sigma)$ its *domain*, $img(\sigma)$ its *image*, and $u\sigma$ its application to a term u. The *positions* of a term are defined as usual. Given a term t, the function symbol occurring at position ϵ in t is denoted $\mathsf{root}(t)$, and we denote $St(t)$ the set of the *subterms* of t. Two terms u_1 and u_2 are *unifiable* when there exists a substitution σ such that $u_1\sigma = u_2\sigma$. The most general unifier between u_1 and u_2 is denoted $mgu(u_1, u_2)$.

Any constructor f comes with its sort, i.e.

$$\mathsf{f} : (s_1 \times \ldots \times s_n) \to s_0$$

where n is the arity of f, $s_0 = \mathsf{bitstring}$, and $s_i \in \{\mathsf{atom}, \mathsf{bitstring}\}$ for $1 \leq i \leq n$. Given a constructor term $t \in \mathcal{T}(\Sigma_{\mathsf{c}}, D)$, p is an *atomic position* of t if it corresponds to a position where an atom is expected, i.e., $p = p'.i$, $t|_{p'} = \mathsf{f}(t_1, \ldots, t_n)$, with $\mathsf{f} \in \Sigma_{\mathsf{c}} : (s_1 \times \ldots \times s_n) \to s_0$ and $s_i = \mathsf{atom}$. We say that a constructor term t is *well-sorted* if $t|_p \in \mathcal{N} \uplus \mathcal{X} \uplus \Sigma_0^{\mathsf{atom}}$ for any atomic position p of t, i.e. any subterm is of the right sort.

Example 1. *Public-key encryption, signature, and pair can be modeled by considering* $\Sigma^{\mathsf{ex}} = \Sigma_{\mathsf{c}}^{\mathsf{ex}} \cup \Sigma_{\mathsf{d}}^{\mathsf{ex}} \cup \Sigma_{\mathsf{test}}^{\mathsf{ex}}$ *with:*

- $\Sigma_{\mathsf{c}}^{\mathsf{ex}} = \{\mathsf{aenc},\ \mathsf{pk},\ \mathsf{sign},\ \mathsf{vk},\ \mathsf{ok},\ \langle\ \rangle\}$;
- $\Sigma_{\mathsf{d}}^{\mathsf{ex}} = \{\mathsf{adec},\ \mathsf{getmsg},\ \mathsf{proj}_1,\ \mathsf{proj}_2\}$;
- $\Sigma_{\mathsf{test}}^{\mathsf{ex}} = \{\mathsf{check}\}$.

The symbols aenc *and* adec *(both of arity* 2*) represent resp. asymmetric encryption and decryption. The symbol* pk *(arity* 1*) is the key function:* $\mathsf{pk}(sk)$ *is the public key associated to the private key* sk*. Signatures are modeled with the symbol* sign *(arity* 2*). We assume that the content of a signature can be extracted (symbol* getmsg *of arity* 1*). Its validity can be checked with* check *(arity* 2*). The symbol* vk *(arity* 1*) is again a key function which modeled the verification key associated to a signing key. Pairing is modeled using* $\langle\ \rangle$ *(arity* 2*), and projection functions are denoted* proj_1 *and* proj_2 *(both of arity* 1*). The sort of our constructors are as follows:*

$$\begin{aligned} \mathsf{aenc}: &\ \ \mathsf{bitstring} \times \mathsf{bitstring} \to \mathsf{bitstring} \\ \mathsf{pk}: &\ \ \mathsf{atom} \to \mathsf{bitstring} \\ \mathsf{sign}: &\ \ \mathsf{bitstring} \times \mathsf{atom} \to \mathsf{bitstring} \\ \mathsf{vk}: &\ \ \mathsf{atom} \to \mathsf{bitstring} \\ \langle\ ,\ \rangle: &\ \ \mathsf{bitstring} \times \mathsf{bitstring} \to \mathsf{bitstring} \end{aligned}$$

Consider $k, ska \in \mathcal{N}$ *and* $ekc \in \Sigma_0$ *(all of sort* atom*), the term* $u_0 = \mathsf{aenc}(\mathsf{sign}(k, ska), \mathsf{pk}(ekc))$ *is a constructor term that represents an encryption (by the public key associated to the private key* ekc*) of a signature. This private key is modeled using a public constant, and is therefore known to the attacker. The atomic position of* u_0 *are* $p_1 = 2.1$ *and* $p_2 = 1.2$*, and since* ska *and* ekc *are indeed atoms,* u_0 *is well-sorted.*

Our main result relies on a typing result established in [14], and thus we need to consider a similar setting. In particular, in [14], a notion of *shape* is introduced, whose purpose is to describe the expected pattern of a message. For example, if asymmetric encryption is represented by $\mathsf{aenc}(m, \mathsf{pk}(k))$ then it should not be applied to other keys, e.g. $\mathsf{vk}(k)$. Formally, to each constructor function symbol f, we associate a linear term $\mathsf{f}(u_1, \ldots, u_n) \in \mathcal{T}(\Sigma_{\mathsf{c}}, \mathcal{X})$ denoted sh_{f} which is called the *shape* of f. Shapes have to be compatible, i.e. for any $\mathsf{f}(t_1, \ldots, t_n)$ occurring in a shape, we have that $\mathsf{sh}_{\mathsf{f}} = \mathsf{f}(t_1, \ldots, t_n)$. A term is *well-shaped* if it complies with the shapes, that is, any subterm of t, heading with a constructor symbol f is an instance of the shape of f. More formally, a constructor term $t \in \mathcal{T}(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{X})$ is well-shaped if for any $t' \in St(t)$ such that $\mathsf{root}(t') = \mathsf{f}$, we have that $t' = \mathsf{sh}_{\mathsf{f}}\sigma$ for some substitution σ. Given $D \subseteq \mathcal{N} \cup \Sigma_0 \cup \mathcal{X}$, we denote $\mathcal{T}_0(\Sigma_{\mathsf{c}}, D)$ the subset of constructor terms built over D that are well-shaped and well-sorted. Terms in $\mathcal{T}_0(\Sigma_{\mathsf{c}}, \mathcal{N} \cup \Sigma_0)$ are called *messages*.

Example 2. *Continuing Example 1, the shapes associated to each constructor symbol are as follows:*

$$\begin{array}{rl} \mathsf{aenc}: & \mathsf{sh}_{\mathsf{aenc}} = \mathsf{aenc}(x_1, \mathsf{pk}(x_2)) \\ \mathsf{pk}: & \mathsf{sh}_{\mathsf{pk}} = \mathsf{pk}(x_2) \\ \mathsf{sign}: & \mathsf{sh}_{\mathsf{sign}} = \mathsf{sign}(x_1, x_2) \\ \mathsf{vk}: & \mathsf{sh}_{\mathsf{vk}} = \mathsf{vk}(x_1) \\ \langle\,\rangle: & \mathsf{sh}_{\langle\,\rangle} = \langle x_1, x_2\rangle \end{array}$$

It is easy to see that these shapes are compatible. The term $u_0 = \mathsf{aenc}(\mathsf{sign}(k, ska), \mathsf{pk}(ekc))$ *is well-sorted and well-shaped thus in* $\mathcal{T}_0(\Sigma_{\mathsf{c}}, \mathcal{N} \cup \Sigma_0)$*, whereas* $\mathsf{aenc}(k, ekc)$ *is well-sorted but not well-shaped.*

To model the effect of destructors, we use a set $\mathcal{R}$ of rewriting rules built from Σ.

Example 3. *The properties of the primitives given in Example 1 are reflected through the following rewriting rules:*

$$\begin{array}{rcl} \mathsf{adec}(\mathsf{aenc}(x, \mathsf{pk}(y)), y) & \to & x \\ \mathsf{getmsg}(\mathsf{sign}(x, y)) & \to & x \\ \mathsf{check}(\mathsf{sign}(x, y), \mathsf{vk}(y)) & \to & \mathsf{ok} \end{array} \qquad \begin{array}{rcl} \mathsf{proj}_1(\langle x, y\rangle) & \to & x \\ \mathsf{proj}_2(\langle x, y\rangle) & \to & y \end{array}$$

Given a set $\mathcal{R}$ of rewriting rules, a term u can be *rewritten in* v using $\mathcal{R}$ if there is a position p in u, and a rewriting rule $\mathsf{g}(t_1, ..., t_n) \to t$ in $\mathcal{R}$ 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, in particular they do not contain destructor symbols. We consider sets of rewriting rules that yield convergent rewriting systems. As usual, we denote $\to^*$ the reflexive-transitive closure of $\to$, and $u{\downarrow}$ the *normal form* of a term u.

An attacker builds his own messages by applying public 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 $\mathcal{T}(\Sigma, \mathcal{W} \uplus \Sigma_0)$.

Example 4. *The set of rewriting rules given in Example 3 yields a convergent rewriting system. We have that:*

$$v = \mathsf{getmsg}(\mathsf{adec}(u_0, ekc)) \to \mathsf{getmsg}(\mathsf{sign}(k, ska)) \to k$$

Therefore, we have that $v{\downarrow} = k$*. The term* $R_0 = \mathsf{getmsg}(\mathsf{adec}(\mathsf{w}_1, ekc))$ *with* $\mathsf{w}_1 \in \mathcal{W}$ *is a recipe.*

B. Protocols

Our process algebra is inspired from the applied pi calculus [1], [2]. Our setting allows both pattern matching on the input construct and explicit filtering using the match construct.

We assume an infinite set $\mathcal{C}h$ of *channels* and an infinite set $\mathcal{L}$ of *labels*. We consider processes built using the following grammar:

$$\begin{array}{lll} P, P_1, \ldots, P_j, Q := & & \\ \quad 0 & & \text{null process} \\ \quad \mid\ \mathsf{in}^{\alpha}(c, u).P & & \text{input} \\ \quad \mid\ \mathsf{out}^{\alpha}(c, u).P & & \text{output} \\ \quad \mid\ \mathsf{new}\, n.P & & \text{name generation} \\ \quad \mid\ P \mid Q & & \text{parallel} \\ \quad \mid\ i : P & & \text{phase} \\ \quad \mid\ !P & & \text{replication} \\ \quad \mid\ \mathsf{new}\, c'.\mathsf{out}(c, c').P & & \text{channel generation} \\ \quad \mid\ \mathsf{match}\, x\, \mathsf{with} & & \text{filtering} \\ \qquad (u_1 \to P_1 \mid \ldots \mid u_j \to P_j) & & \end{array}$$

where $\alpha \in \mathcal{L}$, $u, u_1, \ldots, u_k \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \Sigma_0 \uplus \mathcal{N} \uplus \mathcal{X})$, $c, c' \in \mathcal{C}h$, $n \in \mathcal{N}$, and $i, j \in \mathbb{N}$. Given a process P, we denote $fv(P)$ the set of its *free variables*, i.e. those not bound by an input, nor by a filtering in a match construct. A *protocol* is a process with no free variable and with distinct labels.

Example 5. *Consider the following process* P*:*

$$P = \mathsf{in}(c, x).\mathsf{match}\, x\, \mathsf{with}\, \big(\mathsf{f}(y) \to \mathsf{out}(c, \mathsf{ok}) \mid\ y' \to \mathsf{out}(c, \mathsf{error})\ \big)$$

This process represents an agent who is waiting for a message. If the message received is of the form $\mathsf{f}(y)$ *(for some value of* y*), the constant* ok *will be sent. Otherwise, an error message will be emitted. The match construct is used to model conditional branching. We have that* $fv(P) = \emptyset$*. Indeed, the variable* x *is bound by the input construct, and* y *(resp.* y'*) is bound by the filtering* $\mathsf{f}(y)$ *(resp.* y'*) in the match construct.*

The construct $\mathsf{in}^{\alpha}(c, u)$ and $\mathsf{out}^{\alpha}(c, u)$ are the usual input and output actions, except that they are now decorated with labels α. These labels have no impact on the semantics of processes. They are used to refer to a precise action of a process and will be necessary to reason on the number of sessions needed for an attack. The construct $\mathsf{new}\, n.P$ generates a new name n and proceeds as P. The parallel composition of two processes P and Q is built as $P \mid Q$. We consider phases in our setting. They are useful to model protocols that have several phases, for example some general setup followed by the actual start of the main part of the protocol. A process $i : P$ may only react at phase i and can be discarded once the phase is strictly greater than i.

As usual, replication of processes is denoted $!P$. We consider a special construction to introduce a fresh public channel: $\mathsf{new}\, c'.\mathsf{out}(c, c').P$ generates a new channel c' and immediately publishes it on c. This is to allow for the generation of public channels while private channels are not considered in our setting. Finally, $\mathsf{match}\, x\, \mathsf{with}\, (u_1 \to P_1 \mid \ldots \mid u_j \to P_j)$ will try to unify the content of x with $u_1, \ldots, u_j$ (in order) and will proceed as P_i as soon as one successful unification is found for some u_i.

Example 6. *We consider a variant of the signature-based Denning-Sacco protocol as given in [8]. The protocol aims at ensuring the secrecy of the key* k *freshly generated by agent* A

and sent to B relying on signature and asymmetric encryption. Informally, we have that:

$$A \to B : \mathsf{aenc}(\mathsf{sign}(k, ska), \mathsf{pk}(ekb))$$

As shown in [8], this variant is vulnerable to an attack. Indeed, a dishonest agent C may reuse a signature $\mathsf{sign}(k_c, ska)$ sent by A to him to fool an honest agent B (see Example 7 for more details).

The two roles of A and B are represented by the following processes:

$$\begin{array}{rcl} P_A & = & \mathsf{new}\, k.\mathsf{out}^{\alpha_1}(c, \mathsf{aenc}(\mathsf{sign}(k, ska), \mathsf{pk}(ekb))) \\ P_B & = & \mathsf{in}^{\beta_1}(c, \mathsf{aenc}(\mathsf{sign}(x, ska), \mathsf{pk}(ekb))) \end{array}$$

where $k, ska, ekb \in \mathcal{N}$ and $x \in \mathcal{X}$. Then, the whole protocol is modeled by the parallel composition of these (replicated) processes, with an extra process P_K that reveals public keys to the attacker, i.e.

$$P_{\mathsf{DS}} = 1 :! P_A \mid 1 :! P_B \mid 0 : P_K$$

with $P_K = \mathsf{out}^{\gamma_1}(c, \mathsf{pk}(ekb)).\mathsf{out}^{\gamma_2}(c, \mathsf{vk}(ska))$.

Of course, we may want to consider a richer scenario involving a dishonest agent c. In this case, we can consider in addition the following processes (here $ekc \in \Sigma_0$):

$$\begin{array}{rcl} P'_A & = & \mathsf{new}\, k'.\mathsf{out}^{\alpha'_1}(c, \mathsf{aenc}(\mathsf{sign}(k', ska), \mathsf{pk}(ekc))) \\ P'_B & = & \mathsf{in}^{\beta'_1}(c, \mathsf{aenc}(\mathsf{sign}(x', ska), \mathsf{pk}(ekc))) \end{array}$$

yielding to the protocol $P'_{\mathsf{DS}} = P_{\mathsf{DS}} \mid 1 :! P'_A \mid 1 :! P'_B$.

Note that ekc is known to the attacker since $ekc \in \Sigma_0$.

The operational semantics of a process is defined as 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.
- ϕ is a *frame*, that is a substitution with a (finite) domain $dom(\phi) \subset \mathcal{W}$, and such that $img(\phi)$ only contains messages. Intuitively, ϕ corresponds to messages sent on the (public) network.
- σ is a substitution such that $fv(\mathcal{P}) \subseteq dom(\sigma)$, and $img(\sigma)$ only contains messages. It represents the current instantiation of protocol variables.

By abuse of notation, given a protocol P, we will write P for the *configuration* $(\{P\}, \emptyset, \emptyset, 0)$.

The relation $\xrightarrow{\ell}$ defining the operational semantics is described in Figure 1 and follows the intended semantics. Note that a process with a match is blocked if no possible match is found in the list of u_i. For the sake of conciseness, we sometimes write $P \uplus \mathcal{P}$ instead of $\{P\} \uplus \mathcal{P}$.

An action (or a step) is said *visible* when it is different from τ. The relation $\xrightarrow{\ell_1 \dots \ell_n}$ between configurations (where $\ell_1 \dots \ell_n$ is a sequence of actions) is defined as the transitive closure of $\xrightarrow{\ell}$. Given a sequence of visible actions tr and two configurations $\mathcal{K}$ and $\mathcal{K}'$, we write $\mathcal{K} \xRightarrow{\mathsf{tr}} \mathcal{K}'$ when there exists a sequence $\ell_1 \dots \ell_n$ such that $\mathcal{K} \xrightarrow{\ell_1 \dots \ell_n} \mathcal{K}'$ and tr is obtained from $\ell_1 \dots \ell_n$ by erasing all occurrences of τ.

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

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

Note that, by definition of $\mathsf{trace}(\mathcal{K})$, we have that $\mathsf{tr}\phi\!\downarrow$ only contains messages for any $(\mathsf{tr}, \phi) \in \mathsf{trace}(\mathcal{K})$.

Example 7. *Continuing Example 6, we have that $(\mathsf{tr}_0, \phi_0) \in \mathsf{trace}(P'_{\mathsf{DS}})$ where:*

$$\begin{array}{l} \mathsf{tr}_0 = \mathsf{out}^{\gamma_1}(c, \mathsf{w}_0).\mathsf{phase}\,1.\mathsf{out}^{\alpha'_1}(c, \mathsf{w}_1).\mathsf{in}^{\beta_1}(c, R_1); \\ \phi_0 = \{\mathsf{w}_0 \triangleright \mathsf{pk}(ekb), \mathsf{w}_1 \triangleright \mathsf{aenc}(\mathsf{sign}(k, ska), \mathsf{pk}(ekc))\} \end{array}$$

The recipe $R_1 \stackrel{\mathsf{def}}{=} \mathsf{aenc}(\mathsf{adec}(\mathsf{w}_1, ekc), \mathsf{w}_0)$ means that the attacker decrypts the message he received (from A) and he re-encrypts it with the public-key of B. Note that $R_1\phi_0\!\downarrow = \mathsf{aenc}(\mathsf{sign}(k, ska), \mathsf{pk}(ekb))$, and thus B will accept this message thinking that the key k is a secret shared with him and the agent A. However, we have that $R_0\phi_0\!\downarrow = k$ meaning that this key is actually known by the attacker (with $R_0 = \mathsf{getmsg}(\mathsf{adec}(\mathsf{w}_1, ekc))$).

C. Equivalence

Some security properties are expressed as equivalence properties where the attacker wins if she can distinguish between two scenarios. For example, Alice is traceable if an attacker can distinguish the case where Alice is taking part several times in a protocol from the case where different users are involved.

First, we say that an attacker can distinguish between two sequences of messages ϕ_1 and ϕ_2 if she can construct a test that holds in ϕ_1 and not ϕ_2. She can also distinguish if some evaluation (e.g. a decryption) succeeds in ϕ_1 and not ϕ_2.

Definition 1. *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 $\phi_2 \sqsubseteq_s \phi_1$.*

Then we can define *trace equivalence* between configurations $\mathcal{K}$ and $\mathcal{K}'$: any trace of a configuration $\mathcal{K}$ should have a corresponding trace in $\mathcal{K}'$ with the same visible actions and such that their frames are in static equivalence.

Definition 2. *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 exists $(\mathsf{tr}', \phi') \in \mathsf{trace}(\mathcal{K}')$ such that $\mathsf{tr} =_{\mathcal{L}} \mathsf{tr}'$ (where $=_{\mathcal{L}}$ is equality without taking into account labels from $\mathcal{L}$), 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 $\mathcal{K}' \sqsubseteq_t \mathcal{K}$.*

This notion of trace equivalence slightly differs from the original one (given e.g. in [11]), where the frames are required

IN $(i : \mathsf{in}^{\alpha}(c,u).P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\mathsf{in}^{\alpha}(c,R)} (i : P \uplus \mathcal{P}; \phi; \sigma \uplus \sigma_0; i)$
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 $(i : \mathsf{out}^{\alpha}(c,u).P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\mathsf{out}^{\alpha}(c,\mathsf{w})} (i : P \uplus \mathcal{P}; \phi \uplus \{\mathsf{w} \triangleright u\sigma\}; \sigma; i)$
with w a fresh variable from $\mathcal{W}$, and $u\sigma$ is a message.

NEW $(i : \mathsf{new}\, n.P \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\tau} (i : P\{^{m}/_{n}\} \uplus \mathcal{P}; \phi; \sigma; i)$
where $m \in \mathcal{N}$ is a fresh name.

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

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 bound variables are renamed.

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

MATCH $(\{i : \mathsf{match}\, x \,\mathsf{with}\, (u_1 \to P_1 \mid \ldots \mid u_j \to P_j)\} \uplus \mathcal{P}; \phi; \sigma; i) \xrightarrow{\tau} (i : P_{j_0} \uplus \mathcal{P}; \phi; \sigma \uplus \sigma_0; i)$
where j_0 is the smallest index such that $x\sigma$ and $u_{j_0}\sigma$ unify and $\sigma_0 = mgu(x\sigma, u_{j_0}\sigma)$.

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

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

CLEAN $(i : P \uplus \mathcal{P}; \phi; \sigma; i') \xrightarrow{\tau} (\mathcal{P}; \phi; \sigma; i')$ when $i' > i$.

Fig. 1. Semantics for processes

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* [9], and in particular for the class of simple protocols that will be introduced later on (see Definition 9) when considering the case of equivalence.

III. ASSUMPTIONS

We consider two main assumptions in our setting. First, we cannot consider arbitrary primitives. Instead, we propose a generalization of decryption-encryption rules that allows us to consider all standard primitives and a few additional ones. One advantage of this generalization is not really its expressivity (we are not much more general than the standard primitives) but its flexibility. For example, encryption can be randomized, several encryption or hash functions can be considered, etc. We could consider a (long) list of decryption-like rules but this would render the proofs unnecessarily cumbersome, with dozen of cases to be considered. Our second main assumption is the fact that protocols must be type-compliant, which intuitively guarantees that each two encrypted messages of the protocol that can be unified have the same type.

A. Shaped rewriting systems

Our main result relies on the fact that, thanks to [14], we can consider only some particular form of traces (well-typed). Hence we need to consider a similar setting. We consider rewriting rules that apply a symbol in $\Sigma_{\mathsf{d}} \uplus \Sigma_{\mathsf{test}}$ on top of constructor terms that are linear, well-sorted, and well-shaped. Moreover, we strictly control the non-linearity of the rules, and we assume the standard subterm property, as recalled below. More formally, our set $\mathcal{R}$ is divided into two parts, i.e. $\mathcal{R} = \mathcal{R}_{\mathsf{d}} \uplus \mathcal{R}_{\mathsf{test}}$, and for each symbol $\mathsf{g} \in \Sigma_{\mathsf{d}} \uplus \Sigma_{\mathsf{test}}$, we assume there is exactly one rule of the form $l \to r$ such that:

1) $l = \mathsf{g}(t_1, \ldots, t_n)$ where each t_i is either a variable or equal to $\mathsf{sh}_{\mathsf{root}(t_i)}$ up to a bijective renaming of variables;
2) either l is a linear term, or there is a unique variable x with several occurrences in l and:
   - $l = \mathsf{g}(\mathsf{f}(t_1^1, \ldots, t_1^k), t_2, \ldots, t_n)$;
   - $\{x\} \subseteq \{t_1^{j_x}, t_2, \ldots, t_n\} \subseteq \{x\} \cup \{\mathsf{f}(x) \mid \mathsf{f} \in \Sigma_{\mathsf{c}}\}$ for some j_x;
   - x occurs exactly once in $t_1^{j_x}$, in atomic position, and does not occur in the other t_1^j for $j \neq j_x$.

We denote $l_{\mathsf{g}} \to r_{\mathsf{g}}$ the rewriting rule in $\mathcal{R}$ associated to the symbol $\mathsf{g} \in \Sigma_{\mathsf{d}} \uplus \Sigma_{\mathsf{test}}$. Then, we assume that for any $\mathsf{g} \in \Sigma_{\mathsf{test}}$, the associated rule $l_{\mathsf{g}} \to r_{\mathsf{g}}$ is such that $r_{\mathsf{g}} \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \emptyset)$. In this case, $l_{\mathsf{g}} \to r_{\mathsf{g}}$ is a rule of $\mathcal{R}_{\mathsf{test}}$. When $\mathsf{g} \in \Sigma_{\mathsf{d}}$, we assume that the associated rule $\mathsf{g}(t_1, \ldots, t_n) \to r_{\mathsf{g}}$ is such that r_{g} is a direct and strict subterm of t_1, i.e. $t_1 = \mathsf{f}(t_1^1, \ldots, t_1^k)$ with $\mathsf{f} \in \Sigma_{\mathsf{c}}$, and $r_{\mathsf{g}} = t_1^{j'}$ for some $j' \in \{1, \ldots k\}$. In this case, $l_{\mathsf{g}} \to r_{\mathsf{g}}$ is a rule of $\mathcal{R}_{\mathsf{d}}$.

Lastly, we assume the existence of at least one non linear rule in $\mathcal{R}$. This last assumption is needed for the typing result stated and proved in [14], and is satisfied as soon as a rewriting rule modeling e.g. symmetric (or asymmetric) encryption is present in $\mathcal{R}$. A rewriting system satisfying our conditions is called a *shaped rewriting system*. In what follows, we only consider shaped rewriting systems.

All standard primitives such as symmetric and asymmetric encryption, signatures, mac, hash, can be modeled as shaped rewriting systems. We can also consider a few more primitives.

Example 8. *The rewriting system given in Example 3 is a shaped rewriting system. We can also consider encryption schemes where 1 out of n keys suffices to decrypt, with rewriting rules of the form:*

$$\begin{array}{lcl} \mathsf{adec}_1(\mathsf{aenc}(y, \mathsf{pk}(x), \mathsf{pk}(x')), x) & \to & y \\ \mathsf{adec}_2(\mathsf{aenc}(y, \mathsf{pk}(x'), \mathsf{pk}(x)), x) & \to & y \end{array}$$

However, adding a rewriting rule:

$$\mathsf{samekey}(\mathsf{aenc}(x_1, \mathsf{pk}(x)), \mathsf{aenc}(x_2, \mathsf{pk}(x))) \to \mathsf{ok}$$

allowing one to check whether two ciphertexts have been produced relying on the same key does not satisfy our requirements since the left-hand-side is not linear, and $\mathsf{aenc}(x_2, \mathsf{pk}(x))$ *is not of the form* $\mathsf{f}(x)$.

B. Type compliance

Intuitively, types allow us to specify the expected structure of a message.

Definition 3. *A* (structure-preserving) typing system *is a pair* $(\Delta_{\mathsf{init}}, \delta)$ *where* Δ_{init} *is a set of elements called* initial types*, and* δ *is a function mapping data in* $\Sigma_0 \uplus \mathcal{N} \uplus \mathcal{X}$ *to types* τ *generated using the following grammar:*

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

Then, δ *is extended to constructor terms as follows:*

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

Example 9. *We consider the typing system* $(\Delta_{\mathsf{DS}}, \delta_{\mathsf{DS}})$ *generated from the set* $\Delta_{\mathsf{DS}} = \{\tau_{ska}, \tau_{ekb}, \tau_{ekc}, \tau_k\}$ *of initial types, and such that:*

- $\delta_{\mathsf{DS}}(k) = \delta_{\mathsf{DS}}(k') = \delta_{\mathsf{DS}}(x) = \delta_{\mathsf{DS}}(x') = \tau_k$*, and*
- $\delta_{\mathsf{DS}}(\mathtt{xx}) = \tau_{\mathtt{xx}}$ *for* $\mathtt{xx} \in \{ska, ekb, ekc\}$.

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

In [14], protocols are defined to be type-compliant if any two unifiable encrypted subterms are of the same type. "Encrypted" means any term headed by a constructor symbol that cannot be opened freely, such as encryption. Conversely, some constructors are *transparent* in the sense that they can be opened without any extra information, such as pairs, tuples, or lists. Formally, a constructor symbol f of arity n is *transparent* if there exists a term $\mathsf{f}(R_1^{\mathsf{f}}, \ldots, R_n^{\mathsf{f}}) \in \mathcal{T}(\Sigma, \Box)$ such that for any term $t \in \mathcal{T}_0(\Sigma, \Sigma_0 \uplus \mathcal{N} \uplus \mathcal{X})$ such that $\mathsf{root}(t) = \mathsf{f}$, we have that $\mathsf{f}(R_1^{\mathsf{f}}, \ldots, R_n^{\mathsf{f}})\{\Box \to t\}\downarrow = t$.

We write $ESt(t)$ for the set of *encrypted subterms* of t, i.e. the set of subterms that are not headed by a transparent function.

$$ESt(t) = \{u \in St(t) \mid u \text{ is of the form } \mathsf{f}(u_1, \ldots, u_n) \text{ and } \mathsf{f} \text{ is not transparent}\}$$

Since replicated processes can produce several messages with a similar structure, we define the k-unfolding $\mathsf{unfold}_k(P)$ of the replicated process P as a finite version of P such that each replication has been unfolded exactly k times. For instance, we have that $\mathsf{unfold}_1(P)$ is the process obtained from P by simply removing the ! operator whereas $\mathsf{unfold}_0(P)$ is the process obtained from P by removing parts of the process under a replication.

Example 10. *The encrypted subterms occurring in the* 2*-unfolding of* P'_{DS} *are:*

- $\mathsf{pk}(ekb), \mathsf{vk}(ska)$*;*
- $\mathsf{sign}(k_i, ska), \mathsf{aenc}(\mathsf{sign}(k_i, ska), \mathsf{pk}(ekb))$*;*
- $\mathsf{sign}(x_i, ska), \mathsf{aenc}(\mathsf{sign}(x_i, ska), \mathsf{pk}(ekb))$*;*
- $\mathsf{sign}(k'_i, ska), \mathsf{aenc}(\mathsf{sign}(k'_i, ska), \mathsf{pk}(ekc))$*;*
- $\mathsf{sign}(x'_i, ska), \mathsf{aenc}(\mathsf{sign}(x'_i, ska), \mathsf{pk}(ekc))$

where indices $i \in \{1, 2\}$ *are used to distinguish names/variables coming from different unfoldings. They are given the same type:* $\delta_{\mathsf{DS}}(k_1) = \delta_{\mathsf{DS}}(k_2)$*,* etc

Definition 4. *A protocol* P *is* type-compliant *w.r.t. a typing system* $(\Delta_{\mathsf{init}}, \delta)$ *if*

- *for every* $t, t' \in ESt(\mathsf{unfold}_2(P))$ *we have that* t *and* t' *unifiable implies that* $\delta(t) = \delta(t')$.
- *for every construction* $\mathsf{match}\, x \,\mathsf{with}\, (u_1 \to P_1 \mid \ldots \mid u_j \to P_j)$ *occurring in* P*, we have that* $\delta(x) = \delta(u_1) = \ldots = \delta(u_j)$.

Example 11. *Continuing our running example, we have that* P'_{DS} *(and* P_{DS} *as well) is type-compliant w.r.t.* $(\mathcal{T}_{\mathsf{DS}}, \delta_{\mathsf{DS}})$ *given in Example 9. Indeed, since* k_i*,* k'_i*, and* x_i*,* x'_i *(with* $i \in \{1, 2\}$*) are given the same type, we have that any two unifiable encrypted subterms occurring in* $\mathsf{unfold}_2(P'_{\mathsf{DS}})$ *(those terms are listed in Example 10) have the same type.*

Compared to [14], we have generalized the definition to the match construct (see item 2 - Definition 4). We give here an example showing that it was necessary to obtain a typing result for reachability.

Example 12. *Consider the process* P *introduced in Example 5. Obviously, it is possible to reach* $\mathsf{out}(c, \mathsf{ok})$ *with the trace* $\mathsf{tr} = \mathsf{in}(c, \mathsf{f}(a))$*. Assume* $\delta(x) = \delta(y) = \delta(y') = \tau_0 \in \Delta_{\mathsf{init}}$*. As there is only one encrypted subterm in* P*, the typing system satisfies the first item of Definition 4. However,* tr *is not well-typed for* P*. More importantly, any trace that allows to reach* $\mathsf{out}(c, \mathsf{ok})$ *must unify* x *and* $\mathsf{f}(y)$*. Thus, it will only be well-typed if* $\delta(x) = \delta(\mathsf{f}(y))$*. For a (structure-preserving) typing system, it implies that* $\delta(x) = \mathsf{f}(\delta(y))$*. In particular, there is no well-typed attack trace for the typing system we have defined. Note that the condition* $\delta(x) = \mathsf{f}(\delta(y))$ *is guaranteed by the second item of Definition 4.*

Extending the result of [14], we obtain that for any type-compliant protocol, we can restrict our attention to well-typed traces. We define $\overline{\mathsf{tr}}$ obtained from tr by replacing any action $\mathsf{in}^{\alpha}(c, R)$ by $\mathsf{in}^{\alpha}(c, \_)$, any $\mathsf{out}^{\alpha}(c, \mathsf{w})$ by $\mathsf{out}^{\alpha}(c, \_)$, while $\mathsf{phase}\, i$ and $\mathsf{out}(c, c')$ actions are left unchanged. Intuitively, we only keep the type of actions, and the channels.

Theorem 1. *Let P be a protocol type-compliant w.r.t. (Δ_0, δ_0). If $P \xrightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ then there exists a well-typed execution $P \xrightarrow{\mathsf{tr}'} (\mathcal{P}; \phi'; \sigma'; i)$ such that $\overline{\mathsf{tr}'} = \overline{\mathsf{tr}}$.*

Of course, this theorem will have more effect when the type system is as precise as possible, for example distinguishing between several classes of constants or stating that some variables can be typed as atomic constants.

Example 13. *The execution corresponding to the trace given in Example 7 is well-typed. Indeed, the resulting substitution is $\sigma = \{x \mapsto k\}$ and we have that $\delta_{\mathsf{DS}}(x) = \delta_{\mathsf{DS}}(k) = \tau_k$.*

IV. Computation of a tight bound

Our main contribution is a procedure that, given a protocol, computes a (tight) over-approximation of the number of sessions that need to be considered to find an attack. More precisely, to each trace corresponds a multiset of labels. Our procedure computes (an over-approximation of) the possible multisets of actions that can occur in minimal attack traces.

A. Dependencies

Message dependencies. One key step of our procedure consists in inspecting, for each term output by the protocol, what are the type of the terms that can be deduced from it. More formally, given a type τ, we compute a set of tuples $(\tau', p)\#S$. Intuitively, each tuple $(\tau', p)\#S$ indicates that a term of type τ' may be deduced, at position p, from terms of type τ, provided the attacker knows some terms whose types are contained in S (as multiset inclusion).

Definition 5. *Given a type τ, we define $\rho(\tau)$ to be $\rho(\tau, \epsilon, \emptyset)$ where $\rho(\tau, p, S)$ is recursively defined as the set $\{(\tau, p)\#S\} \cup E$ where $E = \emptyset$ when τ is an initial type. Otherwise, $\tau = \mathsf{f}(\tau_1, \ldots, \tau_k)$ and E is defined as:*

$$E = \bigcup_{\substack{\mathsf{g}(\ell_1, \ldots, \ell_n) \to r_{\mathsf{g}} \in \mathcal{R}_{\mathsf{d}} \\ \theta = mgu(\ell_1, \mathsf{f}(\tau_1, \ldots, \tau_k)) \\ i_0 \in \{1, \ldots, k\} \text{ is such that } r_{\mathsf{g}} = \ell_1|_{i_0}}} \rho(\tau_{i_0}, p.i_0, S \uplus \{\ell_2\theta, \ldots, \ell_n\theta\})$$

Example 14. *Let $\tau_{\mathsf{msg}} \stackrel{\mathsf{def}}{=} \mathsf{aenc}(\mathsf{sign}(\tau_k, \tau_{ska}), \mathsf{pk}(\tau_{ekb}))$. First the element $(\tau_{\mathsf{msg}}, \epsilon)\#\emptyset$ is in $\rho(\tau_{\mathsf{msg}})$, and we are left to compute $\rho(\mathsf{sign}(\tau_k, \tau_{ska}), 1, \{\tau_{ekb}\})$ since the rule $\mathsf{adec}(\mathsf{aenc}(y, \mathsf{pk}(x)), x) \to y$ is the only one that can be applied to extract a message from a ciphertext. Then, we have that:*

$$\rho(\mathsf{sign}(\tau_k, \tau_{ska}), 1, \{\tau_{ekb}\}) = \left\{ \begin{matrix} (\mathsf{sign}(\tau_k, \tau_{ska}), 1)\#\{\tau_{ekb}\} \\ (\tau_k, 1.1)\#\{\tau_{ekb}\} \end{matrix} \right\}$$

This last element represents the fact that a message of type τ_k can be extracted from a signature at position 1 using the rule $\mathsf{getmsg}(\mathsf{sign}(x, y)) \to x$, and this does not require additional knowledge. The set $\rho(\tau_{\mathsf{msg}})$ contains 3 elements.

In order to compute a tighter bound, we will sometimes skip the computation of some dependencies, for some *marked position*. A *marked position* of a protocol P w.r.t. a typing system (Δ_0, δ_0) is a pair (α, p) where $\mathsf{out}^\alpha(c, u)$ is an output action occurring in P, and p is a position of the term $\delta_0(u)$. For the rest of this section, we will assume given a set of marked position and we will explain later on how to soundly mark positions. By default, the reader may simply assume that no position is marked.

Sequential dependencies. Another, simpler, type of dependencies is sequential dependency: some action may occur only if the previous steps of the same process have been executed. We let $\mathsf{pred}(\alpha)$ be the first visible action that occurs before the action labeled by α. More formally, a process P can be seen as a tree whose nodes are actions $\mathsf{in}(c, u)$, $\mathsf{out}(c, u)$, $\mathsf{new}\, n$, $|$, etc, and vertices are there to indicate the continuation of the process. For instance, a node labeled with the action "$\mathsf{match}\ x\ \mathsf{with}$" will have j sons representing the j branches of the match construct. Then, given an action of the form $\mathsf{in}^\alpha(c, u)$ (resp. $\mathsf{out}^\alpha(c, u')$), we denote $\mathsf{pred}(\alpha)$ its first predecessor that corresponds to an input/output of a message (not a channel name). We have that $\mathsf{pred}(\alpha) = \bot$ if there is no such predecessor in the tree.

We also introduce the notion of *cv-alien types*, that are types that do not appear as type of the constants and variables in the protocol under study.

Definition 6. *Consider a protocol P and a typing system (Δ_0, δ_0). A type τ_h is* cv-alien *for P if τ_h is a type alien w.r.t. the constants and variables of P, that is, $\tau_h \neq \delta_0(a)$ for any constant/variable $a \in \Sigma_0 \cup \mathcal{X}$ occurring in P.*

We say that a term is cv-alien-free *if it does not contain any constant from Σ_0 of cv-alien type. This notion is lifted to traces, frames, configurations and executions as expected.*

We will show that the attacker may only use cv-alien-free terms since it is not useful to introduce constants whose type does not appear in the protocol.

Example 15. *Continuing our running example, we have that τ_{ska} and τ_{ekb} are cv-alien type. Actually, we have that any type but τ_k and τ_{ekc} is a cv-alien type. Regarding τ_k, this comes from the fact that $\delta_{\mathsf{DS}}(x) = \delta_{\mathsf{DS}}(x') = \tau_k$ (see Example 9).*

B. Main procedure

We assume given a protocol P type-compliant w.r.t. a typing system. We propose a procedure, denoted dep, that, given a label α occurring in P, computes $\mathsf{dep}(\alpha)$, a set of multisets of labels. Each multiset of labels represents the sessions that may be needed to reach label α.

We first introduce the operation $\otimes$ to compute some kind of "cartesian product" on two sets of multisets. The result is not a pair of multisets but instead we merge the two components of each pair to obtain a multiset. Formally,

$$\begin{aligned} \{a_1, \ldots, a_k\} \otimes \{b_1, \ldots, b_l\} \stackrel{\mathsf{def}}{=} \\ \big\{ & a_1 \uplus b_1, a_1 \uplus b_2, \ldots, a_1 \uplus b_l, \\ & a_2 \uplus b_1, a_2 \uplus b_2, \ldots, a_2 \uplus b_l, \\ & \vdots \\ & a_k \uplus b_1, a_k \uplus b_2, \ldots, a_k \uplus b_l \big\} \end{aligned}$$

where $a_1, \ldots, a_k, b_1, \ldots, b_l$ are multisets of labels. Note that given a multiset set S, we have that: $S \otimes \{\emptyset\} = \{\emptyset\} \otimes S = S$, and $S \otimes \emptyset = \emptyset \otimes S = \emptyset$.

We define inductively a family of functions dep^i on labels and types as follows: $\mathsf{dep}^i(\bot) = \{\emptyset\}$; and

If α is an output, $i > 0$,

- $\mathsf{dep}^0(\alpha) = \emptyset$,
- $\mathsf{dep}^i(\alpha) = \{\{\alpha\}\} \otimes \mathsf{dep}^{i-1}(\mathsf{pred}(\alpha))$.

If α is an input of type τ, $i > 0$,

- $\mathsf{dep}^0(\alpha) = \emptyset$,
- $\mathsf{dep}^i(\alpha) = \{\{\alpha\}\} \otimes \mathsf{dep}^{i-1}(\mathsf{pred}(\alpha)) \otimes \mathsf{dep}^{i-1}(\tau)$.

The definition of dep^i is extended as expected to multisets:

$$\mathsf{dep}^i(S) = \bigcup_{\alpha \in S} \mathsf{dep}^i(\alpha)$$

When dep^i is applied to a type, we need a family of auxiliary functions S^i_{out}, inductively defined as follows, $i \geq 0$:

$$S^i_{\mathsf{out}}(\tau) = \bigcup_{\substack{\mathsf{out}^\alpha(c,u) \text{ occurring in } P \\ (\tau,p)\#\{\tau_1,\ldots,\tau_n\} \in \rho(\delta_0(u)) \\ (\alpha,p) \text{ not marked}}} \left(\mathsf{dep}^i(\alpha) \otimes \mathsf{dep}^i(\tau_1) \otimes \ldots \otimes \mathsf{dep}^i(\tau_n)\right)$$

Intuitively, $S^i_{\mathsf{out}}(\tau)$ explores all the possibilities to extract a term of type τ. Then, we define $\mathsf{dep}^0(\tau) = \emptyset$ when τ is a cv-alien type; $\mathsf{dep}^0(\tau) = \{\emptyset\}$ otherwise. The reason is that if there is an attack trace, then there is one which is well-typed and which does not involve any constant of cv-alien type. Thus, there is no need to consider this case when exploring all the possibilities to build a term having such a type, and thus $\mathsf{dep}^0(\tau) = \emptyset$. Otherwise, we have to consider the case where the term of type τ is a constant known by the attacker, and this does not lead to further dependencies, thus $\mathsf{dep}^0(\tau) = \{\emptyset\}$. Finally, for any $i > 0$, we let:

- $\mathsf{dep}^i(\tau) = \mathsf{dep}^0(\tau) \cup S^{i-1}_{\mathsf{out}}(\tau)$ for an initial type τ,
- $\mathsf{dep}^i(\tau) = \mathsf{dep}^0(\tau) \cup S^{i-1}_{\mathsf{out}}(\tau) \cup \left(\{\emptyset\} \otimes \mathsf{dep}^{i-1}(\tau_1) \otimes \ldots \otimes \mathsf{dep}^{i-1}(\tau_k)\right)$ for a non-initial type τ of the form $\mathsf{f}(\tau_1, \ldots, \tau_k)$.

Note that a set $\mathsf{dep}^i(\alpha)$ can only increase with i. More formally, for any type τ and any label α, we have that:

$$\mathsf{dep}^i(\tau) \subseteq \mathsf{dep}^{i+1}(\tau) \qquad \mathsf{dep}^i(\alpha) \subseteq \mathsf{dep}^{i+1}(\alpha)$$

$$S^i_{\mathsf{out}}(\tau) \subseteq S^{i+1}_{\mathsf{out}}(\tau)$$

since $A_1 \otimes \cdots \otimes A_n \subseteq B_1 \otimes \cdots \otimes B_n$ when $A_i \subseteq B_i$.

Hence we define $\mathsf{dep}(\tau)$ (resp. $S_{\mathsf{out}}(\tau)$) as the limit of the $\mathsf{dep}^i(\tau)$ (resp. $S^i_{\mathsf{out}}(\tau)$), that is,

$$\mathsf{dep}(\tau) = \bigcup_{i=0}^{\infty} \mathsf{dep}^i(\tau) \qquad S_{\mathsf{out}}(\tau) = \bigcup_{i=0}^{\infty} S^i_{\mathsf{out}}(\tau)$$

We see the interest of cv-alien type in the definition of dep^0: $\mathsf{dep}^0(\tau) = \emptyset$ if τ is a cv-alien type. Hence, if no term of type τ can be extracted from the outputs of the protocol (i.e. $S_{\mathsf{out}}(\tau) = \emptyset$), we have that $\mathsf{dep}(\tau) = \emptyset$, and simplifications arise since $S \otimes \emptyset = \emptyset$.

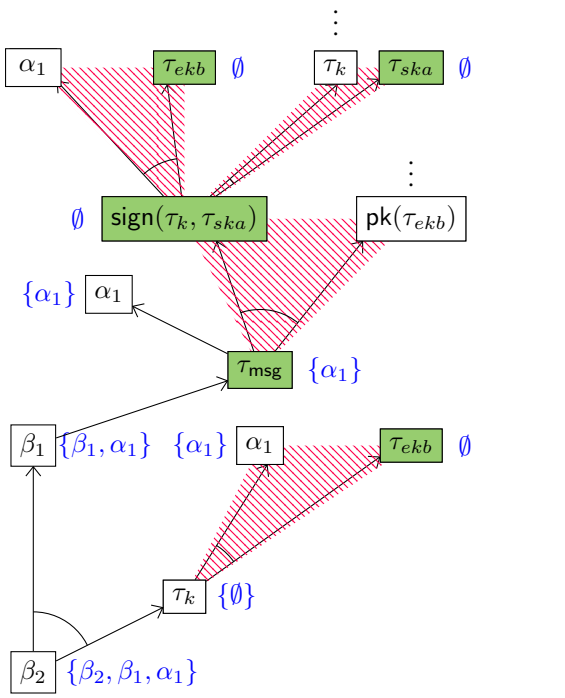

Fig. 2. Computation of $\mathsf{dep}(\beta_2)$.

Example 16. *Continuing our running example, we may want to consider the secrecy of the key k as received by B. To encode this property, we add an action at the end of process P_B, that checks whether the key can transit in clear on the network, yielding process*

$$P_B^+ = \mathsf{in}^{\beta_1}(c, \mathsf{aenc}(\mathsf{sign}(x, ska), \mathsf{pk}(ekb))).\mathsf{in}^{\beta_2}(c, x).$$

We modify P_{DS} accordingly (with P_B^+), yielding P_{DS}^+. Note that, even if P_{DS}^+ does not involve the dishonest agent c, the protocol features replication, and thus the question of deciding whether an action labelled β_2 is reachable is not trivial.

In order to compute $\mathsf{dep}(\beta_2)$, we first remark that τ_{ska} and τ_{ekb} are cv-alien types, as explained in Example 15. Therefore, since $S_{\mathsf{out}}(\tau_{ska}) = S_{\mathsf{out}}(\tau_{ekb}) = \emptyset$, we have that $\mathsf{dep}(\tau_{ska}) = \mathsf{dep}(\tau_{ekb}) = \emptyset$. Then, we have that $S_{\mathsf{out}}(\mathsf{sign}(\tau_k, \tau_{ska})) = \emptyset$, and thus $\mathsf{dep}(\mathsf{sign}(\tau_k, \tau_{ska})) = \emptyset$.

We now follow the dep *algorithm step by step, as illustrated*

in Figure 2.

$$\begin{array}{rcl}\mathsf{dep}(\beta_2) & = & \{\{\beta_2\}\} \otimes \mathsf{dep}(\beta_1) \otimes \mathsf{dep}(\tau_k) \\ & = & \{\{\beta_2, \beta_1\}\} \otimes \mathsf{dep}(\tau_{\mathsf{msg}}) \otimes \mathsf{dep}(\tau_k)\end{array}$$

Actually $S_{\mathsf{out}}(\tau_k) = \emptyset$*, and thus* $\mathsf{dep}(\tau_k) = \{\emptyset\}$*. We have seen in Example 15 that* τ_k *is not of cv-alien type, and thus* $\mathsf{dep}^0(\tau_k) = \{\emptyset\}$*.*

We have also that:

$$\begin{array}{rcl}\mathsf{dep}(\tau_{\mathsf{msg}}) & = & \{\emptyset\} \otimes \mathsf{dep}(\mathsf{sign}(\tau_k, \tau_{ska})) \otimes \mathsf{dep}(\mathsf{pk}(\tau_{ekb})) \\ & & \cup\, S_{\mathsf{out}}(\tau_{\mathsf{msg}}).\end{array}$$

To conclude, it is sufficient to see that:

- $S_{\mathsf{out}}(\tau_{\mathsf{msg}}) = \mathsf{dep}(\alpha_1) = \{\{\alpha_1\}\}$*; and*
- $\begin{array}{rcl}\mathsf{dep}(\mathsf{sign}(\tau_k, \tau_{ska})) & = & \{\emptyset\} \otimes \mathsf{dep}(\tau_k) \otimes \mathsf{dep}(\tau_{ska}) \\ & & \cup\, S_{\mathsf{out}}(\mathsf{sign}(\tau_k, \tau_{ska})) \\ & = & \emptyset\end{array}$

Hence, finally, we have that $\mathsf{dep}(\beta_2) = \{\{\beta_2, \beta_1, \alpha_1\}\}$*.*

C. Correctness of the bound

We denote $\mathsf{Label}(\mathsf{tr})$ the multiset of labels (from $\mathcal{L}$) occurring in tr. The algorithm $\mathsf{dep}(\alpha)$ computes an upper bound of the actions/labels that need to be considered to reach some action labeled α.

Theorem 2. *Let* P *be a protocol type-compliant w.r.t. some typing system* (Δ_0, δ_0)*. Let* α *be a label of* P *and assume* $(\mathsf{tr}.\ell, \phi) \in \mathsf{trace}(P)$ *for some* tr*,* ℓ*,* ϕ *such that* $\mathsf{Label}(\ell) = \{\alpha\}$*. Then there exists* tr', ℓ'*,* ϕ'*, and* $A \in \mathsf{dep}(\alpha)$ *such that* $(\mathsf{tr}'.\ell', \phi') \in \mathsf{trace}(P)$ *with* $\mathsf{Label}(\ell') = \{\alpha\}$*; and* $\mathsf{Label}(\mathsf{tr}'.\ell') \subseteq A$*.*

This theorem shows that it is sufficient to consider traces with labels in $\mathsf{dep}(\alpha)$ to access an action labeled by α. Secrecy can easily be encoded with such an accessibility property, introducing some special action α_0 that can be reached only if the secret is known to the attacker.

Example 17. *Continuing our running example, and thanks to Theorem 2, it is sufficient to consider one instance of* P_A*, and one instance of* P_B^+ *when looking for an attack on the secrecy in* P_{DS}^+*. In particular, no need to unfold replications more than once, and no need to consider process* P_K*.*

Let us now consider a modified process P'_{DS}*, that now includes a session between the server, the initiator, and a dishonest agent* c*. After applying* dep *recursively, we get:*

$$\mathsf{dep}(\beta_2) = \left\{ \begin{array}{l} \{\beta_2, \beta_1, \alpha_1, \alpha'_1\}, \\ \{\beta_2, \beta_1, \alpha'_1, \alpha'_1, \gamma_1\} \end{array} \right\}$$

In other words, to analyze secrecy of k *in this richer scenario, we can restrict ourselves to two simple scenarios. The first one requires only one instance of the roles* P_B^+*,* P_A *and* P'_A*. The second one involves half of the process* P_K *(only the first action can be triggered), one instance of* P_B^+*, and two instances of* P'_A *(initiator role played by* a *with the dishonest agent* c*).*

We prove our Theorem 2 in two main steps.

(i) We first rely on type-compliance and the typing result given in [14], extended here to deal with processes with match construct. As stated in Theorem 1, this allows us to restrict our attention to well-typed traces. Actually, we further show that traces can be assumed to be cv-alien-free, and to only involve simple recipes (some constructors are applied on top of recipes that are almost destructor-only)[1]. Lastly, we show that each message of such traces can be computed as soon as possible (asap). Intuitively, recipes should refer to the earliest occurrence of a message. More formally, we have that:

Definition 7. *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'*, we have* $R < R'$ *if, and only if,* $vars^{\#}(R) <_{\mathsf{mul}} vars^{\#}(R')$*, where* $vars^{\#}(R)$ *denotes the multiset of variables occurring in* R*, and* $<_{\mathsf{mul}}$ *is the multiset extension of* $<$*.*

(ii) Second, our procedure dep is used to consider all the possible ways of deducing a term of a certain type τ (or reaching a specific action α). This is done by considering all the sequential dependencies, and all the message dependencies. For this, we heavily rely on the fact that our witness only involves simple recipes. We thus know the shape of these recipes, and compliance with types also imposes us some restrictions that are exploited in our procedure.

D. Marking criteria

We mainly consider two marking criteria. First, we mark any position p occurring in an output $\mathsf{out}(c, u)$ such that $\delta_0(u)|_p$ is a public type.

Definition 8. *Given a 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* $\delta_0(n) \notin St(\tau_p)$*.*

Actually, a public type is a type for which all the terms of this type are known by the attacker from the beginning (in a well-typed cv-alien-free execution). Thus, we can safely ignore those terms when computing dep.

Second, we also mark any position p occurring in an output $\mathsf{out}(c, u)$ when an occurrence of $u|_p$ already occurs in the process (before the output under consideration) and it was less protected. The intuition is that an attacker who will try to deduce each term as soon as possible, will never use this output $\mathsf{out}(c, u)$ to extract $u|_p$. We illustrate this second criterion through an example.

Example 18. *Consider the following process:*

$$P = \mathsf{in}^{\alpha}(c, \mathsf{senc}(\langle \mathsf{req}, x\rangle, k)).\mathsf{out}^{\beta}(c, \mathsf{senc}(\langle \mathsf{rep}, x\rangle, k)).$$

We can safely mark $(\beta, 1.2)$ *corresponding to the term* x *which is protected by* k*.*

[1] A formal defintion is given in Appendix

We formally show that our two marking criteria are sound, namely that we can safely ignore marked positions in dep. In other words, we show that Theorem 2 still holds with these two criteria.

V. Extension to equivalence properties

We can also bound the number of sessions in case of equivalence properties. We however consider a more restricted class of protocols, without the match construct (hence without else branches) and with a *simple structure*: each process emits on a distinct channel. This corresponds to the case, common in practice, where sessions can be identified, for example with session identifiers. More formally, we consider the class of simple protocols.

Definition 9. *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 the channel names $c_1, \ldots, c_n, c_{n+1}, \ldots, c_{n+m}$ are pairwise distinct, and 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:

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

where $u \in \mathcal{T}_0(\Sigma_c, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$, $\alpha \in \mathcal{L}$, and $j \in \mathbb{N}$.

Note that our definition of simple protocol assumes a fresh, distinct channel for each session. In particular, two sessions of the same process will use different channels. Hence our model does not assume that the attacker can identify an agent across different sessions, this will depend on the protocol. Therefore simple processes can still be used to model anonymity or unlinkability properties.

A. Our procedure

The computation of dep for reachability properties no longer suffices for equivalence properties. Indeed, the attacker not only may need several sessions to reach some interesting step of the protocol, but may also need to deduce auxiliary information to mount a test that allows her to distinguish between two protocols. Hence, the computation of dep will now also depend on the rewriting rules in $\mathcal{R}_{\mathsf{test}}$.

Formally, we keep our definition of dep on labels and types and we extend it to protocols. We define:

$$\mathsf{dep}(P) = \{\emptyset\} \cup S_{\mathsf{reach}}(P) \cup S_{\mathsf{test}}(P) \cup S_{\mathsf{check}}(P)$$

where S_{reach}, S_{test}, and S_{check} are given in Figure 3.

Intuitively, $\mathsf{dep}(P)$ explores the different cases where trace inclusion of P in some protocol Q may fail. The first case is when some action can be reached in P but not in Q. The corresponding sets of labels are computed by $S_{\mathsf{reach}}(P)$. A second case is when some equality holds in P and not in Q. We rely here on a precise characterization of static inclusion, where we show that it is possible to consider tests of the form $M = N$ where N only contains destructors. This case is explored by $S_{\mathsf{test}}(P)$ where the possible types of a destructor-only recipe is computed by S^+_{out}. Finally, the last relevant case is when a term can be reduced in P (according to the equational theory) and not in Q. This corresponds to S_{check}. Note that S_{out} coincides with S^+_{out} except that marked positions (α, p) are no longer ignored when $p = \epsilon$. The reason is that even if a position is marked and hence the corresponding term could be obtained earlier, it may be necessary to consider the position in order to check its equality with an earlier term. In other words, when looking for a test $M = N$, we can no longer consider that both M, and N are asap recipes. However, we have shown that we can safely consider that at least one of them is asap, whereas the other one can be assumed to be subterm asap, meaning that all its direct subterms are asap (but not necessarily the term itself).

$$S_{\mathsf{reach}}(P) = \bigcup_{\alpha \in \mathsf{Label}(P)} \mathsf{dep}(\alpha)$$

$$S_{\mathsf{test}}(P) = \bigcup_{\tau \in St(\delta_0(P))} \mathsf{dep}(\tau) \otimes S^+_{\mathsf{out}}(\tau)$$

$$S_{\mathsf{check}}(P) = \bigcup_{\substack{\tau \in St(\delta_0(P)) \\ \ell = \mathsf{g}(t_1, \ldots, t_n) \to r \\ \theta = mgu(t_1, \tau)}} S_{\mathsf{out}}(\tau) \otimes \mathsf{dep}(t_2\theta) \otimes \ldots \otimes \mathsf{dep}(t_n\theta)$$

$$S^+_{\mathsf{out}}(\tau) = \bigcup_{\substack{\mathsf{out}^\alpha(c,u) \text{ occurring in } P \\ (\tau, p)\#\{\tau_1, \ldots, \tau_n\} \in \rho(\delta_0(u)) \\ (\alpha, p) \text{ not marked or } p = \epsilon}} \mathsf{dep}(\alpha) \otimes \mathsf{dep}(\tau_1) \otimes \ldots \otimes \mathsf{dep}(\tau_n)$$

Fig. 3. Definitions of S_{reach}, S_{test}, and S_{check}.

B. Correctness

When searching for an attack against an equivalence property specified as $P \approx_t Q$, it is sufficient to consider sessions as prescribed by $\mathsf{dep}(P)$ and $\mathsf{dep}(Q)$.

Theorem 3. *Let P be a* simple *protocol type-compliant w.r.t. some typing system (Δ_0, δ_0). Let Q be another* simple *protocol such that $P \not\sqsubseteq_t Q$. There exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ witnessing this non-inclusion such that $\mathsf{Label}(\mathsf{tr}) \subseteq A$ for some $A \in \mathsf{dep}(P)$.*

The proof of this theorem follows the same lines as the one for reachability. Additional difficulties arise due to the fact that we also need to provide a bound regarding static inclusion. Considering an equality test $R_1 = R_2$ that witnesses non static inclusion, we show that R_1 and R_2 can be chosen with a specific shape that allows one to precisely characterize the actions that may be involved in such a test.

VI. Experiments

We have implemented our procedure into a tool HowMany, that we run on several protocols of the literature, studying both reachability and equivalence properties. When the tool

returns a list of (finite) scenarios, we can then use two existing tools, SAT-Equiv and DeepSec, developed for a bounded number of sessions, and directly conclude that security holds in the unbounded case (unless the tools find an attack). The specifications of all protocols, as well as the files to reproduce the experiments, can be found in [20].

A. HowMany

The function dep may return infinite sets, hence does not directly yield a terminating algorithm. Therefore, we define dep', a terminating algorithm that returns the same result than dep whenever it is finite, and returns $\bot$ otherwise.

The main idea is to first decide whether a given type τ (resp. label α) is such that $\mathsf{dep}(\tau) = \emptyset$. In this case, since $\emptyset$ is an absorbing element w.r.t. $\otimes$, we may conclude that, e.g.

$$\mathsf{dep}(\tau) \otimes \mathsf{dep}(\tau_1) \otimes \ldots \otimes \mathsf{dep}(\tau_n) = \emptyset$$

without computing $\mathsf{dep}(\tau_1), \ldots, \mathsf{dep}(\tau_n)$. Once empty elements have been identified, relations can be simplified, and it is relatively easy to identify a loop and to return $\bot$ to indicate that one of the resulting multisets will be infinite.

The algorithm dep' has been implemented in the tool HowMany. It will either return $\bot$ or a set of multisets of labels. Each element in the set corresponds to a finite scenario that needs to be analyzed. Thanks to Theorems 2 and 3, we can conclude that the protocol is secure if it is secure in all the scenarios identified by HowMany. A multiset of labels tells us the maximal number of sessions that may be involved in a minimal attack. Actually, it is even more precise than that since our algorithm gives us a list of scenarios, and each scenario corresponds to a precise number of unfolding of a replication, possibly truncated.

B. Security properties

We have considered three types of security properties, depending on the protocol under study.

The first one is *weak secrecy* (WSEC), a reachability property. We always consider secrecy of the key (sometimes the nonce) as received by the responder. This is done by adding an instruction of the form $\mathsf{in}^{\alpha}(c, k)$ at the end of the responder role. Then, we ask for reachability of the label α.

The second one is *key privacy* (KPRIV). Intuitively, a key k is secure if an attacker cannot learn any information on messages that are encrypted by k. We model this by adding at the end of the responder' role $\mathsf{senc}(\mathsf{m}_1, k)$ on the left, and $\mathsf{senc}(\mathsf{m}_2, k')$ on the right, where k' is fresh, and m_1, m_2 are two public constants.

For these two security properties, we consider a process where each role is instantiated (arbitrarily many times) by all possible players among 2 honest agents and a dishonest one.

Lastly, we analyzed some protocols from the e-passport application, and we consider the unlinkability property (UNLINK). This property is modeled relying on phases. In a first phase, the attacker interacts with two passports and two readers possibly many times. In a second phase, the attacker interacts with either one instance of the first passport (and a reader) or the second one. The protocol is linkable if the attacker is able to distinguish between the two cases.

C. Outcome of Howmany

HowMany computes a set of scenarios, and each scenario involves several sessions of the protocol. When more than one scenario is returned, HowMany also computes a *unique* scenario that over-approximates all the other ones. Indeed, there is trade-off between considering multiple simple scenarios or a unique but more complex one corresponding to the over-approximation of all the simple ones. Depending on the tool and the protocol, one approach may be more efficient than the other, hence we consider both cases (multiple and unique scenarios).

On all the examples mentioned in this section, HowMany concludes in few seconds on a standard laptop but the Kao-Chow example for KPRIV, which takes around 2 min. The detailed outcome of our experiments is displayed in Table I and Table II and we further comment them below. Compared with [21], this shows a significant improvement. For the simple Denning-Sacco protocol, in the case of reachability, [21] yields a bound of at least 10^{19} and the bound would be even larger in the other cases.

Reachability. The results regarding the weak secrecy property WSEC are reported in the right part of Table I. The table can be read as follows. For instance, for the Yahalom-Paulson protocol, HowMany states that we may either consider 25 scenarios among which the biggest one is made up of 19 sessions in parallel and 35 actions in total; or we can decide to analyze directly the more complex one which features 30 sessions in parallel for a total of 56 actions.

Equivalence. Actually, when analyzing KPRIV, we may restrict our attention to consider one inclusion. Indeed, in case $P \not\approx_t Q$, we necessarily have that $P \not\sqsubseteq_t Q$ since there are more equalities on P side. To study this inclusion, we only need to compute $\mathsf{dep}(P)$. Regarding the property UNLINK, we focus again on one inclusion due to the symmetry of the relation under study. Moreover, we consider a simplified variant of the protocols, without else branches, since else branches are not supported by our approach.

Our experiments show that for all these protocols, a small number of sessions is sufficient, up to 55 sessions in parallel for the Kao-Chow protocol if one wishes to analyze a unique (big) scenario only. The only cases where HowMany cannot conclude are the Yahalom-Lowe protocol and the (flawed) Needham-Schroeder protocol. For several protocols such as Denning-Sacco or Wide-Mouth Frog, we even retrieve that 2-3 sessions are sufficient, which corresponds to a very simple scenario where one honest instance of each role is considered.

D. SAT-Equiv, DeepSec

Even if our first and main contribution is theoretical, our result shows that it is possible to find a reasonable bound on the number of sessions on several protocols of the literature. Actually, thanks to the recent advances of the verification

	Reachability (WSEC)							**Equivalence** (KPRIV)						
	HowMany			**SAT-Equiv**		**DeepSec**		**HowMany**			**SAT-Equiv**		**DeepSec**	
	nb	size		mult.	unique	mult.	unique	nb	size		mult.	unique	mult.	unique
Symmetric protocols														
Denning-Sacco	1	3 (8)		<1s		<1s		5	5 (12)	14 (31)	<1s	<1s	<1s	<1s
Needham-Schroeder	16	20 (45)	28 (63)	12s	5s	32s	18m	83	33 (72)	47 (107)	6m	1m	TO	TO
Otway-Rees⋆	4	12 (20)	16 (28)	2s	1s	< 1s	1s	22	15 (23)	27 (48)	8s	7s	1s	48m
Wide-Mouth-Frog⋆	1	3 (6)		<1s		<1s		4	5 (8)	12 (20)	<1s	<1s	<1s	<1s
Kao-Chow (variant)⋆	48	15 (27)	28 (47)	4m	1m	3s	2m	385	29 (48)	55 (91)	10h	2h	TO	TO
Yahalom-Paulson⋆	25	19 (35)	30 (56)	2m	44s	4h	TO	147	29 (50)	45 (85)	45m	8m	TO	TO
Yahalom-Lowe⋆	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Asymmetric protocols														
Denning-Sacco	1	2 (4)		<1s		<1s		5	3 (4)	8 (11)	<1s	<1s	<1s	<1s
Needham-Schroeder⋆	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NS-Lowe⋆	2	7 (16)	8 (18)	<1s	<1s	< 1s	<1s	20	9 (19)	14 (31)	<1s	<1s	<1s	<1s

⋆: the tagged version of the protocol has been considered to ensure type-compliance.
nb columns: number of scenarios returned by HowMany.
"size" columns (white): maximal number of sessions involved in the multiple scenarios, and in parentheses, the number of protocol actions.
"size" columns (gray): idem but for the unique, aggregated scenario given by HowMany.

mult.: time for the analysis of all the multiple scenarios. unique: time for the analysis of the unique, aggregated scenario. TO: time out (>24h).

TABLE I
ANALYSIS FOR WSEC AND KPRIV

	Equivalence (UNLINK)				
	HowMany			**SAT-Equiv**	
	nb	size		mult.	unique
BAC	23	12 (34)	26 (76)	10s	5s
PA	6	7 (10)	31 (45)	<1s	<1s
AA	6	7 (10)	31 (46)	<1s	<1s

Columns are organized as explained in Table I. Since DeepSec does not handle phases, we could not use it for these protocols.

TABLE II
ANALYSIS FOR UNLINK

tools dedicated to a bounded number of sessions, our approach extends the scope of existing tools such as Avispa [4], DeepSec [12], or SAT-Equiv [19] to an unbounded number of sessions. We consider two of them, namely DeepSec and SAT-Equiv, based on different verification techniques. We selected them because they are known for their efficiency, they are suitable for the class of protocols we consider in this paper, and they are able to deal with both reachability and equivalence properties.

DeepSec. DeepSec is based on constraint solving. As any other tool based on this approach, it suffers from a combinatorial explosion when the number of sessions increases. To tackle this issue partial-order reductions (POR) techniques that eliminate redundant interleavings have been implemented and provide a significant speedup. This is only possible for the class of determinate processes, but this assumption is actually satisfied by our case studies.

SAT-Equiv. SAT-Equiv proceeds by reduction to planning problem and SAT-formula, and is quite efficient on the specific class of protocols that it handles. This class is similar to the one considered in this paper. This approach is less impacted by the state-space explosion problem when the number of sessions increases.

We used both tools to analyze all the scenarios returned by HowMany. Even if parallelism is available in DeepSec, we do not rely on this feature, and we consider a timeout of 24h. SAT-Equiv concludes on all the scenarios and is more efficient when analyzing the unique, aggregated scenario than all the small ones. Regarding DeepSec, when more than 40 sessions are involved, the tool is not able to conclude within 24h. It is interesting to note that, contrary to SAT-Equiv, DeepSec is more efficient when analyzing many small scenarios rather than the unique aggregated one.

VII. CONCLUSION

We have proposed an algorithm that soundly bounds the number of sessions needed for an attack, both for reachability and equivalence properties. This provides some insights on why, in practice, attacks require only a small number of sessions. Note that $\mathsf{dep}(P)$ may potentially be infinite. In that case, our theorem does not provide any concrete bound but may be of theoretical interest.

In our experiments, we have assumed a finite number of agents (two honest agents and a dishonest one). Agents can soundly be bounded for reachability properties [17] and equivalence [18] when there is no else branches. Alternatively, an additional process can be considered in the model, that creates agents and keys, and distributes them to the other roles. It then remains to check whether HowMany can still bound the number of sessions in this setting. Further experiments would need to be conducted.

We could extend our approach to deal with correspondence properties, for example simple authentication properties of the form $\mathsf{end}(x) \rightarrow \mathsf{start}(x)$. We could indeed exploit our extension of the typing result that preserves a certain number of disequalities. An other interesting direction for future work could be to extend our class of protocols and to consider e.g. private channels. This requires first to extend the underlying typing result but seems to be doable. Regarding reachability

properties, our dep algorithm will require some small adaptations. The case of equivalence properties appears to be more difficult since the addition of private channels takes us away from the class of simple protocols.

While HowMany provides a small bound in many cases (smaller than 15 sessions), the bound can be quite big in some cases. We plan to further refine our bound by identifying spurious scenarios, for example exploiting further dependencies in the case of phases. Lastly, there is a trade-off between analyzing many simple scenarios and a big one. Whereas it makes sense to put together simple scenarios when the overlap is important, we should not gather scenarios that are almost disjoint. Providing a clever way to group scenarios could simplify the security analysis.

REFERENCES

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. *ACM Sigplan Notices*, 36(3):104–115, 2001.

[2] M. Abadi and C. Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *Journal of the ACM*, 65(1):1–41, 2017.

[3] M. Arapinis and M. Duflot. Bounding messages for free in security protocols. In *Foundations of Software Technology and Theoretical Computer Science (FST&TCS'07), New Delhi, India*, volume 4855 of *LNCS*, pages 376–387. Springer, 2007.

[4] 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 *Proc. 17th International Conference on Computer Aided Verification, (CAV'05)*, volume 3576 of *LNCS*, pages 281–285. Springer, 2005.

[5] M. Baudet. Deciding security of protocols against off-line guessing attacks. In *Proc. 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 16–25, Alexandria, Virginia, USA, Nov. 2005. ACM Press.

[6] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *Proc. Symposium on Security and Privacy, (S&P'14)*, pages 98–113. IEEE Computer Society, 2014.

[7] B. Blanchet. An automatic security protocol verifier based on resolution theorem proving (invited tutorial). In *Proc. 20th International Conference on Automated Deduction (CADE'05)*, Tallinn, Estonia, July 2005.

[8] B. Blanchet. *Vérification automatique de protocoles cryptographiques : modèle formel et modèle calculatoire.* Mémoire d'habilitation à diriger des recherches, Université Paris-Dauphine, Nov. 2008.

[9] R. Chadha, Ş. Ciobâcă, and S. Kremer. Automated verification of equivalence properties of cryptographic protocols. In *Proc. 21st European Symposium on Programming Languages and Systems (ESOP'12)*, volume 7211 of *LNCS*, pages 108–127, Tallinn, Estonia, 2012. Springer.

[10] V. Cheval, H. Comon-Lundh, and S. Delaune. Trace equivalence decision: Negative tests and non-determinism. In *Proc. 18th ACM Conference on Computer and Communications Security (CCS'11)*, Chicago, Illinois, USA, Oct. 2011. ACM Press.

[11] V. Cheval, V. Cortier, and S. Delaune. Deciding equivalence-based properties using constraint solving. *Theoretical Computer Science*, 492:1–39, June 2013.

[12] V. Cheval, S. Kremer, and I. Rakotonirina. Deepsec: deciding equivalence properties in security protocols theory and practice. In *2018 Symposium on Security and Privacy (S&P'18)*, pages 529–546. IEEE, 2018.

[13] T. Chothia and V. Smirnov. A traceability attack against e-passports. In *14th International Conference on Financial Cryptography and Data Security (FC'10)*, 2010.

[14] R. Chrétien, V. Cortier, A. Dallon, and S. Delaune. Typing messages for free in security protocols. *ACM Transactions On Computational Logic (TOCL)*, 21(1):1–52, 2019.

[15] R. Chrétien, V. Cortier, and S. Delaune. Typing messages for free in security protocols: the case of equivalence properties. In *Proc. 25th International Conference on Concurrency Theory (CONCUR'14)*, volume 8704 of *LNCS*, pages 372–386, Rome, Italy, Sept. 2014. Springer.

[16] R. Chrétien, V. Cortier, and S. Delaune. Decidability of trace equivalence for protocols with nonces. In *Proc. 28th Computer Security Foundations Symposium (CSF'15)*, pages 170–184. IEEE Computer Society, 2015.

[17] H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. In *Proc. 12th European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 99–113, Warsaw, Poland, Apr. 2003. Springer.

[18] V. Cortier, A. Dallon, and S. Delaune. Bounding the number of agents, for equivalence too. In *Proc. 5th International Conference on Principles of Security and Trust (POST'16)*, LNCS, pages 211–232. Springer, 2016.

[19] V. Cortier, A. Dallon, and S. Delaune. SAT-Equiv: an efficient tool for equivalence properties. In *Proc. 30th Computer Security Foundations Symposium (CSF'17)*, pages 481–494. IEEE Computer Society, August 2017.

[20] V. Cortier, A. Dallon, and S. Delaune. A small bound on the number of sessions for security protocols. Technical report, HAL report 03473179, 2021.

[21] V. Cortier, S. Delaune, and V. Sundararajan. A decidable class of security protocols for both reachability and equivalence properties. *Journal of Automated Reasoning*, 65(4):479–520, 2021.

[22] J. Dawson and A. Tiu. Automating open bisimulation checking for the spi-calculus. In *Proc. 23rd Computer Security Foundations Symposium (CSF'10)*. IEEE Computer Society, 2010.

[23] E. D'Osualdo, L. Ong, and A. Tiu. Deciding secrecy of security protocols for an unbounded number of sessions: The case of depth-bounded processes. In *Proc. 30th Computer Security Foundations Symposium, (CSF'17)*, pages 464–480. IEEE Computer Society, 2017.

[24] E. D'Osualdo and F. Stutz. Decidable inductive invariants for verification of cryptographic protocols with unbounded sessions. In *Proc. 31st International Conference on Concurrency Theory (CONCUR'20)*, volume 171 of *LIPIcs*, pages 31:1–31:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[25] N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. 1999.

[26] S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006.

[27] S. Fröschle. Leakiness is decidable for well-founded protocols? In *Proc. 4th Conference on Principles of Security and Trust (POST'15)*, LNCS. Springer, Apr. 2015.

[28] J. D. Guttman. Shapes: Surveying crypto protocol runs. In V. Cortier and S. Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, pages 222–257. IOS Press, 2011.

[29] A. V. Hess and S. Mödersheim. A typing result for stateful protocols. In *Proc. 31st Computer Security Foundations Symposium, (CSF'18)*, pages 374–388. IEEE Computer Society, 2018.

[30] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In N. Sharygina and H. Veith, editors, *Proc. 25th Computer Aided Verification, 25th International Conference (CAV'13)*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.

[31] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 8th ACM Conference on Computer and Communications Security (CCS'01)*, 2001.

[32] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Proc. 14th Computer Security Foundations Workshop (CSFW'01)*, pages 174–190. IEEE Computer Society, 2001.

APPENDIX A
TYPING RESULTS

A. Typing result for reachability

In [14], the existence of a well-typed witness for processes which do not feature the match construction is established. This result is stated for processes without replication, and then lifted to configuration which feature replication. The way

this result is proved allows one to derive an extra property regarding the shape of the resulting well-typed witness. This result (stated and proved in [14] in a slightly different version as Theorem 3.8) is recalled below. Note that in this version, the protocol can contain $\mathsf{new}\, n$ and $\mathsf{new}\, c.\mathsf{out}(c, c')$ instructions (as it is not hard to add them when there is no replication).

Theorem 4. *Let P be a protocol type-compliant w.r.t. (Δ_0, δ_0) which does not feature replication, nor match constructions and $\mathcal{S}$ be a set of pairs of terms.*

If $P \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$, then there exists a well-typed execution $P \xRightarrow{\mathsf{tr}'} (\mathcal{P}; \phi'; \sigma'; i)$ w.r.t. such that $\overline{\mathsf{tr}'} = \overline{\mathsf{tr}}$.

Moreover, w.l.o.g. we can assume that the substitution σ' is such that $u\sigma'$ and $v\sigma'$ are not unifiable for any $u = v \in \mathcal{S}$ such that $u\sigma$ and $v\sigma$ are not unifiable.

Proof. (sketch of proof) The main part of the theorem (without the consideration of the set $\mathcal{S}$) has been proved in [14]. Actually, it has been shown that the substitution σ' is directly derived from σ as follows: We have that $\sigma' = \rho \circ \sigma_S$ where:

- $\sigma_S = mgu(\Gamma)$ and $\Gamma = \{(u, v) \mid u, v \in ESt(P)$ such that $u\sigma = v\sigma\}$; and
- ρ is a bijective renaming from variables in $dom(\sigma) \setminus dom(mgu(\Gamma))$ to fresh constants (in particular those constants do not occur in $\mathcal{S}$).

Moreover, the existence of a substitution λ whose domain is $dom(\lambda) = dom(\rho)$ and which satisfies $\lambda \circ \sigma_S = \sigma$ is established.

Let $u = v \in \mathcal{S}$. We have to establish that, if $u\sigma$ and $v\sigma$ are not unifiable, then $u\sigma'$ and $v\sigma'$ are not unifiable either. We proceed by contraposition: we assume that $u\sigma'$ and $v\sigma'$ are unifiable, and we want to prove that $u\sigma$ and $v\sigma$ are unifiable too. As $u\sigma'$ and $v\sigma'$ are unifiable, there exists γ such that $(u\sigma')\gamma = (v\sigma')\gamma$ and since $img(\sigma')$ only contains ground terms, we actually have that:

$$u(\sigma' \uplus \gamma) = v(\sigma' \uplus \gamma).$$

Since ρ is a bijective renaming and fresh constants in $img(\rho)$ does not occur in u and v, we have that:

$$u(\sigma'\rho^{-1} \uplus \gamma\rho^{-1}) = v(\sigma'\rho^{-1} \uplus \gamma\rho^{-1}).$$

This allows us to conclude that:

$$u(\sigma_S \uplus \gamma\rho^{-1}) = v(\sigma_S \uplus \gamma\rho^{-1}).$$

Then, applying λ, we obtain that:

$$[u(\sigma_S \uplus \gamma\rho^{-1})]\lambda = [v(\sigma_S \uplus \gamma\rho^{-1})]\lambda.$$

Therefore, we have that:

$$[(u\sigma_S)\lambda](\gamma\rho^{-1}\lambda) = [(v\sigma_S)\lambda](\gamma\rho^{-1}\lambda).$$

This allows us to conclude that $u\sigma$ and $v\sigma$ are unifiable. This concludes the proof. □

Then, the idea is to show how relying on Theorem 4, we can establish a similar result for protocols which feature replication and match constructions.

Theorem 1. *Let P be a protocol type-compliant w.r.t. (Δ_0, δ_0). If $P \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ then there exists a well-typed execution $P \xRightarrow{\mathsf{tr}'} (\mathcal{P}; \phi'; \sigma'; i)$ such that $\overline{\mathsf{tr}'} = \overline{\mathsf{tr}}$.*

Proof. (sketch of proof) First, we make the observation that every attack trace is an attack trace against a finite execution. More formally, we have that if there exists an execution of P of length at most ℓ, then a similar execution can be done starting from $\mathsf{unfold}_\ell(P)$. Conversely, any execution obtained starting from $\mathsf{unfold}_\ell(P)$ can be done starting from P itself.

Now, we are able to establish our typing result for reachability in presence of replication and match constructions. Assuming we have a trace tr starting from the configuration P, we have that this trace can be performed starting from $P' = \mathsf{unfold}_\ell(P)$ for some ℓ.

Along this execution, we consider all equalities $\Gamma^{\mathsf{tr}}_{\mathsf{match}}$ coming from the execution of a match construction, i.e. $\Gamma^{\mathsf{tr}}_{\mathsf{match}}$ gathers all the pair (x, u_i) corresponding to an execution of the i^{th} branch of a match construction of the form $\mathsf{match}\, x\, \mathsf{with}\, (u_1 \to P_1 \mid \ldots \mid u_j \to P_j)$. Then, we consider $\sigma^{\mathsf{tr}}_{\mathsf{match}} = mgu(\Gamma^{\mathsf{tr}}_{\mathsf{match}})$, and we denote P'' the process obtained from P' by replacing each match instruction $\mathsf{match}\ x\ \mathsf{with}(u_1 \to P_1 | \ldots | u_j \to P_j)$ directly by its continuation P_i (the one used in the execution trace tr), and we apply the substitution $\sigma^{\mathsf{tr}}_{\mathsf{match}}$. The resulting configuration P'' does not feature match construction anymore, and we have that $ESt(P'') \subseteq ESt(P'\sigma^{\mathsf{tr}}_{\mathsf{match}})$.

Now, Theorem 4 applies on P''. First, let us show that type-compliance is satisfied on configuration P''. Let $u_1, u_2 \in ESt(P'')$ such that u_1 and u_2 are unifiable. Up to a bijective renaming of names and variables, we have that $u_1, u_2 \in ESt(\mathsf{unfold}_2(P))$ are two unifiable terms, and thus by definition of type-compliance as stated in Definition 4, we know that $\delta(u_1) = \delta(u_2)$. Thus, type-compliance is satisfied by P''. Second, for $\mathcal{S}$, we consider the set of all the pairs of terms corresponding to match that did not pass. In other words, for each match instruction $\mathsf{match}\ x\ \mathsf{with}\ (u_1 \to P_1 | \ldots | u_j \to P_j)$ that executes its i^{th} branch in tr, we add $(x, u_1), \ldots, (x, u_{i-1})$ in $\mathcal{S}$. Theorem 4 gives us the existence of a well-typed execution $P'' \xRightarrow{\mathsf{tr}''} (\mathcal{P}; \phi''; \sigma''; i)$ such that $\overline{\mathsf{tr}''} = \overline{\mathsf{tr}}$. Moreover, w.l.o.g. we can assume that the substitution σ'' is such that $u\sigma''$ and $v\sigma''$ are not unifable for any $u = v \in \mathcal{S}$ such that $u\sigma = v\sigma$ were not unifiable.

Therefore, we have that P' can execute tr'' leading to a configuration with subsitution $\sigma^{\mathsf{tr}}_{\mathsf{match}}\sigma'' \cup \sigma''$. Thus we have:

$$\sigma' = \sigma^{\mathsf{tr}}_{\mathsf{match}}\sigma'' \cup \sigma'' \tag{1}$$

Both substitutions being well-typed, we have that $\sigma^{\mathsf{tr}}_{\mathsf{match}}\sigma'' \cup \sigma''$ is well-typed too. Then, it is easy to see that this well-typed execution starting from P' (a finite version of P) can be mimicked starting from P. This concludes the proof. □

B. Some extra properties

In addition to consider a well-typed witness, we want a witness that involes simple asap recipes, and the witness has

also to be cv-alien-free. The first property (simple asap) is easy to establish.

We rely on the notion of *forced normal form* as introduced in [14] and denoted $u\downarrow$. This is the normal form obtained when applying rewrite rules as soon as the symbol in $\Sigma_{\mathsf{d}} \uplus \Sigma_{\mathsf{test}}$ and the constructor match. Formally, we reuse the definition as stated in [14]:

Definition 10. *Given a rewriting rule $\ell_{\mathsf{g}} \to r_{\mathsf{g}}$ as defined in Section III-A, its* associated forced rewriting rule *is $\ell'_{\mathsf{g}} \to r_{\mathsf{g}}$ where ℓ'_{g} is obtained from ℓ_{g} by keeping only the path to r_{g} in ℓ_{g}. Formally, ℓ'_{g} is defined as follows:*

1) *$\ell'_{\mathsf{g}} = \mathsf{g}(x_1, \ldots, x_n)$ when $\mathsf{g} \in \Sigma_{\mathsf{test}}$;*
2) *otherwise denoting $p_0 = 1.j$ the unique position of ℓ_{g} such that $\ell_{\mathsf{g}}|_{p_0} = r_{\mathsf{g}}$, we have that ℓ'_{g} is the linear term such that:*
   - *$\mathsf{root}(\ell'_{\mathsf{g}}|_\epsilon) = \mathsf{root}(\ell_{\mathsf{g}}|_\epsilon)$, $\mathsf{root}(\ell'_{\mathsf{g}}|_1) = \mathsf{root}(\ell_{\mathsf{g}}|_1)$; and $\ell'_{\mathsf{g}}|_{p_0} = r_{\mathsf{g}}$;*
   - *for any other position p' of ℓ'_{g}, we have that $\ell'_{\mathsf{g}}|_{p'}$ is a variable.*

Note that the forced rewriting system associated to a rewriting system is well-defined. In particular, the position of r_{g} in ℓ_{g} is uniquely defined. In particular, the position of r_{g} in $\ell_{\mathsf{g}}|_1$ is uniquely defined since $\ell_{\mathsf{g}}|_1$ is a linear term and r_{g} contains a variable when $\mathsf{g} \in \Sigma_{\mathsf{d}}$.

Given a rewriting system $\mathcal{R} = \mathcal{R}_{\mathsf{des}} \uplus \mathcal{R}_{\mathsf{test}}$, we define $\mathcal{R}_f$ the set of forced rewriting rules associated to $\mathcal{R}$. A term u can be rewritten to v using $\mathcal{R}_f$ if there is a position p in u, and a rewriting rule $\mathsf{g}(t_1, \ldots, t_n) \twoheadrightarrow t$ in $\mathcal{R}_f$ such that $u|_p = \mathsf{g}(t_1, \ldots, t_n)\theta$ for some substitution θ, and $v = u[t\theta]_p$. We denote $\twoheadrightarrow^*$ the reflexive and transitive closure of $\twoheadrightarrow$, and $u\downarrow$ the normal form of u.

As established in [14] (in a slightly more general setting), we may restrict our attention to recipes in forced normal form.

Lemma 1. *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 $R\downarrow\phi\downarrow = u$.*

We say that a recipe R is *almost destructor-only* when R is in forced normal form and $\mathsf{root}(R|_p) \in \Sigma_{\mathsf{d}} \cup \mathcal{W}$ for any position p of the form $1 \ldots 1$ in R.

A recipe R is *simple* when $R = C[R_1, \ldots, R_n]$ for some context C built using symbols in $\Sigma_{\mathsf{c}} \cup \Sigma_0$, and each R_i is almost destructor-only. Note that C can be empty, and thus an almost destructor-only recipe is also a simple recipe.

Lemma 2. *Let R be a recipe in forced normal form such that $\mathsf{root}(R) \in \Sigma_{\mathsf{d}} \cup \mathcal{W}$, and ϕ be a frame such that $R\phi\downarrow$ is a message. We have that R is almost destructor-only and $R|_p\phi\downarrow \in St(\phi)$ for any position p of the form $1 \ldots 1$ in R.*

Proof. We show this result by structural induction on R.

Base case: We have that $R \in \mathcal{W}$, and we easily conclude.

Induction case: We have that $R = \mathsf{f}(R_1, \ldots, R_k)$ for some $\mathsf{f} \in \Sigma_{\mathsf{d}}$, and we know that $R_1, \ldots, R_k$ are in forced normal form. Let $\ell_{\mathsf{f}} \to r_{\mathsf{f}}$ be the rule in $\mathcal{R}$ associated to f, and $\ell'_{\mathsf{f}} \to r_{\mathsf{f}}$ its associated forced rewriting rule. We have that $\ell_{\mathsf{f}} = \mathsf{f}(\mathsf{g}(t_1^1, \ldots, t_1^p), t_2, \ldots, t_k)$ for some $\mathsf{g} \in \Sigma_{\mathsf{c}}$ and $r_{\mathsf{f}} = t_1^j$ for some j. Since $R\phi\downarrow$ is a message, we know that $\mathsf{root}(R_1\phi\downarrow) = \mathsf{g}$. Assume by contradiction that $\mathsf{root}(R_1) \in \Sigma_{\mathsf{c}}$, then $\mathsf{root}(R_1) = \mathsf{g}$. Now, since R is in forced normal form, we know that $\mathsf{root}(R_1) \neq \mathsf{g}$. So we deduce that $\mathsf{root}(R_1) \notin \Sigma_{\mathsf{c}}$. Therefore, R_1 is in forced normal form, $\mathsf{root}(R_1) \in \Sigma_{\mathsf{d}} \cup \mathcal{W}$, $R_1\phi\downarrow$ is a message. We conclude relying on our induction hypothesis. □

Lemma 3. *Let R be a recipe in forced normal form and ϕ be a frame such that $R\phi\downarrow$ is a message. We have that R is a simple recipe.*

Proof. We prove this result by structural induction on R.

Base case: In case $\mathsf{root}(R) \in \Sigma_{\mathsf{d}} \cup \mathcal{W}$, we conclude relying on Lemma 2.

Induction case: We have that either $R \in \Sigma_0$ or $R = \mathsf{f}(R_1, \ldots, R_k)$ for some $\mathsf{f} \in \Sigma_{\mathsf{c}} \cup \Sigma_{\mathsf{test}}$. In case $R \in \Sigma_0$, we have that R is simple by definition. Otherwise, applying our induction hypothesis on $R_1, \ldots, R_n$, we deduce that $R_1, \ldots, R_n$ are simple recipes, and thus R is simple too. It remains the case where $\mathsf{f} \in \Sigma_{\mathsf{test}}$. Actually, this case is impossible since R is in forced normal form. □

To justify marking, we have to restrict to witnesses relying on asap recipes (see Definition 7). Whenever a message is deducible, we can also find an asap recipe of the message which is also simple.

Lemma 4. *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' a simple asap recipe such that $R'\phi\downarrow = u$.*

Proof. We first chose among all the recipes $\{R' \mid R'\phi\downarrow = u\}$, one that is minimal. Let R_0 be such a recipe, and then we consider $R_0\downarrow$. Thanks to Lemma 1, we have that $R_0\downarrow$ is a recipe for u. Thanks to Lemma 3, we have that $R_0\downarrow$ is imple. It is also asap since $R_1 \twoheadrightarrow R_2$ implies $R_2 \leq R_1$. This allows us to conclude. □

Let $\mathsf{Cst}(u)$ be the set of constants from Σ_0 occurring in u. We have the following result.

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 $R = \mathsf{g}(R_1, \ldots, R_k)$, and thanks to Lemma 2, we know that $R_1\phi\downarrow \in St(\phi)$. Regarding $R_2, \ldots, R_k$, we have that

$\mathsf{Cst}(R_2\phi\downarrow) \cup \ldots \cup \mathsf{Cst}(R_k\phi\downarrow) \subseteq \mathsf{Cst}(R_1\phi\downarrow)$. Therefore, applying our induction hypothesis on R_i that are simple, we have that:

$$\begin{array}{rcl} \mathsf{Cst}(R) & = & \mathsf{Cst}(R_1) \cup \ldots \cup \mathsf{Cst}(R_k) \\ & \subseteq & \mathsf{Cst}(\phi) \cup \mathsf{Cst}(R_1\phi\downarrow) \cup \ldots \cup \mathsf{Cst}(R_k\phi\downarrow)) \\ & \subseteq & \mathsf{Cst}(\phi) \cup \mathsf{Cst}(R_1\phi\downarrow) \\ & \subseteq & \mathsf{Cst}(\phi) \end{array}$$

This concludes the proof. □

Lemma 6. *Let $\mathcal{K}_0$ be a configuration of the form $(\mathcal{P}_0; \phi_0; \emptyset; 0)$. Let σ be a mgu of terms of $\mathcal{P}_0$. We have that $St(\mathcal{K}_0\sigma) \subseteq St(\mathcal{K}_0)\sigma$.*

Proof. We consider σ the mgu between pairs of terms occurring in $St(\mathcal{K}_0)$, and we show that $St(\mathcal{K}_0\sigma) \subseteq St(\mathcal{K}_0)\sigma$.

Let Γ be the set of pairs of terms such that $\sigma = mgu(\Gamma)$. Let $t \in St(\mathcal{K}_0\sigma)$ be such that $t \notin St(\mathcal{K}_0)\sigma$. Therefore, we have that $t \in St(img(\sigma))$. As the mgu does not create variables, if t is a variable then $t \in St(\Gamma) \subseteq St(\mathcal{K}_0)$, so we assume t is not a variable.

Let $\overline{\sigma} = \sigma\delta$ where δ replaces any occurrence of t in $img(\sigma)$ by a fresh variable x. We have that:

- $u\overline{\sigma} = v\overline{\sigma}$ for any equation $u = v \in \Gamma$. Indeed, we know that $u\sigma = v\sigma$ for any such $u = v$, and thus $(u\sigma)\delta = (v\sigma)\delta$. Since $t \notin ESt(\mathcal{K}_0)\sigma$, and $u, v \in ESt(\mathcal{K}_0)$, we deduce that $u\delta = u$ and:
$$(u\sigma)\delta = u(\sigma\delta \cup \delta) = u\overline{\sigma}$$
and similarly $(v\sigma)\delta = v\overline{\sigma}$.
- $\overline{\sigma}$ is strictly more general that σ. Indeed, we have that $\sigma = \overline{\sigma}\tau$ considering $\tau = \{x \mapsto t\}$ and t is not a variable.

This leads to a contradiction since $\sigma = mgu(\Gamma)$ and concludes the proof. □

We denote by $\mathsf{Terms}(\phi)$ the set of terms occurring in $img(\phi)$. This notation is also used to denote the set of terms occurring in a trace, e.g. $\mathsf{Terms}(\mathsf{tr}\phi\downarrow)$ denotes the set of terms occurring in the trace $\mathsf{tr}\phi\downarrow$.

We are now able to state and prove a stronger version of Theorem 1 in which the resulting witness tr' satisfied some additional properties.

Theorem 5. *Let P be a protocol type-compliant w.r.t. (Δ_0, δ_0) (which may feature replication and match constructions).*

If $P \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ then there exists a well-typed execution $P \xRightarrow{\mathsf{tr}'} (\mathcal{P}; \phi'; \sigma'; i)$ involving only simple asap recipes such that $\overline{\mathsf{tr}'} = \overline{\mathsf{tr}}$.

Moreover, we may assume that tr' and ϕ' are cv-alien-free.

Proof. (sketch of proof) The first part of the theorem is a direct consequence of Theorem 1. Then, the fact that we can consider simple asap recipes is an easy consequence of Lemma 4. It remains to establish that we can consider tr' and ϕ' to be cv-alien-free. We have seen in the proof of Theorem 1 (see Equation (1)) that the well-typed substitution σ' is of the form $\sigma' = \sigma^{\mathsf{tr}}_{\mathsf{match}}\sigma'' \cup \sigma''$ where $\sigma'' = \rho \circ \sigma_S$ where:

- $\sigma^{\mathsf{tr}}_{\mathsf{match}}$ is the most general unifier of $\Gamma^{\mathsf{tr}}_{\mathsf{match}}$ the set of pairs (x, u_i) corresponding to match instructions that are executed along tr;
- σ_S is the most general unifier of Γ where:
$$\Gamma = \left\{ \begin{array}{c|c} (u, v) & u, v \in ESt(P\sigma^{\mathsf{tr}}_{\mathsf{match}}) \\ & \text{such that } u\sigma = v\sigma \end{array} \right\}$$
- ρ is a bijective renaming from variables in $dom(\sigma) \setminus dom(\sigma_S)$ to some fresh constants preserving type.

We have that $St((P\sigma^{\mathsf{tr}}_{\mathsf{match}})\sigma_S) \subseteq St(P\sigma^{\mathsf{tr}}_{\mathsf{match}})\sigma_S$ since σ_S is a mgu between subterms occurring in P by Lemma 6. Then, since ρ is a renaming, we deduce that $St(P\sigma') \subseteq St(P\sigma^{\mathsf{tr}}_{\mathsf{match}})\sigma''$. Similarly, we have that: $St(P\sigma^{\mathsf{tr}}_{\mathsf{match}}) \subseteq St(P)\sigma^{\mathsf{tr}}_{\mathsf{match}}$ since $\sigma^{\mathsf{tr}}_{\mathsf{match}}$ is a mgu between subterms occurring in P. Therefore, we deduce that:

$$St(P\sigma') \subseteq (St(P)\sigma^{\mathsf{tr}}_{\mathsf{match}})\sigma'' = St(P)\sigma' \tag{2}$$

We now show that $\mathsf{tr}'\phi'\downarrow$ is cv-alien-free. Assume by contradiction that there exists a constant c_h from Σ_0 of cv-alien 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 (due to some unfolding). Thus, we have that $c_h \in St(P\sigma')$. Thanks to the property 2, we deduce that $c_h \in St(P)\sigma'$, and since c_h does not occur in $St(P)$ (as any constant from Σ_0 of cv-alien type), we deduce that there exists $x \in St(P)$ such that $x\sigma' = c_h$. However, this is impossible too since no variable having such a type can occur in P. This allows us to conclude that $\mathsf{tr}'\phi'\downarrow$ is cv-alien-free.

We have that $\mathsf{Terms}(\phi') \subseteq \mathsf{Terms}(\mathsf{tr}'\phi'\downarrow)$, and thus we easily deduce that ϕ' is cv-alien-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 from Σ_0 of cv-alien type occurs in ϕ' and $\mathsf{tr}'\phi\downarrow$, we are done. □

APPENDIX B
COMPUTATION OF THE BOUND

A. Preliminaries

We consider that a marking is appropriate 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 not to take into account this subterm during the computation of dep.

An almost destructor-only recipe R such that $R\phi\downarrow$ is a message, intuitively tries to dig in a 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:

- ϵ if R is a variable w
- $\mathsf{target}(R|_1).i_0$ if $\mathsf{root}(R) = \mathsf{g} \in \Sigma_{\mathsf{d}}$ and $\mathsf{g}(t_1, \ldots, t_n) \to r_{\mathsf{g}} \in \mathcal{R}_{\mathsf{d}}$ with $t_1|_{i_0} = r_{\mathsf{g}}$.

Note that the operation target defined above is well-defined for any almost destructor-only recipe.

Definition 11. *Let P be a protocol. A marked position (α, 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 $\mathsf{out}^\alpha(c, \mathsf{w})$ occurring in tr, for any almost 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).*

B. Properties of dep

Now, we show that dep satisfies the properties stated in Figure 4. These properties will be used in the following section to establish our main result. As an illustrative example, we detailed below the proof of Property 5. The other ones can be established in a similar way.

Proof of Property 5. Let τ be an initial type. For any $i > 0$, we have that:

$$\mathsf{dep}(\tau) \supseteq \mathsf{dep}^i(\tau) = \mathsf{dep}^0(\tau) \cup S_{\mathsf{out}}^{i-1}(\tau)$$

We are going to show first that $\mathsf{dep}(\tau) \supseteq \mathsf{dep}^0(\tau) \cup S_{\mathsf{out}}(\tau)$. To establish this, let $e \in \mathsf{dep}^0(\tau) \cup S_{\mathsf{out}}(\tau)$. Since $\mathsf{dep}^0(\tau) \subseteq \mathsf{dep}(\tau)$, in case $e \in \mathsf{dep}^0(\tau)$, we easily conclude. Otherwise, we have that $e \in S_{\mathsf{out}}(\tau)$, and thus $e \in S_{\mathsf{out}}^i(\tau)$ for some $i > 0$ by definition of $S_{\mathsf{out}}(\tau)$, and this allows us to conclude that $e \in \mathsf{dep}(\tau)$.

Conversely, let $e \in \mathsf{dep}(\tau)$. Then, we have that $e \in \mathsf{dep}^i(\tau)$ for some $i > 0$. Hence, we have that: $e \in \mathsf{dep}^0(\tau) \cup S_{\mathsf{out}}^{i-1}(\tau) \subseteq \mathsf{dep}^0(\tau) \cup S_{\mathsf{out}}(\tau)$ which allows us to conclude.

When considering recipes for a witness of non static inclusion, we can not asume anymore that these recipes are all asap, and thus marking is not justified. However, we can assume they are subterm asap, and this is the reason why we still consider a bit of marking.

The definition of $S_{\mathsf{out}}^+(\tau)$ is given in Section V. In addition, we define

$$\mathsf{dep}^+(\tau) = \mathsf{dep}^0(\tau) \cup (\{\emptyset\} \otimes \mathsf{dep}(\tau_1) \otimes \ldots \otimes \mathsf{dep}(\tau_k)) \cup S_{\mathsf{out}}^+(\tau)$$

for any non-initial type τ such that $\tau = \mathsf{f}(\tau_1, \ldots, \tau_k)$.

C. Main result

To prove our main result, we need to define ρ on terms. The definition is similar to the one for types.

Definition 12. *Given a term t, we define $\rho(t)$ to be $\rho(t, \epsilon, \emptyset)$ where $\rho(t, p, S)$ is recursively defined as the set $\{(t, p)\#S\} \cup E$ where $E = \emptyset$ when t is an atom (e.g. a constant or a name). Otherwise, we have that $t = \mathsf{f}(t_1, \ldots, t_k)$ and E is defined as:*

$$E = \bigcup_{\substack{\mathsf{g}(\ell_1, \ldots, \ell_n) \to r_{\mathsf{g}} \in \mathcal{R}_{\mathsf{d}} \\ \theta = mgu(\ell_1, \mathsf{f}(t_1, \ldots, t_k)) \\ i_0 \in \{1, \ldots, k\} \text{ is such that } r_{\mathsf{g}} = \ell_1|_{i_0}}} \rho(t_{i_0}, p.i_0, S \uplus \{\ell_2\theta, \ldots, \ell_n\theta\})$$

In the definition, S is a multiset of terms.

Of course, there is a link between the definition of ρ on types and the one on terms.

Lemma 7. *Let (Δ_0, δ_0) be a typing system, and u be a term. We have that:*

$$(v, p)\#S \in \rho(u) \text{ implies } (\delta_0(v), p)\#\delta_0(S) \in \rho(\delta_0(u)).$$

Proof. We establish the following result by structural induction on u.

$$(v, p)\#S \in \rho(u, p_0, S_0) \text{ implies } (\delta_0(v), p)\#\delta_0(S) \in \rho(\delta_0(u), p_0, \delta_0(S_0)).$$

The result is a direct consequence of this property. □

Note that the implication from types to terms does not hold since the structure of a type may be finer than the structure of the term. For instance, we may have $\delta(c) = \langle \tau_1, \tau_2 \rangle$.

Lemma 8. *Let ϕ be a frame and R be an almost destructor-only recipe such that $R\phi\downarrow$ is a message. We have that $(R\phi\downarrow, \mathsf{target}(R))\#S \in \rho(\mathsf{w}\phi)$ where $S = \{R|_p\phi\downarrow \mid p = 1 \ldots 1.i$ with $i \geq 2$ and p a position in $R\}$, and w is the variable occurring in R at its leftmost position.*

Proof. We establish the following result by structural induction on R:

$$\rho(R\phi\downarrow, \mathsf{target}(R), S) \subseteq \rho(\mathsf{w}\phi)$$

where $S = \{R|_p\phi\downarrow \mid p = 1 \ldots 1.i$ with $i \geq 2$ and p a position in $R\}$, and w is the variable occurring in R at its leftmost position. Note that the result stated in the lemma is a direct consequence of the property stated above.

Base case: $R \in \mathcal{W}$. In such a case, we have that $R\phi\downarrow = \mathsf{w}\phi$. We have that $\mathsf{target}(R) = \epsilon$, and $S = \emptyset$. By definition, we have that $\rho(\mathsf{w}\phi) = \rho(\mathsf{w}\phi, \epsilon, \emptyset)$. Thus the result holds.

Induction case: $\mathsf{root}(R) \in \Sigma_{\mathsf{d}}$. In such a case, we have that $R = \mathsf{g}(R_1, \ldots, R_n)$ for some $\mathsf{g} \in \Sigma_{\mathsf{d}}$. By induction hypothesis we have that $\rho((R_1\phi\downarrow, \mathsf{target}(R_1), S_1) \subseteq \rho(\mathsf{w}\phi)$ where

$$S_1 = \left\{ R_1|_p\phi\downarrow \;\middle|\; \begin{array}{r} p = 1 \ldots 1.i \text{ with } i \geq 2 \\ \text{and } p \text{ a position in } R_1 \end{array} \right\}.$$

We also know that $t = R_1\phi\downarrow = \mathsf{f}(t_1, \ldots, t_k)$ for some $t_1, \ldots, t_k$ and some f, and we consider the rule $\ell \to r \in \mathcal{R}_{\mathsf{d}}$ that is applied to lead to a message. We have that $\ell = \mathsf{g}(\mathsf{f}(\ell_1^1, \ldots, \ell_k^1), \ell_2, \ldots, \ell_n)$, and $r = \ell_{i_0}^1$ for some $i_0 \in \{1, \ldots, k\}$.

According to the definition of $\rho(t, \mathsf{target}(R_1), S_1)$, we have that it contains $\rho(t_{i_0}, \mathsf{target}(R_1).i_0, S_1 \cup \{\ell_2\theta, \ldots, \ell_n\theta\})$ where $\theta = mgu(\ell_1, \mathsf{f}(t_1, \ldots, t_k))$. Thus, we have that $\rho(t_{i_0}, \mathsf{target}(R_1).i_0, S_1 \cup \{\ell_2\theta, \ldots, \ell_n\theta\}) \subseteq \rho(t, \mathsf{target}(R_1), S_1)$. We have that:

$$\begin{array}{ll} & \rho(R\phi\downarrow, \mathsf{target}(R), S) \\ = & \rho(t_{i_0}, \mathsf{target}(R_1).i_0, S_1 \cup \{R_2\phi\downarrow, \ldots, R_n\phi\downarrow\}) \\ = & \rho(t_{i_0}, \mathsf{target}(R_1).i_0, S_1 \cup \{\ell_2\theta, \ldots, \ell_n\theta\}) \\ \subseteq & \rho(t, \mathsf{target}(R_1), S_1) \\ \subseteq & \rho(\mathsf{w}\phi) \end{array}$$

This allows us to conclude this case. □

1) $\mathsf{dep}(\bot) = \{\emptyset\}$;
2) In case α is an output, we have that: $\mathsf{dep}(\alpha) = \{\{\alpha\}\} \otimes \mathsf{dep}(\mathsf{pred}(\alpha))$.
3) In case α is an input of type τ, we have that: $\mathsf{dep}(\alpha) = \{\{\alpha\}\} \otimes \mathsf{dep}(\mathsf{pred}(\alpha)) \otimes \mathsf{dep}(\tau)$.
4) for any type τ, we have that

$$S_{\mathsf{out}}(\tau) = \bigcup_{\substack{\mathsf{out}^\alpha(c,u) \text{ occurring in } P \\ (\tau,p)\#\{\tau_1,\ldots,\tau_n\} \in \rho(\delta_0(u)) \\ (\alpha,p) \text{ not marked}}} \big(\mathsf{dep}(\alpha) \otimes \mathsf{dep}(\tau_1) \otimes \ldots \otimes \mathsf{dep}(\tau_n)\big).$$

5) for an initial type τ, we have that $\mathsf{dep}(\tau) = \mathsf{dep}^0(\tau) \cup S_{\mathsf{out}}(\tau)$;
6) for a non-initial type τ such that $\tau = \mathsf{f}(\tau_1,\ldots,\tau_k)$, we have that:

$$\mathsf{dep}(\tau) = \mathsf{dep}^0(\tau) \cup (\{\emptyset\} \otimes \mathsf{dep}(\tau_1) \otimes \ldots \otimes \mathsf{dep}(\tau_k)) \cup S_{\mathsf{out}}(\tau)$$

where $\mathsf{dep}^0(\tau) = \emptyset$ when τ is an cv-alien type; and $\mathsf{dep}^0(\tau) = \{\emptyset\}$ otherwise.

Fig. 4. Properties of dep

An execution $\mathcal{K}_0 \xRightarrow{\mathsf{tr}} (\mathcal{P};\phi;\sigma;i)$ of a protocol P can be seen as a dag D whose vertices are actions of tr with their label, and edges represent sequential and data dependencies.

Definition 13. *Given a dag $D = (V,E)$ and a set of nodes $N \subseteq V$, we define the pruning $D_N = (V_N, E_N)$ of D w.r.t. N as follows:*

- $V_N = \{v \in V \mid \exists\, r \in N, r \to^* v\}$;
- $E_N = \{(u,v) \in E \mid u,v \in V_N\}$

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

When we consider an execution and its associated dag, we can decide to prune the dag w.r.t. a given set of nodes N, then the resulting dag corresponds also to an execution of P.

Given an execution $exec$ and its associated dag D, we denote $exec|_v$ the execution obtained by pruning D w.r.t. the set of nodes $\{v\}$, and given a recipe R such that $vars(R) = \{\mathsf{w}_{i_1},\ldots,\mathsf{w}_{i_k}\}$, we denote $exec|_R$ the execution obtained by pruning D w.r.t. $\{v_1,\ldots,v_k\}$ where v_j is the node corresponding to the output w_{i_j}.

We denote $\mathsf{Label}(\mathsf{tr})$ (resp. $\mathsf{Label}(exec)$) the multiset of labels (from $\mathcal{L}$) occurring in tr (resp. $exec$), and given a node v of a dag (coming from an execution trace), we denote $\mathsf{Label}(v)$ its label.

A simple recipe is *subterm asap* w.r.t. ϕ if all its direct subterms are asap w.r.t. ϕ. Note that this implies that all its strict subterms are asap too.

Then, we are able to establish this key result by induction on the length of the trace.

Proposition 1. *Let P be a protocol type-compliant w.r.t. some typing systems (Δ_0,δ_0). We consider a well-typed and cv-alien-free execution $exec : P \xRightarrow{\mathsf{tr}} (\mathcal{P};\phi;\sigma;i)$ involving only simple asap recipes. We consider the dag D corresponding to this execution. We have that:*

1) *for any node v of D, we have that there exists a multiset $A \in \mathsf{dep}(\mathsf{Label}(v))$ such that $\mathsf{Label}(exec|_v) \subseteq A$;*
2) *for any almost destructor-only receipe R which is also asap (resp. subterm asap), and cv-alien-free, and such that $R\phi\!\downarrow$ is a message of type τ, we have that there exists a multiset $A \in S_{\mathsf{out}}(\tau)$ (resp. $S^+_{\mathsf{out}}(\tau)$) such that $\mathsf{Label}(exec|_R) \subseteq A$.*
3) *for any simple asap (resp. subterm asap), cv-alien-free recipe R such that $R\phi\!\downarrow$ is a message of type τ, we have that there exists a multiset $A \in \mathsf{dep}(\tau)$ (resp. $\mathsf{dep}^+(\tau)$) such that $\mathsf{Label}(exec|_R) \subseteq A$.*

Proof. We prove these results by induction on the length of tr.

Base case: tr is empty. In such a case, we have that there is no node v in D, and therefore item 1 trivially holds. Regarding item 2, since $dom(\phi) = \emptyset$ there is no recipe satisfying the condition. Regarding item 3, we have $R\phi\!\downarrow = R\!\downarrow = R$, and R is cv-alien-free. We show by induction on $|R|$ that $\mathsf{dep}(\tau) \neq \emptyset$ (where τ is the type of R). In case $|R| = 1$, we have that either $R \in \Sigma_{\mathsf{c}}$, and thus $\tau = R$, and $\emptyset \in \mathsf{dep}(\tau)$; or $R \in \Sigma_0$. In this case, we have that τ is not an cv-alien type, and thus $\emptyset \in \mathsf{dep}(\tau)$. Now, in case $R - \mathsf{f}(R_1,\ldots,R_n)$ with $\mathsf{f} \in \Sigma_{\mathsf{c}}$, by induction hypothesis, we have that $R_1,\ldots,R_n$ are cv-alien-free, and thus for any $i \in \{1,\ldots,n\}$, we have that $\mathsf{dep}(\tau_i) \neq \emptyset$ where τ_i is the type of R_i. This allows us to conclude that $\mathsf{dep}(\tau) = \mathsf{dep}(\mathsf{f}(\tau_1,\ldots,\tau_n)) \neq \emptyset$. Therefore, we have that $\mathsf{dep}(\tau) \neq \emptyset$ (resp. $\mathsf{dep}^+(\tau) \neq \emptyset$). We have that $\mathsf{Label}(exec|_R) = \emptyset$, and thus the result holds.

Inductive case: $\mathsf{tr} = \mathsf{tr}'.\alpha$. We denote $exec'$ the prefix of $exec$ corresponding to the trace tr'. We distinguish several cases depending on the type of the action α.

Case of an output. We assume that $\mathsf{tr} = \mathsf{tr}'.(\alpha : \mathsf{out}(c,\mathsf{w}))$ or $\mathsf{tr} = \mathsf{tr}'.(\alpha : \mathsf{out}(c,c'))$, and we establish the two items.

1) Let v be a node of D. In case α does not correspond to the node v, we have that $\mathsf{Label}(exec|_v) = \mathsf{Label}(exec'|_v)$. Relying on our induction hypothesis, we know that there exists $A \in \mathsf{dep}(\mathsf{Label}(v))$ such that $\mathsf{Label}(exec'|_v) \subseteq A$. This allows us to conclude.

Now, we assume that α corresponds to the node v.
In case $\mathsf{pred}(\alpha) = \bot$, then $\mathsf{dep}(exec|_v) = \{\alpha\}$ and $\mathsf{dep}(\mathsf{Label}(v)) = \{\{\alpha\}\}$, thus the result holds. Otherwise, we have that $\mathsf{pred}(\alpha) = \alpha'$ for some α', and we denote v' the node corresponding to this action in our dag D. Relying on our induction hypothesis, we have that there exists $A' \in \mathsf{dep}(\mathsf{Label}(v'))$ such that $\mathsf{Label}(exec'|_{v'}) \subseteq A'$.
By definition of dep, we have that $\mathsf{dep}(\mathsf{Label}(v)) = \{\{\alpha\}\} \otimes \mathsf{dep}(\mathsf{Label}(v'))$, and by definition of Label, we have that $\mathsf{Label}(exec|_v) = \mathsf{Label}(exec'|_{v'}) \uplus \{\alpha\}$. It is easy to see that $A = A' \uplus \{\alpha\}$ satisfies the requirement.

2) Let R be an *almost destructor-only recipe* which is also asap (resp. subterm asap), and cv-alien-free, and such that $R\phi\downarrow$ is a message of type τ. In case $\mathsf{w} \notin vars(R)$, we have that $\mathsf{Label}(exec|_R) = \mathsf{Label}(exec'|_R)$. Relying on our induction hypothesis, we know that there exists a multiset $A \in S_{\mathsf{out}}(\tau)$ (resp. $S^+_{\mathsf{out}}(\tau)$) such that $\mathsf{Label}(exec'|_R) \subseteq A$. This allows us to conclude.
Now, we assume that $\mathsf{w} \in vars(R)$. We establish the existence of $A \in S_{\mathsf{out}}(\tau)$ (resp. $S^+_{\mathsf{out}}(\tau)$ in case R is subterm asap) such that $\mathsf{Label}(exec|_R) \subseteq A$. We show this result by induction on the size of the asap (resp. subtem asap) recipe R. We have that $R\phi\downarrow$ is a message and R is an almost destructor-only recipe.. We denote $p = \mathsf{target}(R)$, w_0 the variable at the leftmost position in R (i.e. at position $1 \dots 1$), and $R_1, \dots, R_k$ the recipes at position $1 \dots 1.i$ with $i \geq 2$ in R for any such position which is well-defined. First, we may note that, for any $i \in \{1, \dots, k\}$, we have that $R_i\phi\downarrow$ is a message, R_i is in forced normal form, asap and cv-alien-free, and thus, thanks to Lemma 3, we know that for any $i \in \{1, \dots, k\}$, R_i is simple. Actually, we have that R_1 is an almost destructor-only recipe. Therefore, we can apply our induction hypothesis (item 3), and denoting $\tau_1, \dots, \tau_k$ the types of $R_1\phi\downarrow, \dots, R_k\phi\downarrow$, we obtain that there exists $A_i \in \mathsf{dep}(\tau_i)$ such that $\mathsf{Label}(exec|_{R_i}) \subseteq A_i$ for each $i \in \{1, \dots, k\}$.
Applying Lemma 8 on ϕ and R, we have that $(R\phi\downarrow, p)\#\{R_1, \dots, R_k\} \in \rho(\mathsf{w}\phi)$. Then, thanks to Lemma 7, we deduce that $(\tau, p)\#\{\tau_1, \dots, \tau_k\} \in \rho(\beta)$ where β is the label associated to the output w_0, and $\tau_1, \dots, \tau_k$ are the types of $R_1\phi\downarrow, \dots, R_k\phi\downarrow$. By definition of $S_{\mathsf{out}}(\tau)$, we have that:

$$\mathsf{dep}(\beta) \otimes \mathsf{dep}(\tau_1) \otimes \dots \otimes \mathsf{dep}(\tau_k) \in S_{\mathsf{out}}(\tau).$$

Note that by definition of a marked position to be appropriate, we know that (β, p) is not marked. Indeed, otherwise, this would contradict the fact that R is asap. Now, in case we only know that R is subterm asap, it remains to establish that (β, p) is not marked or $p = \epsilon$. Assume that $p \neq \epsilon$. We thus have that $p = p'.i_0$ for some i_0, and we therefore know that $R \notin \mathcal{W}$. We have that $R = \mathsf{g}(R_1, \dots, R_n)$ for some $\mathsf{g} \in \Sigma_{\mathsf{d}}$. We have that R_1 is cv-alien-free, $\mathsf{target}(R_1) = p'$, w_0 is the variable at the leftmost position in R_1, and $R_1\phi\downarrow = \mathsf{w}_0\phi|_{p'}$. Since R_1 is asap and almost destructor-only, by definition of an appropriate marking, we deduce that (β, p) is not marked.
Now, relying again on our induction hypothesis, we have that there exists $A_0 \in \mathsf{dep}(\mathsf{Label}(v_0)) = \mathsf{dep}(\beta)$ such that $\mathsf{Label}(exec|_{v_0}) \subseteq A_0$ where v_0 is the node associated to $\mathsf{out}(c, \mathsf{w}_0)$. Note that this has been established in item 1 above in case $\alpha = \beta$. To conclude, it is easy to see that $A = A_0 \uplus A_1 \uplus \dots \uplus A_k$ satisfies the requirement.

3) Let R be a simple asap (resp. subterm asap) and cv-alien-free recipe such that $R\phi\downarrow$ is a message of type τ. In case $\mathsf{w} \notin vars(R)$, we have that $\mathsf{Label}(exec|_R) = \mathsf{Label}(exec'|_R)$. Relying on our induction hypothesis, we know that there exists a multiset $A \in \mathsf{dep}(\tau)$ (resp. $\mathsf{dep}^+(\tau)$) such that $\mathsf{Label}(exec'|_R) \subseteq A$. This allows us to conclude.
Now, we assume that $\mathsf{w} \in vars(R)$. We establish the existence of $A \in \mathsf{dep}(\tau)$ (resp. $\mathsf{dep}^+(\tau)$ in case R is subterm asap) such that $\mathsf{Label}(exec|_R) \subseteq A$. We show this result by induction on the size of the asap (resp. subtem asap) recipe R.
In case R is an almost destructor-only recipe, we conclude relying on item 2. Note that $S_{\mathsf{out}}(\tau) \subseteq \mathsf{dep}(\tau)$ and $S^+_{\mathsf{out}}(\tau) \subseteq \mathsf{dep}^+(\tau)$.
Thus, we now consider the case where $R = \mathsf{f}(R_1, \dots, R_k)$ (with $k \geq 1$), and $\mathsf{f} \in \Sigma_{\mathsf{c}}$ or $R \in \Sigma_0$. In case $R = \mathsf{c}_0 \in \Sigma_0$, we know that τ is not cv-alien (since R is cv-alien-free), and therefore we have that $\emptyset \in \mathsf{dep}(\tau)$, and $\mathsf{dep}(exec|_R) = \{\emptyset\}$, and the result holds. Note that $\mathsf{dep}(\tau) \subseteq \mathsf{dep}^+(\tau)$.
Otherwise, we have that $R = \mathsf{f}(R_1, \dots, R_k)$, and $R_1, \dots, R_k$ are simple, asap, and cv-alien-free. Note that, R_i not asap will contradict the fact that R is asap or subterm asap. Relying on our induction hypothesis, we have that, for each $i \in \{1, \dots, k\}$, there exists $A_i \in \mathsf{dep}(\tau_i)$ such that $\mathsf{Label}(exec|_{R_i}) \subseteq A_i$, and we have that $\mathsf{dep}(\tau_1) \otimes \dots \otimes \mathsf{dep}(\tau_k) \in \mathsf{dep}(\tau) \subseteq \mathsf{dep}^+(\tau)$. It is therefore easy to see that $A = A_1 \uplus \dots \uplus A_k$ satisfies the requirement.

Case of an input. Now, we assume that $\mathsf{tr} = \mathsf{tr}'.(\alpha : \mathsf{in}(c, R_{\mathsf{in}}))$ and we establish the two items.

1) Let v be a node in D. In case α does not correspond to the node v, we have that $\mathsf{Label}(exec|_v) = \mathsf{Label}(exec'|_v)$. Relying on our induction hypothesis, we know that there exists $A \in \mathsf{dep}(\mathsf{Label}(v))$ such that $\mathsf{Label}(exec'|_v) \subseteq A$. This allows us to conclude.
Now, we assume that α corresponds to the node v.
We establish the existence of $A \in \mathsf{dep}(\mathsf{Label}(v))$ with $\mathsf{Label}(exec|_v) \subseteq A$.
In case $\mathsf{pred}(\alpha) = \bot$, then $\mathsf{Label}(exec|_v) = \mathsf{Label}(exec'|_{R_{\mathsf{in}}}) \uplus \{v\}$. Applying our induction hypothesis on tr' (item 3) with R_{in}, we deduce that there exists $A'_0 \in \mathsf{dep}(\tau)$ such that $\mathsf{Label}(exec'|_{R_{\mathsf{in}}}) \subseteq A'_0$.
By definition, we have that $\{\{\alpha\}\} \otimes \mathsf{dep}(\tau) = \mathsf{dep}(\mathsf{Label}(v))$. It is easy to see that $A = A'_0 \uplus \{\alpha\}$

satisfies the requirement.
Now, in case $\mathsf{pred}(\alpha) = \alpha'$ for some α'. We denote v' the node corresponding to this action in our execution dag. Relying on our induction hypothesis, we know that there exists $A' \in \mathsf{dep}(\mathsf{Label}(v')) = \mathsf{dep}(\alpha')$ such that $\mathsf{Label}(exec'|_{v'}) \subseteq A'$. We have that

$$\mathsf{Label}(exec|_v) = \mathsf{Label}(exec'|_{R_{\mathsf{in}}}) \uplus \mathsf{Label}(exec'|_{v'}) \uplus \{v\}.$$

Applying our induction hypothesis on tr' (item 3) with R_{in} we deduce that there exists $A'_0 \in \mathsf{dep}(\tau)$ such that $\mathsf{Label}(exec'|_{R_{\mathsf{in}}}) \subseteq A'_0$. By definition, we have that $\{\{\alpha\}\} \otimes \mathsf{dep}(\alpha') \otimes \mathsf{dep}(\tau) = \mathsf{dep}(\mathsf{Label}(v))$. It is easy to see that $A = A'_0 \uplus A' \uplus \{\alpha\}$ satisfies the requirement.

2) Let R be an almost destructor-only asap (resp. subterm asap), cv-alien-free recipe such that $R\phi\downarrow$ is a message of type τ. Since α is an input, we have that $\mathsf{Label}(exec|_R) = \mathsf{Label}(exec'|_R)$. Relying on our induction hypothesis, we know that there exists a multiset $A \in S_{\mathsf{out}}(\tau)$ (resp. $S^+_{\mathsf{out}}(\tau)$) such that $\mathsf{Label}(exec'|_R) \subseteq A$. This allows us to conclude.
3) Let R be a simple asap (resp. subterm asap), cv-alien-free recipe such that $R\phi\downarrow$ is a message of type τ. Since α is an input, we have that $\mathsf{Label}(exec|_R) = \mathsf{Label}(exec'|_R)$. Relying on our induction hypothesis, we know that there exists a multiset $A \in \mathsf{dep}(\tau)$ (resp. $\mathsf{dep}^+(\tau)$) such that $\mathsf{Label}(exec'|_R) \subseteq A$. This allows us to conclude.

This concludes the proof of this result. □

D. Proof of Theorem 2

We are now able to prove our main result regarding reachability.

Theorem 2. *Let P be a protocol type-compliant w.r.t. some typing system (Δ_0, δ_0). Let α be a label of P and assume $(\mathsf{tr}.\ell, \phi) \in \mathsf{trace}(P)$ for some tr, ℓ, ϕ such that $\mathsf{Label}(\ell) = \{\alpha\}$. Then there exists $\mathsf{tr}', \ell', \phi'$, and $A \in \mathsf{dep}(\alpha)$ such that $(\mathsf{tr}'.\ell', \phi') \in \mathsf{trace}(P)$ with $\mathsf{Label}(\ell') = \{\alpha\}$; and $\mathsf{Label}(\mathsf{tr}'.\ell') \subseteq A$.*

Proof. By hypothesis, we have that $(\mathsf{tr}.\ell, \phi) \in \mathsf{trace}(P)$, and thus there is an execution $exec$ witnessing this fact. Thanks to Theorem 5, we can assume that this execution is well-typed, and involves simple asap recipes. Moreover, we can assume that $\mathsf{tr}.\ell$ and ϕ are cv-alien-free. Thus, this execution is cv-alien-free.

We consider the dag D corresponding to this execution, and we denote v the node corresponding to the action ℓ. Thanks to Proposition 1 (item 1), we have that there exists a multiset $A \in \mathsf{dep}(\mathsf{Label}(v))$ such that $\mathsf{Label}(exec|_v) \subseteq A$. Note that $\mathsf{Label}(v) = \mathsf{Label}(\ell) = \{\alpha\}$, and thus if we denote tr'' the trace underlying the execution $exec|_v$, we have that $\mathsf{tr}'' = \mathsf{tr}'.\ell'$ for some tr' and ℓ' such that $\mathsf{Label}(\ell') = \{\alpha\}$. Moreover, we have that $(\mathsf{tr}'.\ell', \phi') \in \mathsf{trace}(P)$ for some ϕ' (indeed by definition of $exec|_v$ we keep all the dependencies, and thus $exec|_v$ is an execution of P). Thus, we deduce the existence of $\mathsf{tr}', \ell', \phi'$, and $A \in \mathsf{dep}(\alpha)$ such that $(\mathsf{tr}'.\ell', \phi') \in \mathsf{trace}(P)$ with $\mathsf{Label}(\ell') = \{\alpha\}$, and $\mathsf{Label}(\mathsf{tr}'.\ell') \subseteq A$. □

E. Marking criteria

We now establish the following result regarding terms with a public type (see Definition 8).

Lemma 9. *Let P be a protocol, (Δ_0, δ_0) be a typing system, and u be a term having a public type. Let $P \xrightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ 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. To show the result, we establish that each leaf of u is either a constant in Σ_0 or a constant in Σ_{c}. Let l be a leaf of u. We have that l is either a name or a constant. In case l is a constant, we are done. Otherwise, we have that $l \in \mathcal{N}$, and we have that $\delta_0(l) \in St(\tau_p)$. Since $R\phi\downarrow = u$ for some recipe R, we have that l occurs somewhere in ϕ, and thus a name n such that $\delta_0(n) = \delta_0(l)$ occurs in P. Therefore, we have that $\delta_0(n) \in St(\tau_p)$ for some name n occurring in P. This contradicts the fact that τ_p is a public type, and allows us to conclude the proof. □

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

Lemma 10. *Let (α, p) be a marked position of a protocol P w.r.t. a typing system (Δ_0, δ_0). Let u be the term such that $\mathsf{out}^\alpha(c, u)$ occurs in P. If $\delta_0(u)|_p$ has a public type then (α, p) is appropriate.*

Proof. We consider a well-typed execution $P \xrightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ and $\mathsf{out}^\alpha(c, \mathsf{w})$ occurring in tr. Let R be a recipe in forced normal form such that $\mathsf{root}(R) \in \Sigma_{\mathsf{d}} \cup \mathcal{W}$ 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 9, and therefore R (which contains w) is not an asap recipe for $R\phi\downarrow = m = \mathsf{w}\phi|_p$. □

Our second criterion is a *precedence criterion*. We extend the criterion proposed in [16], [21] in order to make it applicable in presence of match constructions.

Example 19. *Consider the following process P modelling a simple challenge response exchange relying on a match construction:*

$$P = \mathsf{in}^\alpha(c, y).\mathsf{match}\ y\ \mathsf{with}\ \mathsf{senc}(\langle \mathsf{req}, x\rangle, k) \to \mathsf{out}^\beta(c, \mathsf{senc}(\langle \mathsf{rep}, x\rangle, k))$$

Type-compliance is satisfied considering $\delta_0(y) = \delta_0(\mathsf{senc}(\langle \mathsf{req}, x\rangle, k))$. Clearly, we have that $(\beta, 1.2)$ is an appropriate marked position, Indeed, extracting the value corresponding to x from the output β is actually useless since this value was accessible before through the input α under the exact same protection (here the key k). Any asap recipe will favour the use of the recipe used in input rather than the one which digs into the output.

Given a protocol P, we say that the action labeled β *follows the action* labeled α in P if β is sequentially after α, i.e. in case α is an output, we have that $\mathsf{out}^\alpha(c,\cdot).Q$ is a subprocess of P and β occurs in Q. Given an action labeled α occurring in a protocol P, we denote by $\sigma^\alpha_{\mathsf{match}}$ the mgu of all the equations $u = v$ corresponding to an instruction $\mathsf{match}\ u\ \mathsf{with}\ v \to \_$ encountered from the root of P to its action labeled α. Note that for a protocol P that does not feature match construction, we have that $\sigma^\alpha_{\mathsf{match}} = \emptyset$ for any label α occurring in it.

Example 20. *Going back to Example 19, we have that β follows α in P, and $\sigma^\beta_{\mathsf{match}} = \{y \mapsto \mathsf{senc}(\langle \mathsf{req}, x\rangle, k)\}$.*

The inclusion $S \subseteq^\# S'$ denotes the fact that for any element $e \in S$, the multiplicity of e in S is smaller than the multiplicity of e in S'.

Lemma 11. *Let (β, p) be a marked position of a protocol P w.r.t. a typing system (Δ_0, δ_0) and α a label of an action in P involving term v such that:*

- $\mathsf{out}^\beta(c,u)$ *follows the action α in P;*
- $((u\sigma^\beta_{\mathsf{match}})|_p, p)\#S \in \rho(u\sigma^\beta_{\mathsf{match}})$ *for some S;*
- $((u\sigma^\beta_{\mathsf{match}})|_p, q)\#S' \in \rho(v\sigma^\beta_{\mathsf{match}})$ *for some q, S' such that $S' \subseteq^\# S$.*

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

Proof. We consider a well-typed execution $P \xRightarrow{\mathsf{tr}} (\mathcal{P}; \phi; \sigma; i)$ and $\mathsf{out}^\beta(c, \mathsf{w})$ occurring in tr. Let R be a recipe in forced normal form such that $\mathsf{root}(R) \in \Sigma_{\mathsf{d}} \cup \mathcal{W}$ 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 $\eta = 1 \dots 1$ be a sequence of 1 and $R_0 = R|_\eta$ be such that $\mathsf{target}(R_0) = p$. We have that $R_0\phi\!\downarrow = \mathsf{w}\phi|_p$, and we denote by $\{R^1_{\mathsf{key}}, \dots, R^j_{\mathsf{key}}\}$ the multiset of recipes occurring at position of the form $\eta.k$ with $k \geq 2$ in R_0.

According to our hypothesis, we know that there exist $\mathsf{in}^\alpha(\_, R_{in})$ (resp. $\mathsf{out}^\alpha(\_, \mathsf{w}')$) occurring before $\mathsf{out}^\beta(c, \mathsf{w})$ in tr, and a substitution τ such that $\mathsf{w}\phi = u\sigma^\beta_{\mathsf{match}}\tau$ and $R_{in}\phi\!\downarrow = v\sigma^\beta_{\mathsf{match}}\tau$ in case α corresponds to an input (resp. $\mathsf{w}'\phi = v\sigma^\beta_{\mathsf{match}}\tau$ in case α corresponds to an output). We know that p is a position that exists in $u\sigma^\beta_{\mathsf{match}}$ and similarly the position q exists in $v\sigma^\beta_{\mathsf{match}}$. Moreover, thanks to our hypothesis, we have that $(v\sigma^\beta_{\mathsf{match}})|_q = (u\sigma^\beta_{\mathsf{match}})|_p$. Thus, we have that $R_{in}\phi\!\downarrow|_q = \mathsf{w}\phi|_p$ in case α corresponds to an input (resp. $\mathsf{w}'\phi|_q = \mathsf{w}\phi|_p$ in case α corresponds to an output).

Now, reusing some elements of the multiset $\{R^1_{\mathsf{key}}, ..., R^j_{\mathsf{key}}\}$, we can build a recipe starting from R_{in} (resp. w') and adding destructors in order to extract the subterm at position q in $R_{in}\phi\!\downarrow$ (resp. $\mathsf{w}'\phi$) using elements of the multiset $\{R^1_{\mathsf{key}}, ..., R^j_{\mathsf{key}}\}$. 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 $1...1.k$ with $k \geq 2$ in $\overline{R_0}$, they form a submultiset of those occurring at position $1...1.k$ with $k \geq 2$ 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 $\mathsf{w}\phi|_p$ ($\overline{R_0}$ being smaller), and therefore R (recipe which contains R_0 as a subterm) is not an asap recipe for $R\phi\!\downarrow = \mathsf{w}\phi|_{p.p'}$. Indeed $R[\overline{R_0}]_\eta$ is smaller than R_0 and deduce the same term $\mathsf{w}\phi|_{p.p'}$. □

C Termination

In the previous sections, we considered a mathematical function dep. This function maps labels and types on sets of multisets. It may happen that the multisets are infinite, and in this case it becomes impossible to bound the number of sessions. The purpose of this section is to define a terminating algorithm dep', which computes the function dep whenever its result is finite, and which returns $\bot$ when it is infinite.

For the sake of clarity, we remark the equations satisfied by dep can be rewritten as:

$$\mathsf{dep}(\alpha) = A(\alpha) \cup \bigcup_i \left[B(\alpha) \otimes \left(\otimes_{j=0}^{k_i} \mathsf{dep}(\beta_i^j)\right)\right]$$

where α is a label or a type, $A(\alpha)$ and $B(\alpha)$ are sets depending only on α (without any call to dep) and β_i^j are labels or types. More precisely, $A(\alpha) \in \{\emptyset; \{\emptyset\}\}$ for any label and type α (and $A(\alpha) = \mathsf{dep}^0(\alpha)$), $B(\alpha) = \{\{\alpha\}\}$ for any label α, and $B(\tau) = \{\emptyset\}$ for any type τ. Those definitions can be rewritten as:

$$\mathsf{dep}(\alpha) = A(\alpha) \cup \left[B(\alpha) \otimes \bigcup_i \otimes_{j=0}^{k_i} \mathsf{dep}(\beta_i^j))\right]$$

Later on, it will be convenient to consider

$$\mathsf{next}(\alpha) = \{\beta_i^j \mid \forall i, j\}.$$

Example 21. *For example, when τ is an initial type, we have:*

$$\mathsf{dep}(\tau) = \mathsf{dep}^0(\tau) \cup \bigcup_{\substack{\mathsf{out}^\alpha(c,u)\ \text{occurring in } P \\ (\tau,p)\#\{\tau_1,\dots,\tau_n\} \in \rho(\delta_0(u)) \\ (\alpha,p)\ \text{not marked}}} \left(\mathsf{dep}(\alpha) \otimes \mathsf{dep}(\tau_1) \otimes \dots \otimes \mathsf{dep}(\tau_n)\right)$$

We have $A(\tau) = \mathsf{dep}^0(\tau)$, and if we denote $\{\beta_i^1, \dots, \beta_i^{k_i} \mid i\}$ the set of the $\alpha, \tau_1, \dots, \tau_n$ such that $\mathsf{out}^\alpha(c,u)$ occurs in P, $(\tau,p)\#\{\tau_1,\dots,\tau_n\} \in \rho(\delta_0(u))$, (α,p) is not marked, and $B(\tau) = \{\emptyset\}$ we have:

$$\mathsf{dep}(\tau) = A(\tau) \cup [B(\tau) \otimes \bigcup_i \otimes_{j=0}^{k_i} \mathsf{dep}(\beta_i^j)]$$

A *path* of labels and types is a sequence $\alpha_1 \dots . \alpha_n$ where each α_i is a type or a label. We denote $|p|$ the length of the path p (number of types or labels occuring in p). Our goal is to define a practical dep' that terminates and computes dep, but before doing that we need to identify all the labels and types α such that $\mathsf{dep}(\alpha) = \emptyset$.

A. How to compute the empty dependencies

We start with an example showing that this task it is not obvious.

Example 22. *Let* $P = \mathsf{new}\, s.\mathsf{in}^{\alpha}(c,s).\mathsf{out}^{\beta}(c,s)$. *We have that* $\mathsf{dep}(\delta_0(s)) = \emptyset$. *Indeed, consider the equations for* dep *(note that* $\delta_0(s)$ *is a cv-alien type as there are no variables or constants in* P*) with* $i > 0$.

$$\begin{aligned}\mathsf{dep}^i(\delta_0(s)) &= \mathsf{dep}^{i-1}(\beta)\\ \mathsf{dep}^i(\beta) &= \{\{\beta\}\} \otimes \mathsf{dep}^{i-1}(\alpha)\\ \mathsf{dep}^i(\alpha) &= \{\{\alpha\}\} \otimes \mathsf{dep}^{i-1}(\delta_0(s)) \otimes \mathsf{dep}^{i-1}(\bot)\end{aligned}$$

We have that $\mathsf{dep}^{i-1}(\bot) = \{\emptyset\}$ *and that* $\mathsf{dep}^0(\delta_0(s)) = \mathsf{dep}^0(\alpha) = \mathsf{dep}^0(\beta) = \emptyset$. *Thus,* $\mathsf{dep}^i(\delta_0(s)) = \emptyset$ *for each* i. *We can check that it is a solution of the following equation (resulting from the defining equations above).*

$$\mathsf{dep}(\delta_0(s)) = \{\{\beta\}\} \otimes \{\{\alpha\}\} \otimes \mathsf{dep}(\delta_0(s))$$

It confirms that it is a correct solution. So, there is a type $\delta_0(s)$ *such that the recursive calls defined by the equations loop, but* $\mathsf{dep}(\delta_0(s)) = \emptyset$.

To find the labels and types α such that $\mathsf{dep}(\alpha) = \emptyset$, we define the algorithm Empty. This algorithm takes (α, p) as input, where α is a label or type and p is a path (actually a sequence of labels), and returns $b \in \{\mathsf{True}, \mathsf{False}\}$. Intuitively, $\mathsf{Empty}(\alpha, p)$ is used to know if $\mathsf{dep}(\alpha)$ is empty. We also define auxiliary functions Paths and $m(\alpha, p)$. We want to prove that $\mathsf{Empty}(\alpha, p)$ tells if $\mathsf{dep}(\alpha) = \emptyset$. $\mathsf{Paths}(\alpha, \epsilon)$ represents the set of paths explored during the execution of $\mathsf{Empty}(\alpha, \epsilon)$ while $m(\alpha, p)$ is the number of remaining (inductive) steps required to reach a result. They are defined as:

- If $A(\alpha) \neq \emptyset$ then $\mathsf{Empty}(\alpha, p)$ returns False, $\mathsf{Paths}(\alpha, p)$ returns $\{p.\alpha\}$ and $m(\alpha, p) = 0$.
- Else, if $\alpha \in p$ then $\mathsf{Empty}(\alpha, p)$ returns True, $\mathsf{Paths}(\alpha, p)$ returns $\{p.\alpha\}$, and $m(\alpha, p) = 0$.
- Else set b as the evaluation of formula $\wedge_i \vee_{j=0}^{k_i} \mathsf{Empty}(\beta_i^j, p.\alpha)$ for $\beta_i^j \in \mathsf{next}(\alpha)$, set E as $E = \{p.\alpha\} \cup \cup_{i,j}(\mathsf{Paths}(\beta_i^j, p.\alpha))$. $\mathsf{Empty}(\alpha, p)$ returns b, $\mathsf{Paths}(\alpha, p)$ returns E, and $m(\alpha, p) = 1 + \max_{i,j}(m(\beta_i^j, p.\alpha))$.

Note that, as there are only a finite number of labels and types, and as the paths cannot contain twice the same label and type, those algorithms terminate. Note that all elements of $\mathsf{Paths}(\alpha, p)$ are of length at least 1. We prove the following lemma.

Lemma 12. *For any label or type* α *and any path* p*, we have* $\mathsf{Empty}(\alpha, p) = \mathsf{True}$ *iff* $\mathsf{dep}^i(\alpha) = \emptyset$ *for any* $i \leq m(\alpha, p)$.

Proof. We do the proof by induction on $m(\alpha, p)$.

Base case. If $m(\alpha, p) = 0$, then either $\mathsf{dep}^0(\alpha) = A(\alpha) \neq \emptyset$, and in this case $\mathsf{Empty}(\alpha, p) = \mathsf{False}$, or $A(\alpha) = \emptyset$ and thus $\mathsf{dep}^0(\alpha) = \emptyset$ and $\mathsf{Empty}(\alpha, p) = \mathsf{True}$. It proves the base case.

Inductive case. Let n be an integer. Assume that the result is true for any α, p such that $m(\alpha, p) \leq n$. Let α, p be such that $m(\alpha, p) = n + 1$. Then (as $m(\alpha, p) = n + 1 > 0$):

$$\mathsf{Empty}(\alpha, p) = \wedge_i(\vee_j \mathsf{Empty}(\beta_i^j, p.\alpha)) \tag{3}$$

for $\beta_i^j \in \mathsf{next}(\alpha)$ and we have (as $A(\alpha) = \emptyset$ since $m(\alpha, p) > 0$):

$$\mathsf{dep}^{n+1}(\alpha) = B(\alpha) \otimes \bigcup_i \otimes_j \mathsf{dep}^n(\beta_i^j)$$

But $B(\alpha) \neq \emptyset$, so $\mathsf{dep}^{n+1}(\alpha) = \emptyset$ if the following formula ϕ evaluates as True.

$$\phi = \wedge_i \vee_j [\mathsf{dep}^n(\beta_i^j) = \emptyset]$$

By induction hypothesis, we have that $\mathsf{dep}^n(\beta_i^j) = \emptyset$ iff $\mathsf{Empty}(\beta_i^j, p.\alpha) = \mathsf{True}$. Thus formula (3) evaluates as True iff ϕ evaluates as True. Thus $\mathsf{dep}^{n+1}(\alpha) = \emptyset$ iff $\mathsf{Empty}(\alpha, p) = \mathsf{True}$. As $\mathsf{dep}^i(\alpha) \subseteq \mathsf{dep}^{n+1}(\alpha)$ for any $i \leq n + 1$, it concludes the proof. □

Lemma 13. *Let* α *be a label or type. If* $p \in \mathsf{Paths}(\alpha, \epsilon)$ *then for any prefix* $q \neq \epsilon$ *of* p *we have* $q \in \mathsf{Paths}(\alpha, \epsilon)$.

Proof. We prove the following result by induction on $m(\alpha, p_0)$: if $p \in \mathsf{Paths}(\alpha, p_0)$, then $p = p_0.\alpha.q$, and for any prefix q' of q, we have $p_0.\alpha.q' \in \mathsf{Paths}(\alpha, p_0)$.

Base case. Assume $m(\alpha, p_0) = 0$. Then $\mathsf{Paths}(\alpha, p_0) = \{p_0.\alpha\}$. Let $p \in \mathsf{Paths}(\alpha, p_0)$. We have $p = p_0.\alpha$ (so $q = \epsilon$). For any prefix q' of q, we have $q' = \epsilon$ and $p_0.\alpha.q' = p \in \mathsf{Paths}(\alpha, p_0)$.

Inductive case. Assume the result is true for any α, p_0 such that $m(\alpha, p_0) \leq n$. Let α, p_0 be such that $m(\alpha, p_0) = n + 1 > 0$. So we have that $\mathsf{Paths}(\alpha, p_0) = \{p_0.\alpha\} \cup \cup_{i,j}\mathsf{Paths}(\beta_i^j, p_0.\alpha)$ for $\beta_i^j \in \mathsf{next}(\alpha)$. Let $p \in \mathsf{Paths}(\alpha, p_0)$. Either $p = p_0.\alpha$, and the proof of the base case applies; or $p \in \mathsf{Paths}(\beta_i^j, p_0.\alpha)$ for some $\beta_i^j \in \mathsf{next}(\alpha)$. We have that $m(\beta_i^j, p_0.\alpha) < m(\alpha, p_0) = n + 1$ thus $m(\beta_i^j, p_0.\alpha) \leq n$ and induction hypothesis applies. So $p = p_0.\alpha.\beta_i^j.q$ and for any prefix q' of q, we have:

$$p = p_0.\alpha.\beta_i^j.q' \in \mathsf{Paths}(\beta_i^j, p_0.\alpha)$$

Let q' be a prefix of $\beta_i^j.q$. Either $q' = \beta_i^j.q''$ and we get:

$$p = p_0.\alpha.q' \in \mathsf{Paths}(\beta_i^j, p_0.\alpha) \subseteq \mathsf{Paths}(\alpha, p_0)$$

Or $q' = \epsilon$ and $p = p_0.\alpha$ which has been handled before. This concludes the inductive case.

If we apply the above result to $p \in \mathsf{Paths}(\alpha, \epsilon)$, we get that $p = \alpha.q$ and that for any prefix q' of q, $\alpha.q' \in \mathsf{Paths}(\alpha, p_0)$. As α is a prefix of any non-empty prefix of p, we get the result. □

The following lemma aims to help us proving that there Empty has the expected property of indicating the empty results. However, it is not always true: for example, $\mathsf{Empty}(\alpha, \alpha) = \mathsf{True}$ just by definition of Empty, and even if $\mathsf{dep}(\alpha) \neq \emptyset$. So, we have to restrict ourselves to paths

that actually occur inside a computation, that is elements of $\mathsf{Paths}(\alpha,\epsilon)$. But even for those paths, it might be that $\mathsf{Empty}(\alpha,\epsilon) = \mathsf{True}$, but that $\mathsf{Empty}(\alpha,p) = \mathsf{False}$ for some path p, as examplified below.

Example 23. *Consider equations* $\mathsf{dep}(\alpha) = \mathsf{dep}(\gamma)$, $\mathsf{dep}(\gamma) = \mathsf{dep}(\alpha) \cup \mathsf{dep}(\beta)$ *and* $\mathsf{dep}(\beta) \neq \emptyset$ *(e.g.* $\mathsf{dep}(\beta) = \{\emptyset\}$*), The corresponding set of paths is:*

$$\mathsf{Paths}(\alpha,\epsilon) = \{\alpha, \alpha.\gamma, \alpha.\gamma.\beta, \alpha.\gamma.\alpha\}$$

We have that $\mathsf{Empty}(\alpha,\alpha.\gamma) = \mathsf{True}$ *but* $\mathsf{Empty}(\alpha,\epsilon) = \mathsf{False}$ *and we get that* $\mathsf{dep}(\alpha) \neq \emptyset$ *as it contains* $\mathsf{dep}(\beta)$.

Thus, we have to handle the case where $\mathsf{Empty}(\alpha,p)$ is true, but some other computations will make that $\mathsf{Empty}(\alpha,\epsilon)$ is false, or more generally $\mathsf{Empty}(\alpha,q) = \mathsf{False}$ for some prefix q of p. This is done by possible conclusion (b) in the following lemma. This possible conclusion will be impossible anyway when the lemma is called in Proposition 2 with $p = \epsilon$.

Lemma 14. *Let* α *be a label or a type. Let* $p.\beta \in \mathsf{Paths}(\alpha,\epsilon)$. *If* $\mathsf{Empty}(\beta,p) = \mathsf{True}$*, then (a)* $\mathsf{dep}(\beta) = \emptyset$ *or (b)* $p = p_0.\gamma.p_1$ *for some* p_0,γ,p_1 *such that* $\mathsf{Empty}(\gamma,p_0) = \mathsf{False}$.

Proof. First, we prove the following result by induction on k_0: Let $p.\beta \in \mathsf{Paths}(\alpha,\epsilon)$. If $\mathsf{Empty}(\beta,p) = \mathsf{True}$, then (a) $\mathsf{dep}^k(\beta) = \emptyset$ for any k such that $k + |p| \leq k_0$ or (b) $p = p_0.\gamma.p_1$ for some p_0,γ,p_1 such that $\mathsf{Empty}(\gamma,p_0) = \mathsf{False}$.

Base case. Let $k_0 = 1$ and $p.\beta \in \mathsf{Paths}(\alpha,\epsilon)$. Let k such that $k + |p| \leq k_0$. Either $k = k_0$, or $k < k_0$.

- If $k = k_0$, then $|p| = 0$ hence $p = \epsilon$. So by hypothesis $\mathsf{Empty}(\beta,\epsilon) = \mathsf{True}$, and as $\beta \notin \epsilon$, by definition of $m(\beta,\epsilon)$, we must have $m(\beta,\epsilon) \geq 1$. Lemma 12 applies and we get that $\mathsf{dep}^k(\beta) = \emptyset$ as $k = k_0 = 1 \leq m(\beta,\emptyset)$.
- If $k < k_0$, then $k = 0$ hence $k \leq m(\beta,p)$. Thus by Lemma 12, we have that $\mathsf{dep}^k(\beta,p) = \emptyset$.

Thus condition (a) is satisfied.

Inductive case. Let k_0 be such that the result is true for k_0. Assume toward contradiction that the result is not true for $k_0 + 1$. We can consider a minimal k such that there is a $p.\beta \in \mathsf{Paths}(\alpha,\epsilon)$ for which $k + |p| = k_0 + 1$, $\mathsf{Empty}(\beta,p) = \mathsf{True}$ but $\mathsf{dep}^k(\beta) \neq \emptyset$ and there is no p_0,γ,p_1 such that $p = p_0.\gamma.p_1$ and $\mathsf{Empty}(\gamma,p_0) = \mathsf{False}$. We consider the cases in the computation of $\mathsf{Empty}(\beta,p)$.

- If $A(\beta) \neq \emptyset$, then $\mathsf{Empty}(\beta,p) = \mathsf{False}$ so it is impossible.
- If $A(\beta) = \emptyset$ and $\beta \in p$, then $\mathsf{Empty}(\beta,p) = \mathsf{True}$ and $p = p_0.\beta.p_1$. Either $\mathsf{Empty}(\beta,p_0) = \mathsf{True}$, and by Lemma 13 induction hypothesis applies. Property (a) gives us a contradiction as $k+|p_0| < k+|p| \leq k_0+1$ (thus $k + |p_0| \leq k_0$) and property (b) gives us a contradiction as $p_0 = q_0.\gamma.q_1$ implies that $p = q_0.\gamma.q_1.p_1$, and we have $\mathsf{Empty}(\gamma,q_0) = \mathsf{False}$; or $\mathsf{Empty}(\beta,p_0) = \mathsf{False}$ and there is a contradiction.
- If $A(\beta) = \emptyset$ and $\beta \notin p$, then for $\beta_i^j \in \mathsf{next}(\alpha)$:

  $$\mathsf{Empty}(\beta,p) = \wedge_i(\vee_i \mathsf{Empty}(\beta_i^j, p.\beta))$$

  By hypothesis, we have $\mathsf{Empty}(\beta,p) = \mathsf{True}$. Assume that one of the $\mathsf{Empty}(\beta_i^j,p.\beta) = \mathsf{True}$ and has property (b), that is there is a γ, a p_0 and a p_1 such that $p.\beta = p_0.\gamma.p_1$ and $\mathsf{Empty}(\gamma,p_0) = \mathsf{False}$. We know that $(\gamma,p_0) \neq (\beta,p)$ as $\mathsf{Empty}(\beta,p) = \mathsf{True}$. So $p = p_0.\gamma.p_1'$ and $p_1 = p_1'.\beta$, so we get property (b) which is a contradiction.
  So from now we can assume ($\star$) that for each $\beta_i^j \in \mathsf{next}(\beta)$, either $\mathsf{Empty}(\beta_i^j,p.\beta) = \mathsf{False}$ or property (b) is false.
  We also have (as $k > 0$ and $A(\alpha) = \emptyset$):

  $$\mathsf{dep}^k(\beta) = B(\beta) \cup_i \otimes_j \mathsf{dep}^{k-1}(\beta_i^j, p.\beta)$$

  So $\mathsf{dep}^k(\beta) = \emptyset$ iff the following formula ϕ evaluates as True.

  $$\phi = \wedge_i(\vee_j(\mathsf{dep}^{k-1}(\beta_i^j) = \emptyset))$$

  But by minimality of k and by our assumption ($\star$), for each β_i^j such that $\mathsf{Empty}(\beta_i^j,p.\beta) = \mathsf{True}$, we have that $\mathsf{dep}^k(\beta) = \emptyset$ as $k - 1 + |p.\beta| = k + |p| = k_0 + 1$. As $\mathsf{Empty}(\beta,p) = \mathsf{True}$, we have that for each i there exists a j such that $\mathsf{Empty}(\beta_i^j,p.\beta) = \mathsf{True}$. It implies that for each i there exists a j such that $\mathsf{dep}(\beta_i^j) = \emptyset$ and thus that ϕ is true, so $\mathsf{dep}^k(\beta) = \emptyset$.

It concludes the proof of the inductive case.

Now remark that $\mathsf{dep}(\beta) = \cup_k \mathsf{dep}^k(\beta)$ so case (a) rewrites as $\mathsf{dep}(\beta) = \emptyset$ as it is true for any k_0. □

We prove the expected result that the Empty algorithm determines the α such that $\mathsf{dep}(\alpha) = \emptyset$.

Proposition 2. *Let* α *be a label or a type. Then* $\mathsf{dep}(\alpha) = \emptyset$ *iff* $\mathsf{Empty}(\alpha,\epsilon) = \mathsf{True}$.

Proof. If $\mathsf{dep}(\alpha) = \emptyset$, then $\mathsf{dep}^k(\alpha) = \emptyset$ for any k by definition of dep, and thus $\mathsf{Empty}(\alpha,\epsilon) = \mathsf{True}$ by Lemma 12, as $m(\alpha,\epsilon)$ is finite.

Conversely, remark that $\alpha \in \mathsf{Paths}(\alpha,\epsilon)$ by definition of Paths (more generally, $p.\alpha \in \mathsf{Paths}(\alpha,p)$ for each α,p). By hypothesis, we have $\mathsf{Empty}(\alpha,\epsilon) = \mathsf{True}$. Thus Lemma 14 applies and we have (a) $\mathsf{dep}(\alpha) = \emptyset$, as (b) is impossible. □

B. How to compute dep *in practice (*dep′*)*

In this section, we define algorithm dep' and we prove it coincides with dep. More precisely, dep' always terminates, and it returns either $\perp$ when dep contains an infinite multiset; or the same result as dep when it is a finite set of finite multisets. Algorithm dep' uses Empty to avoid looping on empty dependencies.

Algorithm dep'*.:* We define two operators $\tilde{\otimes}$ and $\tilde{\cup}$ such that:

- $E \tilde{\otimes} F = E \otimes F$ when E and F are sets.
- $E \tilde{\cup} F = E \cup F$ when E and F are sets.
- $\emptyset \tilde{\otimes} \perp = \perp \tilde{\otimes} \emptyset = \emptyset$
- $E \tilde{\otimes} \perp = \perp \tilde{\otimes} E = \perp$ when E is a set and $E \neq$.
- $E \tilde{\cup} \perp = \perp \tilde{\cup} E = \perp$ when E is a set (even if $E = \emptyset$).

Intuitively, $\perp$ represents a set that has an infinite multiset as an element.

Then, for α a label or a type, we define $\mathsf{dep}'(\alpha, p)$ and $m'(\alpha, p)$ as:

- $\mathsf{dep}'(\alpha, p) = \emptyset$, $m'(\alpha, p) = 0$ if $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{True}$;
- $\mathsf{dep}'(\alpha, p) = \bot$, $m'(\alpha, p) = 0$ if α occurs in p and $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{False}$;
- Otherwise, as:

$$\mathsf{dep}'(\alpha, p) = A(\alpha) \tilde{\cup} \big[B(\alpha) \tilde{\otimes} \tilde{\bigcup}_i \big(\tilde{\otimes}_{j=0}^{k_i} \mathsf{dep}'(\beta_i^j, p.\alpha) \big) \big]$$

and $m'(\alpha, p) = 1 + \max_{i,j} m'(\beta_i^j, p.\alpha)$.

The computation of dep' terminates as there are only a finite number of labels and types, so the length of the paths p is bounded (there can be no repetition in p). Moreover, if $\mathsf{dep}'(\alpha, \emptyset) \neq \bot$, it is finite and each of the multisets included in $\mathsf{dep}'(\alpha, \emptyset)$ are finite as they are computed in a finite number of steps.

Our goal is to prove that $\mathsf{dep}'(\alpha, \emptyset) = \mathsf{dep}(\alpha)$ if $\mathsf{dep}(\alpha)$ is finite, and $\mathsf{dep}'(\alpha, \emptyset) = \bot$ if $\mathsf{dep}(\alpha)$ is infinite (for any type or label α).

Lemma 15. *For any label or type α, for any path p, if $\mathsf{dep}'(\alpha, p) = \emptyset$ then $\mathsf{Empty}(\alpha, p) = \mathsf{True}$.*

Proof. We prove the result by induction on $m(\alpha, p)$.

Base case. $m(\alpha, p) = 0$. If $\alpha \notin p$, we have $A(\alpha) \neq \emptyset$ by definition of $m(\alpha, p)$ and as $A(\alpha) \subseteq \mathsf{dep}'(\alpha, p)$, it implies $\mathsf{dep}'(\alpha, p) \neq \emptyset$ which is impossible. So $\alpha \in p$ and $\mathsf{Empty}(\alpha, p) = \mathsf{True}$.

Inductive case. We assume that the result is true for any $m(\alpha, p) \leq n$ and we want to prove it is true for $m(\alpha, p) = n+1$. Let α, p be such that $m(\alpha, p) = n+1 > 0$. By definition of $m(\alpha, p)$, we know that (with $\mathsf{next}(\beta) = \{\beta_i^j \mid i, j\}$):

$$\mathsf{Empty}(\alpha, p) = \wedge_i \vee_j \mathsf{Empty}(\beta_i^j, p.\beta)$$

Moreover, as $\mathsf{dep}'(\alpha, p) = \emptyset$, we have either $\alpha \in p$ and $\mathsf{Empty}(\alpha, p) = \mathsf{True}$ (which is the result); or:

$$\mathsf{dep}'(\alpha, p) = A(\alpha) \tilde{\cup} \big[B(\alpha) \tilde{\otimes} \tilde{\bigcup}_i \big(\tilde{\otimes}_{j=0}^{k_i} \mathsf{dep}'(\beta_i^j, p.\alpha) \big) \big]$$

As $\mathsf{dep}'(\alpha, p) = \emptyset$, $A(\alpha) = \emptyset$, and:

$$\mathsf{Empty}(\alpha, p) = \wedge_i \vee_j \mathsf{Empty}(\beta_i^j, p.\beta)$$

For each i, j, we have that $m(\beta_i^j, p.\beta) < m(\alpha, p) = n + 1$. Therefore induction hypothesis applies and for each $\mathsf{dep}'(\beta_i^j, p.\alpha) = \emptyset$ we have $\mathsf{Empty}(\beta_i^j, p.\alpha) = \mathsf{True}$. As $\mathsf{dep}'(\alpha, p) = \emptyset$, for each i there is a j such that $\mathsf{dep}'(\beta_i^j, p.\alpha) = \emptyset$. Thus for each i there is a j such that $\mathsf{Empty}(\beta_i^j, p.\alpha) = \mathsf{True}$. It implies that $\mathsf{Empty}(\alpha, p) = \mathsf{True}$. □

From this, we can deduce the following lemma.

Lemma 16. *Let α be a label or a type. We have that $\mathsf{dep}'(\alpha, \epsilon) = \emptyset$ iff $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{True}$ iff $\mathsf{dep}(\alpha) = \emptyset$.*

Proof. By Proposition 2, we have that $\mathsf{dep}(\alpha) = \emptyset$ iff $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{True}$. By Lemma 15, we have that $\mathsf{dep}'(\alpha, \epsilon) = \emptyset$ implies $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{True}$. By definition of dep', we have that $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{True}$ implies $\mathsf{dep}'(\alpha, \epsilon) = \emptyset$. It concludes the proof. □

Lemma 17. *If p is a suffix of q, then $m'(\alpha, p) \geq m'(\alpha, q)$.*

Proof. We do the proof by induction on $m'(\alpha, p)$. If $m'(\alpha, p) = 0$, then either $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{True}$ or $\alpha \in p$ and thus $\alpha \in q$. Thus $m'(\alpha, q) = 0$ in both cases.

Now, assume the result is true for any α, p such that $m'(\alpha, p) \leq k$ for some $k \geq 0$. Let α, p such that $m'(\alpha, p) = k + 1$ and q such that p is a suffix of q. As $m'(\alpha, p) = k + 1 > 1$, we have $m'(\alpha, p) = 1 + \max_{i,j} m'(\beta_i^j, p.\alpha)$. Either $m'(\alpha, q) = 0$ and thus $m'(\alpha, q) \leq m'(\alpha, p)$ or $m'(\alpha, q) > 0$ and $m'(\alpha, q) = 1 + \max_{i,j} m'(\beta_i^j, q.\alpha)$. We have that $m'(\beta_i^j, p.\alpha) \leq k$ for each i, j, and $p.\alpha$ is a suffix of $q.\alpha$ so induction hypothesis applies and $m'(\beta_i^j, p.\alpha) \geq m'(\beta_i^j, q.\alpha)$ for each i, j. We conclude that $m'(\alpha, q) \leq m'(\alpha, p)$. □

Proposition 3. *Let α be a label or a type. If $\mathsf{dep}'(\alpha, \epsilon) \neq \bot$, we have that $\mathsf{dep}'(\alpha, \epsilon) = \mathsf{dep}(\alpha)$.*

Proof. We prove that $\mathsf{dep}'(\alpha, p) = \mathsf{dep}(\alpha)$ for any p such that $\mathsf{dep}'(\alpha, p) \neq \bot$ by induction on $m'(\alpha, p)$.

Base case: $m'(\alpha, p) = 0$. In this case, by definition of $m'(\alpha, p)$ we have $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{True}$ as $\mathsf{dep}'(\alpha, p) \neq \bot$. Hence $\mathsf{dep}'(\alpha, \epsilon) = \emptyset$. By Lemma 16, we get $\mathsf{dep}(\alpha) = \emptyset$. By definition of $\mathsf{dep}'(\alpha, p)$, we also get that $\mathsf{dep}'(\alpha, p) = \emptyset$. It proves the result in this case.

Inductive case. Now, assume we have the result for any α such that $m'(\alpha, p) \leq k$ for some k. Then, let α such that $m'(\alpha, p) = k + 1$. We have that $m'(\alpha, \epsilon) \geq m'(\alpha, p)$ by Lemma 17. Hence, $m'(\alpha, \epsilon) \neq 0$. We deduce that $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{False}$ by definition of $m'(\alpha, \epsilon)$. Moreover, let p be such that $\mathsf{dep}'(\alpha, p) \neq \bot$. As $\mathsf{Empty}(\alpha, \epsilon) = \mathsf{False}$, we have:

$$\mathsf{dep}'(\alpha, p) = A(\alpha) \tilde{\cup} \big[B(\alpha) \tilde{\otimes} \tilde{\bigcup}_i \big(\tilde{\otimes}_{j=0}^{k_i} \mathsf{dep}'(\beta_i^j, p.\alpha) \big) \big]$$

We have $m'(\beta_i^j, \alpha.p) < m'(\alpha, p) = k + 1$ for every i, j by definition of m'. So induction hypothesis applies and for any i, j, $\mathsf{dep}'(\beta_i^j, p.\alpha) = \mathsf{dep}(\beta_i^j)$ if $\mathsf{dep}'(\beta_i^j, p.\alpha) \neq \bot$.

As $\mathsf{dep}'(\alpha, p) \neq \bot$, for each i_0, j_0 such that $\mathsf{dep}'(\beta_{i_0}^{j_0}, p.\alpha) = \bot$, there must be a j_1 such that $\mathsf{dep}'(\beta_{i_0}^{j_1}, p.\alpha) = \emptyset$. As $\emptyset \neq \bot$, it implies by induction hypothesis that $\mathsf{dep}(\beta_{i_0}^{j_1}) = \emptyset$ and thus that $\otimes_j \mathsf{dep}(\beta_{i_0}^j) = \emptyset$. We deduce:

$$\otimes_j \mathsf{dep}(\beta_{i_0}^j) = \otimes_j \mathsf{dep}'(\beta_{i_0}^j, p.\alpha)$$

Given i_1 such that there is no j_0 such that $\mathsf{dep}'(\beta_{i_0}^j, p.\alpha) = \bot$, we also have (as the factors are equal):

$$\otimes_j \mathsf{dep}(\beta_{i_1}^j) = \otimes_j \mathsf{dep}'(\beta_{i_1}^j, p.\alpha)$$

Thus we deduce $\mathsf{dep}(\alpha) = \mathsf{dep}'(\alpha, p)$. It concludes the inductive case.

Conclusion. We have proved that for any α,p such that $\mathsf{dep}'(\alpha, p) \neq \bot$, we have $\mathsf{dep}'(\alpha, p) = \mathsf{dep}(\alpha)$. The result is the case where $p = \epsilon$. □

It remains to prove that if $\mathsf{dep}'(\alpha,\epsilon)=\bot$, $\mathsf{dep}(\alpha)$ is infinite. It is quite intuitive to remark that any $\bot$ returned by dep' corresponds to a loop in the computation of dep. However, the converse is not always true, as some of those loops can result in an empty result. To prove the converse of Proposition 3, we need to define loops that do not result in $\emptyset$. To do that, we introduce next' as $\mathsf{next}'(\alpha)=\{\beta_i^j \in \mathsf{next}(\alpha) \mid \forall k, \mathsf{Empty}(\beta_i^k,\epsilon)=\mathsf{False}\}$, that is those elements in $\mathsf{next}(\alpha)$ that occur in a non-empty product $\mathsf{dep}(\beta_i^1)\otimes\cdots\otimes\mathsf{dep}(\beta_i^{k_i})\neq\emptyset$. We can also remark that the definitions of dep and dep' can be rewritten with $\beta_i^j\in\mathsf{next}'(\alpha)$ as:

$$\mathsf{dep}(\alpha)=A(\alpha)\cup\big[B(\alpha)\otimes\big(\bigcup_i \mathsf{dep}(\beta_i^j)\big)\big]$$

and (when $\alpha\notin p$ and $\mathsf{Empty}(\alpha,\emptyset)=\mathsf{False}$):

$$\mathsf{dep}'(\alpha,p)=A(\alpha)\cup\big[B(\alpha)\otimes\big(\bigcup_i \mathsf{dep}'(\beta_i^j,p.\alpha)\big)\big]$$

Now, we can prove that each step of the computation of dep is non-decreasing in some way.

Lemma 18. *If $\beta\in\mathsf{next}'(\alpha)$ for some types or labels α and β, there is a set $E(\alpha.\beta)\neq\emptyset$ such that $E(\alpha.\beta)\otimes\mathsf{dep}(\beta)\subseteq\mathsf{dep}(\alpha)$.*

Proof. If $\beta\in\mathsf{next}'(\alpha)$, there is an equation:

$$\mathsf{dep}(\alpha)=A(\alpha)\cup\big[B(\alpha)\otimes\bigcup_i\big(\otimes_{j=0}^{k_i}\mathsf{dep}(\beta_i^j)\big)\big]$$

where β is among the β_i^j, say $\beta=\beta_0^0$. Then we have:

$$\begin{aligned}\mathsf{dep}(\alpha) \quad &= A(\alpha)\cup B(\alpha)\otimes\\ &\quad[\textstyle\bigcup_{i>0}(\otimes_{j=0}^{k_i}\mathsf{dep}(\beta_i^j))]\cup[\mathsf{dep}(\beta)\otimes(\otimes_{j=1}^{k_0}\mathsf{dep}(\beta_0^j))]\end{aligned}$$

If we write $E(\alpha.\beta)=B(\alpha)\otimes(\otimes_{j=1}^{k_0}\mathsf{dep}(\beta_0^j))$, we have $E(\alpha.\beta)\otimes\mathsf{dep}(\beta)\subseteq\mathsf{dep}(\alpha)$. As the only types or labels τ such that $\mathsf{dep}(\tau)=\emptyset$ are those such that $\mathsf{Empty}(\tau,\epsilon)=\mathsf{True}$ by Proposition 2, and as they do not occur in the equation by definition of next', we have that $E(\alpha.\beta)\neq\emptyset$. □

When $E(\alpha.\beta)=\{\emptyset\}$, we have only proved that $\mathsf{dep}(\alpha)=\mathsf{dep}(\beta)$, which is not very helpful (recall that ultimately, we want to prove that a cycle in the computation means that the result must be infinite). Thus, we want to know exactly when this occurs.

Lemma 19. *If τ is a label or a type, and β a type or a label, $E(\tau.\beta)=\{\emptyset\}$ only if τ is a type and:*

- *either β is a type τ', and τ' is a direct subtype of τ;*
- *or β is an output label.*

Proof. If $E(\tau.\beta)=\{\emptyset\}$, then it means that:

$$\{\emptyset\}=E(\tau.\beta)=B(\tau)\otimes\otimes_{j=1}^{k}\mathsf{dep}(\beta_j)$$

for some $\beta_1,\dots\beta_k$. Thus in particular $B(\tau)=\{\emptyset\}$ which means that τ is a type (as we have $B(\alpha)=\{\alpha\}$ for any label α and $B(\tau)=\{\emptyset\}$ for any type τ). So the associated equation is either (if τ is an initial type):

$$\mathsf{dep}(\tau)=\mathsf{dep}^0(\tau)\cup\bigcup_{\substack{\mathsf{out}^\gamma(c,u)\text{ occurring in }P\\(\tau,p)\#\{\tau_1,\dots,\tau_n\}\in\rho(\delta_0(u))\\(\gamma,p)\text{ not marked}}}\big(\mathsf{dep}(\gamma)\otimes\mathsf{dep}(\tau_1)\otimes\dots\otimes\mathsf{dep}(\tau_n)\big)$$

or (if $\tau=\mathsf{f}(\tau_1,\dots,\tau_n)$):

$$\begin{aligned}\mathsf{dep}(\tau)=&\mathsf{dep}^0(\tau)\cup(\{\emptyset\}\otimes\mathsf{dep}(\tau_1)\otimes\cdots\otimes\mathsf{dep}(\tau_n))\\&\cup\bigcup_{\substack{\mathsf{out}^\gamma(c,u)\text{ occurring in }P\\(\tau,p)\#\{\tau_1,\dots,\tau_n\}\in\rho(\delta_0(u))\\(\gamma,p)\text{ not marked}}}\big(\mathsf{dep}(\gamma)\otimes\mathsf{dep}(\tau_1)\otimes\dots\otimes\mathsf{dep}(\tau_n)\big)\end{aligned}$$

We have that $\mathsf{dep}(\beta)$ occurs in a product $\otimes$ where all the other elements (denoted above as β_0^j) satisfy $\mathsf{dep}(\beta_0^j)=\{\emptyset\}$. But for the label γ of an output, we have $\mathsf{dep}(\gamma)=(\{\gamma\}\otimes D)\cup C$ for some sets of multisets $C\neq\emptyset$, $D\neq\emptyset$. Thus $\mathsf{dep}(\gamma)\neq\{\emptyset\}$. So β does not appear in a product where there is a label of an output. So either β is a direct subtype of τ, or β is itself the label of the output. □

We can extend inductively E to paths through $E(\alpha.\beta.p)=E(\alpha.\beta)\otimes E(\beta.p)$ for any labels or types α, $\beta\in\mathsf{next}'(\alpha)$ and any path $p=p_1.\dots.p_k$ where $p_1\in\mathsf{next}'(\beta)$ and $p_{i+1}\in\mathsf{next}(p_i)$ for each i with $1\leq i<k$. We are now ready to prove the main theorem of this section.

Theorem 6. *Let α be a label or a type. Let p be a path of label and types. Then, $\mathsf{dep}'(\alpha,\epsilon)$ terminates. Moreover, $\mathsf{dep}'(\alpha,\epsilon)\neq\bot$ iff $\mathsf{dep}(\alpha)$ is finite, and in this case $\mathsf{dep}(\alpha,\epsilon)=\mathsf{dep}(\alpha)$.*

Proof. The termination of dep' has been proved when dep' has been defined. It follows directly from the finiteness of the set of labels and types. The case where $\mathsf{dep}(\alpha,\epsilon)\neq\bot$ has been handled by Proposition 3.

It remains to prove that, when $\mathsf{dep}'(\alpha,\epsilon)=\bot$, the set $\mathsf{dep}(\alpha)$ is infinite. Let α be a label or a type. Assume that $\mathsf{dep}'(\alpha,\epsilon)=\bot$. Then there is a minimal path p and a β such that $\mathsf{dep}'(\alpha,\epsilon)$ has called $\mathsf{dep}'(\beta,p)=\bot$ with β occuring in p (only the calls to dep' on an already-explored type or label can create $\bot$ elements).

By Lemma 18 (applied inductively), we have that $\mathsf{dep}(\alpha)=E(\alpha.p.\beta)\otimes\mathsf{dep}(\beta)$. Moreover, as β occurs inside p, we can write $p=p_1.\beta.p_2$ and by Lemma 18 (applied inductively):

$$E(\beta.p_2.\beta)\otimes\mathsf{dep}(\beta)\subseteq\mathsf{dep}(\beta)$$

We have to prove that $E(\beta.p_2.\beta)\neq\{\emptyset\}$. It will prove that $\mathsf{dep}(\beta)$ contains an infinite multiset. It is sufficient to prove that there are two successive elements $e_1.e_2$ in the sequence $\beta.p_2.\beta$, such that $E(e_1.e_2)\neq\{\emptyset\}$. Assume toward contradiction that there are no such two successive elements in $\beta.p_2.\beta$. By Lemma 19, it means that for any $e_1.e_2$ in $\beta.p_2.\beta$, we have that e_1 is a type τ, and:

- either e_2 is a type τ', and τ' is a direct subtype of τ;
- or e_2 is an output label.

However, either $e_2 = \beta$ (it is the last element of the sequence), or (it is not the last element) there is an element e_3 such that $e_2.e_3$ occurs in $\beta.p_2.\beta$. In the second case, e_2 is a type. In the first case, $e_2 = \beta$, which is a type as it is the first element of the sequence. So in both cases, e_2 is a type, and we deduce it is a direct subtype of e_1. Thus $\beta.p_2.\beta$ is a sequence where each element is a direct subtype of the previous one. It implies that β is a strict subtype of itself, which is a contradiction. Thus, there are two successive elements $e_1.e_2$ in $\beta.p_2.\beta$ such that $E(e_1.e_2) \neq \{\emptyset\}$. It implies that $\mathsf{dep}(\beta)$ contains an infinite element, and thus as $\mathsf{dep}(\alpha) = E(\alpha.p_\perp(\alpha)) \otimes \mathsf{dep}(\beta)$, that α contains an infinite element. It is the result we wanted to prove. □

D THE CASE OF EQUIVALENCE

Regarding the class of protocols, we consider protocols *without else branches* and we consider processes in *simple form*. This corresponds to protocols that do not feature match constructions. The main reason for this restriction is the fact that despite our effort, we do not succeed to establish a typing result with else branch. Thus, this result for equivalence only deal with protocols without else branch. Since we are considering protocols without else branch, we can directly built on top of the typing result stated and proved in [14].

Then, following the proof done in [14], we establish our small bound in 2 main steps:

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, cv-alien-free, and involves only simple asap recipes.
2) We then compute a bound regarding the size of a well-typed attack of minimal size. We need for that to establish a new characterization regarding static inclusion.

A. A well-tped witness

As for the reachability case, we first show that we can focus on witnesses of non trace inclusion that have a particular form. A similar result has already been established in [21] considering a fixed set of primitives. The sketch of proof given below follows the exact same lines.

Theorem 7. *Let $\mathcal{K}_P$ be an initial configuration type-compliant w.r.t. (Δ_0, δ_0) 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 $\mathcal{K}_P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$ involving only simple asap recipes witnessing this fact. Moreover, we may assume that tr and ϕ are cv-alien- free.*

Proof. The first part of this theorem is actually a consequence of the typing result that has been established in [14] (Theorem 3.9). Indeed, let $\mathcal{K}_P$, and $\mathcal{K}_Q$ be two configurations as described in the theorem, and consider a witness tr of non-inclusion of minimal length. Thanks to Theorem 3.9 ([14]), we know that there exists a well-typed execution $\mathcal{K}_P \overset{\overline{\mathsf{tr}}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$ witnessing this non-inclusion and we have that tr and $\overline{\mathsf{tr}}$ have the same length. It thus remains to establish that we can consider such an execution which only involves simple asap recipes, and that we can assume that $\overline{\mathsf{tr}}$, and ϕ are cv-alien-free.

We first establish that we can consider a witness involving simple asap recipes. We denote i_0 the first step in the execution that involves a recipe that is not a simple asap one. In case such a i_0 does not exist, we are done. Otherwise, we consider a well-typed witness of non-inclusion such that $|\overline{\mathsf{tr}}| - i_0$ is minimal. We have that the $i_0{}^{\text{th}}$ step is an input of the form $\mathsf{in}(c, R)$ and R is not simple asap. Let $\overline{\mathsf{tr}}_{i_0}$ be the prefix of $\overline{\mathsf{tr}}$ until the $i_0{}^{\text{th}}$ step. We denote ϕ_{i_0} the frame such that $(\overline{\mathsf{tr}}_{i_0}, \phi_{i_0}) \in \mathsf{trace}(\mathcal{K}_P)$, and ψ_{i_0} the frame such that $(\overline{\mathsf{tr}}_{i_0}, \psi_{i_0}) \in \mathsf{trace}(\mathcal{K}_Q)$. Note that these frames are uniquely defined as we consider simple protocols.

Let R' be a simple asap recipe such that $R'\phi_{i_0}\downarrow = R\phi_{i_0}\downarrow$. Note that such a recipe exists thanks to Lemma 4. Since we consider a witness of non-inclusion of minimal length, we know that $\phi_{i_0} \sqsubseteq_s \psi_{i_0}$, and thus we have that $R\psi_{i_0}\downarrow = R'\psi_{i_0}\downarrow$. We consider the trace tr' which is equal to $\overline{\mathsf{tr}}$ replacing at the $i_0{}^{\text{th}}$ step, the recipe R with the recipe R'. We obtain a witness of non-inclusion which contradicts the minimality of the trace $\overline{\mathsf{tr}}$. Hence, we are done, and we can consider that $\mathcal{K}_P \overset{\mathsf{tr}}{\Longrightarrow} (\mathcal{P}; \phi; \sigma; i)$ is a well-typed execution involving only simple asap recipes witnessing the non-inclusion.

It remains to establish that we can assume that tr and ϕ are cv-alien-free. Actually, considering $\mathcal{K}_P \overset{\mathsf{tr}}{\Longrightarrow} \mathcal{K}'_P = (\mathcal{P}; \phi; \sigma; i)$ a witness of non-inclusion, according to Proposition 5.4 stated and proved in [14], 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.

We have that $St(\mathcal{K}_P\sigma_S) \subseteq St(\mathcal{K}_P)\sigma_S$ since σ_S is a mgu between subterms occurring in $\mathcal{K}_P$ by Lemma 6. Then, since ρ is a renaming, we deduce that $St(\mathcal{K}_P\sigma) \subseteq St(\mathcal{K}_P)\sigma$.

In order to conclude, we apply the same reasoning as in the case of reachability, as done e.g. at the end of Theorem 5. We first show that $\mathsf{tr}\phi\downarrow$ is cv-alien-free. Assume by contradiction that there exists a constant c_h from Σ_0 of cv-alien type occurring in $\mathsf{tr}\phi\downarrow$. In other words, c_h occurs in an instantatiation by σ of an input or output action of the initial processes, possibly after renaming names and variables (due to some unfolding). Thus, we have that $c_h \in St(\mathcal{K}_P\sigma)$. Thanks to the inclusion $St(\mathcal{K}_P\sigma) \subseteq St(\mathcal{K}_P)\sigma$, we deduce that $c_h \in St(\mathcal{K}_P)\sigma$, and since c_h does not occur in $St(\mathcal{K}_P)$ (as any constant from Σ_0 of cv-alien type), we deduce that there exists $x \in St(\mathcal{K}_P)$ such that $x\sigma = c_h$. However, this is impossible too since no variable having such a type can occur in $\mathcal{K}_P$. This allows us to conclude that $\mathsf{tr}\phi\downarrow$ is cv-alien-free.

We have that $\mathsf{Terms}(\phi) \subseteq \mathsf{Terms}(\mathsf{tr}\phi\downarrow)$, and thus we easily deduce that ϕ is cv-alien-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 from Σ_0 of cv-alien type occurs in ϕ and $\mathsf{tr}\phi\downarrow$, we are done. □

B. Bounding the length of a minimal witness

We have to explain how to bound the size of a (minimal) witness of trace inclusion, that is well-typed, cv-alien-free, and that only involve simple asap recipes. One additional difficulty is to establish a bound regarding static inclusion. For this, we characterize the form of the test involved in such a witness, and we rely on an alternative definition of static inclusion.

Definition 14. *Let ϕ, ψ be such that $dom(\phi) = dom(\psi)$. We write $\phi \sqsubseteq_s^{\mathsf{simple}} \psi$ if:*

1) *For each almost destructor-only and asap recipe R such that $R\phi\downarrow$ is a (resp. atomic) message, $R\psi\downarrow$ is a (resp. atomic) message.*
2) *For each $C[x_1, \ldots, x_n]$ a strict/direct subterm of a shape $\mathsf{sh_f}$ with $\mathsf{f} \in \Sigma_\mathsf{c}$, for each almost destructor-only and asap recipe R such that $R\phi\downarrow$ is a message. If $R\phi\downarrow = C[x_1, \ldots, x_n]\theta$ for some θ, then $R\psi\downarrow = C[x_1, \ldots, x_n]\theta'$ for some θ'.*
3) *For each simple recipe R and each almost destructor-only recipe R' such that $R\phi\downarrow, R'\phi\downarrow$ are messages, if $R\phi\downarrow = R'\phi\downarrow$, then we have that $R\psi\downarrow = R'\psi\downarrow$. Actually, we can assume in addition that one recipe is asap, and the other one is subterm asap w.r.t. ϕ.*
4) *For each rule $\mathsf{g}(t_1, \ldots, t_n) \to r$ in $\mathcal{R}_{\mathsf{test}} \cup \mathcal{R}_\mathsf{d}$, for each almost destructor-only and asap recipe R_1, and simple asap recipes $R_2, \ldots, R_n$ such that $R_1\phi\downarrow, \ldots, R_n\phi\downarrow$ are messages, if $(R_1\phi\downarrow, \ldots, R_n\phi\downarrow) = (t_1, \ldots, t_n)\theta$ for some θ, then $(R_1\psi\downarrow, \ldots, R_n\psi\downarrow) = (t_1, \ldots, t_n)\theta'$ for some θ'.*

This notion of static inclusion is equivalent to the original one.

Lemma 20. *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 3 are straightforward. Note that atomicity can be ckecked using the non-linear rule we have at our disposal. Given recipes $R_1, \ldots, R_n$ satisfying the assumptions of item 4, we have $\mathsf{g}(R_1, \ldots, , R_n)$ being a message is a test that holds ϕ, and thus it also holds for ψ thanks to our hypothesis, hence the result. Then, given a recipe R satisfying the assumptions of item 2, we can easily build a recipe R' applying f on top of R such that $R\phi\downarrow$ is a message. By hypothesis, we have that $R\psi\downarrow$ is a message too, and thus $R'\psi\downarrow = C[x_1, \ldots, x_n]\theta'$ since to be a message $R\psi\downarrow$ has to be compliant with the shape of f.

Thus, we only consider the other implication. To establish the other implication, we consider another alternative definition of static inclusion, denoted by $\sqsubseteq'_s$. This notion is the same than the one given in Definition 14 but considering arbitrary recipes instead of simple/almost destructor-only/asap/subterm asap 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 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 already been transferred from ϕ to ψ. Given a test T, we denote $R_1, \ldots, R_n$ the recipes that are used in that test T. We consider the measure $\mu(T) = (\mathcal{M}, n)$ where:

1) $\mathcal{M}$ is the multiset of variables occurring in recipes $R_1, \ldots R_n$ but we do not count recipes that are asap and simple;
2) $n = |R_1| + \ldots + |R_n|$ where $|R|$ is simply the size of R, i.e. the number of function symbols occurring in it.

We consider the lexicographic ordering. We now 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 forced normal form. Consider R' such that $R \twoheadrightarrow R'$. We assume that the rewriting step has been done at the innermost position p in R. We have that $R'\phi\downarrow$ is a message (Lemma 1). By induction hypothesis $R'\psi\downarrow$ is a message too. It remains to show that $R\psi\downarrow$ is a message. To show this, we will show that the same rewriting rule applies at root position on $R|_p\phi$ and $R|_p\psi$. We have that $R|_p = \mathsf{g}(R_1, \ldots, R_n)$ for some $\mathsf{g} \in \Sigma_{\mathsf{test}} \cup \Sigma_\mathsf{d}$. Since $R\phi\downarrow$ is a message, we know that $R|_p\phi\downarrow = \mathsf{g}(R_1\phi\downarrow, \ldots, R_n\phi\downarrow)\downarrow$ is a message too, and we know that a rewriting has occurred at root position, and thus atomicity conditions and equality conditions imposed by the rule were satisfied. In order to conclude, we apply the induction hypothesis (item 4) on recipes $R_1, \ldots, R_n$, we deduce that the same rewriting rule can be applied on the ψ side, and this allows us to conclude.
- Case where R is in normal form w.r.t. $\twoheadrightarrow$. Thanks to Lemma 3, we know that R is simple. Either R is almost destructor-only or $R = C[R_1, \ldots, R_k]$ with C a non-empty context built using symbols in Σ_c, and $R_1, \ldots, R_k$ almost destructor-only for any $1 \leq i \leq k$. In the latter case, we easily conclude relying on our induction hypothesis (item 1 and also item 2 for the shape restriction) applied on $R_1, \ldots, R_k$. In the former case, we know that R is almost destructor-only. We first assume that R is not subterm asap, meaning that there exists $1 \leq i \leq k$ such that R_i is not asap. Let R'_i be the asap recipe w.r.t. ϕ allowing us to deduce $R_i\phi\downarrow$. We have that $(R_i = R'_i)\phi\downarrow$, i.e. the test $R_i = R'_i$ holds in ϕ, and relying on our induction hypothesis, we deduce that $R_i = R'_i$ holds in ψ. Consider the recipe $R' = R[R'_i]_i$. We deduce that $R = R'$ holds in ψ. Applying our induction hypothesis on R', we deduce that $R'\psi\downarrow$ is a message which is atomic in case $R'\phi\downarrow$ is atomic, and thus we deduce that $R\psi\downarrow$ is

a message which is atomic in case $R\phi\downarrow$ is atomic too. Now, we asssume that R is subterm asap. First, in case R is asap, we are done. Thus, we know that R is an almost destructor-only recipe which is subterm asap but not asap. Let R' be a simple asap recipe w.r.t. ϕ such that $R' = R$ holds in ϕ. Thus by hypothesis we have that $R' = R$ which holds in ϕ also holds in ψ. Then, applying our induction hypothesis on R' (item 1), we deduce that $R'\psi\downarrow$ is a message (which is atomic in case $R'\phi\downarrow$ is atomic), and thus we easily conclude that $R\psi\downarrow$ is a message which is atomic in case $R\phi\downarrow$ is atomic too.

The test T is of the form R such that R is a recipe, $R\phi\downarrow = C[x_1, \ldots, x_n]\theta$ where $C[x_1, \ldots, x_n]$ is a strict direct subterm of a shape $\mathsf{sh_f}$.

- Case R is *not* in normal form w.r.t. $\twoheadrightarrow$. Let $R' = R\downarrow$. Since $R\phi\downarrow$ is a message, we deduce that $R'\phi\downarrow = R\phi\downarrow$ (Lemma 1). We have already seen (item 1) that $R\psi\downarrow$ is a message too and thus we have that $R'\psi\downarrow = R\psi\downarrow$. Since R holds in ϕ, i.e. $R\phi\downarrow = C[x_1, \ldots, x_n]\theta$ for some θ, we have that R' holds in ϕ, i.e. $R'\phi\downarrow = C[x_1, \ldots, x_n]\theta$ for the same θ. Relying on our induction hypothesis applied on test R' we deduce that the test R' holds in ψ, i.e. $R'\psi\downarrow = C[x_1, \ldots, x_n]\theta'$ for some θ', and thus we conclude that R holds in ψ, i.e. $R\psi\downarrow = C[x_1, \ldots, x_n]\theta'$.
- Otherwise, as in the previous case, thanks to Lemma 3, we deduce that R is a simple recipe. In case $R = \mathsf{f}(R_1, \ldots, R_n)$ for some $\mathsf{f} \in \Sigma_\mathsf{c}$, then we have that $C = \mathsf{f}(C_1, \ldots, C_n)$. Applying our induction hypothesis on each R_i with term $C_i[\_]$ which is also a strict direct subterm of a shape, we have that $R_i\phi\downarrow = C_i[x_1, \ldots, x_n]\theta$, and thus we deduce that $R_i\psi\downarrow = C_i[x_1, \ldots, x_n]\theta'_i$ for each i, and since a subterm of a shape is a linear term, we get the result by considering θ' the union of the θ'_i. Otherwise, we have that R is an almost destructor-only recipe. As in the previous case (see *the test T is a recipe R such that $R\phi\downarrow$ is a message* - item 2), we can show that R is subterm asap. In case R is asap, we are done. Thus, it remains the case where R is subterm asap but not asap. We know that there exists an asap simple recipe R' w.r.t. ϕ such that $R = R'$ holds in ϕ. By hypothesis, we know that $\phi \sqsubseteq_s^{\mathsf{simple}} \psi$, and thus by definition of $\sqsubseteq_s^{\mathsf{simple}}$, we have that $R = R'$ holds in ψ. We have that the test R' is compliant with shape $C[x_1, \ldots, x_n]$ in ϕ, and thus by induction hypothesis R' is compliant with shapre $C[x_1, \ldots, x_n]$ in ψ, and thus this also holds for R. This allows us to conclude.

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 $R\phi\downarrow$ is a message, we deduce that $R''\phi\downarrow = R\phi\downarrow$ (Lemma 1). We have shown that $R\psi\downarrow$ is a message, and thus $R''\psi\downarrow = R\psi\downarrow$ thanks to Lemma 1. 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, as in the previous cases, thanks to Lemma 3, 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 almost destructor-only recipes. If neither C nor C' is empty (that is, neither R nor R' is almost destructor-only) then $\mathsf{root}(R) = \mathsf{root}(R')$, and thus we conclude relying on our induction hypothesis applied on direct subterms of R and R'. Otherwise, we assume w.l.o.g. that R is an almost destructor-only recipe, and R' is simple. In case R is not subterm asap, it means that there exists $1 \leq i_0 \leq k$ such that $R|_{i_0}$ is not asap. Let $\overline{R_{i_0}}$ be an asap recipe w.r.t. ϕ such that $R|_{i_0}\phi\downarrow = \overline{R_{i_0}}\phi\downarrow$. We have that $R|_{i_0} = \overline{R_{i_0}}$ holds in ϕ, and relying on our induction hypothesis, we deduce that $R|_{i_0} = \overline{R_{i_0}}$ holds in ψ. Consider $R'' = R[\overline{R_{i_0}}]_{i_0}$. We deduce that $R = R''$ holds in ψ. Relying on our induction hypothesis applied on $R'' = R'$ which holds in ϕ, we deduce that $R'' = R'$ holds in ψ, and this allows us to conclude that $R = R'$ holds in ψ. Now, we can assume that R is subterm asap. Regarding the recipe R', we have that $R' = \mathsf{f}(R'_1, \ldots, R'_\ell)$. We would like to show that R' is subterm asap. Indeed, assume w.l.o.g. that R'_1 is not asap and let R''_1 be an asap recipe w.r.t. ϕ. We have that $R'_1 = R''_1$ holds in ϕ, and relying on our induction hypothesis, we have that $R'_1 = R''_1$ holds in ψ. Consider $R'' = \mathsf{f}(R''_1, \ldots, R'_\ell)$. We have that $R = R''$ holds in ϕ and ψ. Thus, we have that $R'' = R$ holds in ϕ and relying on our induction hypothesis, we deduce that $R'' = R$ holds in ψ. This allows us to conclude that $R = R'$ holds in ψ.
  Now, it remains to show that R or R' is asap. Assume w.l.o.g. that both are not asap. Let R'' be an asap recipe w.r.t. ϕ such that $R''\phi\downarrow = R\phi\downarrow = R'\phi\downarrow$. Relying on our induction hypothesis, we can transfer the tests $R = R''$ and $R'' = R'$ that both holds in ϕ, and thus we deduce that these tests also hold in ψ, and thus we conclude that $R = R'$ holds in ψ.

The test T is of the form $(R_1, \ldots, R_n)$ such that $R_1, \ldots, R_n$ are recipes, $(R_1\phi\downarrow, \ldots, R_n\phi\downarrow) = (t_1, \ldots, t_n)\theta$ for some θ and a rule $\mathsf{g}(t_1, \ldots, t_n) \to r \in \mathcal{R}_\mathsf{test} \cup \mathcal{R}_\mathsf{d}$.

- Case R_1 (resp. $R_2, \ldots, R_n$) is *not* in normal form w.r.t. $\twoheadrightarrow$. Let $R'_1 = R_1\downarrow$. Since $R_1\phi\downarrow$ is a message, we deduce that $R'_1\phi\downarrow = R_1\phi\downarrow$ (Lemma 1). We have shown that $R_1\psi\downarrow$ is a message too, and thus thanks to Lemma 1, we have that $R'_1\psi\downarrow = R_1\psi\downarrow$. Relying on our induction hypothesis applied on the test $R'_1, R_2, \ldots, R_n$, we deduce that the test $R'_1, R_2, \ldots, R_n$ holds in ψ, and thus we conclude that the test $R_1, R_2, \ldots, R_n$ holds in ψ.
- Otherwise, as in the previous cases, thanks to Lemma 3, we know that $R_1, \ldots, R_n$ are simple. In case, one of them is not asap, e.g. R_i, we can consider R'_i an asap simple recipe such that $R_1 = R'_1$ holds in ϕ, and thus in ψ relying on our induction hypothesis (item 3 we have already proved), and this allows us to conclude. Thus, we can

now assume that $R_1, \ldots, R_n$ are simple and asap. In case R_1 is almost destructor-only, we conclude relying on our hypothesis. Otherwise, we have that $R_1 = \mathsf{f}(R_1^1, \ldots, R_1^k)$ for some $\mathsf{f} \in \Sigma_c$. We distinguish two cases.

Case 1: $\ell = \mathsf{g}(t_1, \ldots, t_n)$ is linear. We thus only have to ensure that, for each $1 \leq i \leq n$ such that t_i is not a variable, $R_i\psi\!\downarrow$ complies with the shape of $\mathsf{sh}_{\mathsf{root}(t_i)}$ knowing that it is the case for $R_i\phi\!\downarrow$. This is actually an easy consequence of our induction hypothesis. Note that item 2 has already been proved, and thus we can rely on it even if our measure has not strictly decreased.Therefore, we are done for this case.

Case 2: $\ell = \mathsf{g}(t_1, \ldots, t_n)$ is not a linear term. In such a case, we necessarily have that $t_1 = \mathsf{f}(t_1^1, \ldots, t_1^k)$ for some $\mathsf{f} \in \Sigma_\mathsf{c}$. We know that t_1 is linear and contains exactly one occurrence of the non-linear variable x. Let $j \in \{1, \ldots, k\}$ be such that x occurs in t_1^j. By hypothesis we have that:

$$\{x\} \subseteq \{t_1^j, t_2, \ldots, t_n\} \subseteq \{x\} \cup \{\mathsf{h}(x) \mid \mathsf{h} \in \Sigma_\mathsf{c}\}$$

Let R_x be a recipe among $\{R_1^j, R_2, \ldots, R_n\}$ such that the term t_x among $\{t_1^j, t_2, \ldots, t_n\}$ associated to it is the variable x. By hypothesis, we know that there exists u (of sort atom) such that:

$$\text{for all } p \text{ such that } \ell|_p = x, \text{ we have that } \mathsf{g}(\mathsf{f}(R_1^1\phi\!\downarrow, \ldots, R_1^k\phi\!\downarrow), R_2\phi\!\downarrow, \ldots, R_n\phi\!\downarrow)|_p = u.$$

To conclude, it remains to show that there exists v (of sort atom) such that:

$$\text{for all } p \text{ such that } \ell|_p = x, \text{ we have that } \mathsf{g}(\mathsf{f}(R_1^1\psi\!\downarrow, \ldots, R_1^k\psi\!\downarrow), R_2\psi\!\downarrow, \ldots, R_n\psi\!\downarrow)|_p = v.$$

It is easy to see that using R_x (almost destructor-only and asap recipe) or $\mathsf{h}(R_x)$ (simple and subterm asap recipe), we can transfer the required equalities from ϕ to ψ. For instance, assume that $t_2 = \mathsf{h}(x)$, and also that $\mathsf{root}(R_2) \neq \mathsf{h}$. We know that R_2 is almost destructor-only and asap. Consider the test $\mathsf{h}(R_x) = R_2$. We have that it holds in ϕ, and thus by hypothesis it also holds in ψ. A similar reasoning allows us to conclude for each occurrence of x in ℓ, and this allows us to conclude.

□

We are now able to prove our main theorem regarding trace equivalence.

Theorem 3. *Let P be a* simple *protocol type-compliant w.r.t. some typing system (Δ_0, δ_0). Let Q be another* simple *protocol such that $P \not\sqsubseteq_t Q$. There exists a trace $(\mathsf{tr}, \phi) \in \mathsf{trace}(P)$ witnessing this non-inclusion such that $\mathsf{Label}(\mathsf{tr}) \subseteq A$ for some $A \in \mathsf{dep}(P)$.*

Proof. Thanks to Theorem 7, we know the existence of a well-typed, cv-alien-free witness of non-inclusion which only involves simple asap recipes. We choose one having a minimal length. We denote D the execution graph associated to the execution $exec$ corresponding to this witness (tr, ϕ) w.r.t. P. Relying on the fact that P and Q are simple protocols, we distinguish the two following cases.

There does not exist ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$. In such a case, we have that $\mathsf{tr} = \mathsf{tr}' \cdot \ell$. This last action ℓ 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 $exec'$ the resulting execution, D' its execution graph, and (tr_0, ϕ_0) the corresponding trace of P. Actually, by definition of pruning, we have that $\mathsf{tr}_0 = \mathsf{tr}_0' \cdot \ell$ where tr_0' is the trace obtained by pruning D w.r.t. the set of nodes N_ℓ corresponding to all the dependencies of ℓ. We have that tr_0' passes in P and also in Q by minimality of the witness tr. Assume now that $\mathsf{tr}_0'.\ell$ does not pass in Q. Then, we have built a smaller witness of non inclusion (contradiction), unless $D = D'$. In the latter case, let v be the node corresponding to the action ℓ. We are done since, thanks to Proposition 1, we have that $\mathsf{Label}(exec|_v) \subseteq A$ for some $A \in \mathsf{dep}(\mathsf{Label}(v))$. Since $D' = D$, we have that $exec' = exec|_v = exec$. Moreover, by definition of $\mathsf{dep}(P)$, we have that $\mathsf{dep}(\mathsf{Label}(v)) \subseteq \mathsf{dep}(P)$, and this allows us to conclude in the case where $D = D'$.

Otherwise, we have that $\mathsf{tr}_0'.\ell$ passes in Q meaning that ℓ 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 that satisfies our requirement.

There exists ψ such that $(\mathsf{tr}, \psi) \in \mathsf{trace}(Q)$ *but* $\phi \not\sqsubseteq_s \psi$. From Lemma 20, we can consider distinguishing tests that satisfy Definition 14. Let $\mathsf{W} \subseteq \mathcal{W}$ be the set of variables involved in such a test. In case $exec|_\mathsf{W}$ is indeed smaller in length than $exec$, then to conclude, it remains to establish that $exec|_\mathsf{W}$ is indeed a witness of non-inclusion. We have that $exec|_\mathsf{W}$ is an execution of P (since we keep all the necessary dependencies), and since we are considering simple protocols, we know that the corresponding trace tr' leads to a frame ϕ' such that $\phi' = \phi|_\mathsf{W}$. Similarly, we have that tr' can be executed starting from Q and this leads to a frame ψ' such that $\psi' = \psi|_\mathsf{W}$. The witness of non-inclusion we considered is therefore still valid on this smaller witness, and this allows us to conclude in case $exec|_\mathsf{W} \neq exec$. Now, we assume that $exec|_\mathsf{W} = exec$, and we show that the result holds, i.e. there exists $A \in \mathsf{dep}(P)$ such that $\mathsf{Label}(exec) \subseteq A$. We distinguish several cases depending on the form of the test.

1) *There exists an almost destructor-only and asap recipe R such that $R\phi\!\downarrow$ is a (resp. atomic) message.* Thanks to Lemma 2, we have that $R\phi\!\downarrow \in St(\phi)$. Then, thanks to Lemma 5, we have that $\mathsf{Cst}(R) \subseteq \mathsf{Cst}(\phi) \cup \mathsf{Cst}(R\phi\!\downarrow)$, and thus $\mathsf{Cst}(R) \subseteq \mathsf{Cst}(\phi)$, and since ϕ is cv-alien-free, we have that R is cv-alien-free too. Thanks to Proposition 1, we know that there exists $A \in S_{\mathsf{out}}(\tau)$ such that $\mathsf{Label}(exec|_R) \subseteq A$ where $\tau = \delta_0(R\phi\!\downarrow)$. We have seen

that $R\phi\downarrow \in St(\phi)$, and since we are considernig well-typed execution, we have that $\tau \in St(\delta_0(P))$. Since there exists $A \in S_{\mathsf{out}}(\tau)$, we have that $\mathsf{dep}(\tau) \neq \emptyset$. Moreover, we have that $S_{\mathsf{out}}(\tau) \subseteq S^+_{\mathsf{out}}(\tau)$, and thus we deduce that $A \in S_{\mathsf{out}}(\tau)$ implies that there exists $A' \in S_{\mathsf{test}}(P)$ such that $A \subseteq A'$.

2) *There exists $C[x_1, \ldots, x_n]$ a strict/direct substerm of a shape sh_f with $\mathsf{f} \in \Sigma_{\mathsf{c}}$, and an almost destructor-only and asap recipe R such that $R\phi\downarrow$ is a message of the form $C[x_1\theta, \ldots, x_n\theta]$.* The reasoning is similar to the one done in the previous item. We have that there exists $A \in \mathsf{dep}(P)$ such that $\mathsf{Label}(exec|_R) = \mathsf{Label}(exec) \subseteq A$.

3) *There exist a simple recipe R and an almost destructor-only recipe R' such that $R\phi\downarrow = R'\phi\downarrow$ (is a message). In addition, one recipe is asap, and the other is subterm asap w.r.t. ϕ.* Thanks to Lemma 2, we have that $R\phi\downarrow = R'\phi\downarrow \in St(\phi)$. Then, thanks to Lemma 5, and since ϕ is cv-alien-free, we have that both R and R' are cv-alien-free. Let $\tau = \delta_0(R\phi\downarrow)$. We have seen that $R\phi\downarrow \in St(\phi)$, and since we are considernig well-typed execution, we have that $\tau \in St(\delta_0(P))$.
First, we assume that R is asap and R' is only subterm asap. Thanks to Proposition 1, we know that:
   - there exists $A \in \mathsf{dep}(\tau)$ such that $\mathsf{Label}(exec|_R) \subseteq A$; and
   - there exists $A' \in S^+_{\mathsf{out}}(\tau)$ such that $\mathsf{Label}(exec|_{R'}) \subseteq A'$.

   Therefore, we deduce that there exists $A_0 \in \mathsf{dep}(\tau) \otimes \S^+_{\mathsf{out}}(\tau)$ such that $\mathsf{Label}(exec|_{\{R,R'\}}) \subseteq A_0$. We thus have that $A_0 \in S_{\mathsf{test}}(P)$ and this allows us to conclude.
   Now, we assume that R is subterm asap, and R' is asap. Thanks to Proposition 1, we know that:
   - there exists $A \in S_{\mathsf{out}}(\tau)$ such that $\mathsf{Label}(exec|_R) \subseteq A$; and
   - there exists $A' \in \mathsf{dep}^+(\tau)$ such that $\mathsf{Label}(exec|_{R'}) \subseteq A'$.

   Therefore, we deduce that there exists $A_0 \in \mathsf{dep}^+(\tau) \otimes S_{\mathsf{out}}(\tau)$ such that $\mathsf{Label}(exec|_{\{R,R'\}}) \subseteq A_0$. Actually, we have that:

$$\begin{aligned}
&\mathsf{dep}^+(\tau) \otimes S_{\mathsf{out}}(\tau)\\
&= \big(\mathsf{dep}^0(\tau) \cup S^+_{\mathsf{out}}(\tau)\big) \otimes S_{\mathsf{out}}(\tau)\\
&= \big(\mathsf{dep}^0(\tau) \otimes S_{\mathsf{out}}(\tau)\big) \cup \big(S^+_{\mathsf{out}}(\tau) \otimes S_{\mathsf{out}}(\tau)\big)\\
&\subseteq \mathsf{dep}(\tau) \otimes S^+_{\mathsf{out}}(\tau)
\end{aligned}$$

   We thus have that there exists $A_0 \in S_{\mathsf{test}}(P)$ such that $\mathsf{Label}(exec) = \mathsf{Label}(exec|_{\{R,R'\}}) \subseteq A_0$.

4) *There exists a rule $\mathsf{g}(t_1, \ldots, t_n) \to r \in \mathcal{R}_{\mathsf{test}} \cup \mathcal{R}_{\mathsf{d}}$, an almost destructor-only and asap recipe R_1, and some simple asap recipes $R_2, \ldots, R_n$ such that $R_1\phi\downarrow, \ldots, R_n\phi\downarrow$ are messages and $(R_1\phi\downarrow, \ldots, R_n\phi\downarrow) = (t_1, \ldots, t_n)\theta$ for some θ.* Thanks to Lemma 2, we have that $R_1\phi\downarrow \in St(\phi)$. Then, thanks to Lemma 5, and since ϕ is cv-alien-free, we have that R_1 is cv-alien-free. This allows us to deduce that $R_2\phi\downarrow, \ldots R_n\phi\downarrow$ are also cv-alien-free since those terms only involve constants occurring in $R_1\phi\downarrow$ or in $l = \mathsf{g}(t_1, \ldots, t_n)$ (left-hand side of a rewriting rule). In this latter case, those constants are constants from Σ_{c} and do not have an cv-alien-type by definition of cv-alien-type. Let $\tau = \delta_0(R_1\phi\downarrow)$. We have seen that $R_1\phi\downarrow \in St(\phi)$, and since we are considering well-typed execution, we have that $\tau \in St(\delta_0(P))$. For any $i \in \{2, \ldots, n\}$, we have that $\delta(t_i\theta) = t_i\delta_0(\theta)$. Let $\tau_i = t_i\delta_0(\theta)$ for any $i \in \{2, \ldots, n\}$.
   Thanks to Proposition 1, we know that:
   - there exists $A_1 \in S_{\mathsf{out}}(\tau)$ such that $\mathsf{Label}(exec|_{R_1}) \subseteq A_1$; and
   - for any $i \in \{2, \ldots, n\}$, there exists $A_i \in \mathsf{dep}(\tau_i)$ such that $\mathsf{Label}(exec|_{R_i}) \subseteq A_i$.

   Therefore, we deduce that there exists $A \in S_{\mathsf{check}}(P)$ such that $\mathsf{Label}(exec) = \mathsf{Label}(exec|_{\mathsf{W}}) \subseteq A$.

Let W be the set of all the variables occurring in the test witnessing the non-inclusion. We have seen that there exists $A \in \mathsf{dep}(P)$ such that $\mathsf{Label}(exec) = \mathsf{Label}(exec|_{\mathsf{W}}) \subseteq A$, and this allows us to conclude. □