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

2022 IEEE 35th Computer Security Foundations Symposium (CSF) | 978-1-6654-8417-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/CSF54842.2022.9919670

***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 sub-programs, 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.

DOI 10.1109/CSF54842.2022.00003

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, as well as the omitted proofs are available 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.

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

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 $\mathsf{f} \in \Sigma_{\mathsf{c}} : (s_1 \times \ldots \times s_n) \rightarrow s_0$ 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:*

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

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:*

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

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 $\mathsf{g}(t_1, ..., t_n) \rightarrow t$ in $\mathcal{R}$ such that $u|_p = \mathsf{g}(t_1, \dots, t_n)\theta$ 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 $t_1\theta, \dots, t_n\theta$ 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 $\rightarrow^*$ the reflexive-transitive closure of $\rightarrow$, 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:*

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

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:

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

where $\alpha \in \mathcal{L}$, $u, u_1, \dots, u_k \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \Sigma_0 \uplus \mathcal{N} \uplus \mathcal{X})$, $c, c' \in \mathcal{C}h$, $n \in \mathcal{N}$, 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:*

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

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, $\mathsf{match}\, x\, \mathsf{with}\, (u_1 \rightarrow P_1 \mid \dots \mid u_j \rightarrow P_j)$ will try to unify the content of x with $u_1, \dots, u_j$ (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:

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

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 $R_1 \overset{\mathsf{def}}{=} \mathsf{aenc}(\mathsf{adec}(\mathsf{w}_1, ekc), \mathsf{w}_0)$ 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

$$\begin{array}{lrcl}
\textsc{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)\\
& \multicolumn{3}{r}{\text{where } R \text{ is a recipe such that } R\phi\!\downarrow \text{ is a message, and } R\phi\!\downarrow = (u\sigma)\sigma_0 \text{ for } \sigma_0 \text{ with } dom(\sigma_0) = vars(u\sigma).}\\
\textsc{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}\rhd u\sigma\};\sigma;i)\\
& \multicolumn{3}{r}{\text{with } \mathsf{w} \text{ a fresh variable from } \mathcal{W}, \text{ and } u\sigma \text{ is a message.}}\\
\textsc{New} & (i:\mathsf{new}\, n.P \uplus \mathcal{P};\phi;\sigma;i) & \xrightarrow{\tau} & (i:P\{^m/_n\}\uplus\mathcal{P};\phi;\sigma;i)\\
& \multicolumn{3}{r}{\text{where } m \in \mathcal{N} \text{ is a fresh name.}}\\
\textsc{Null} & (i:0 \uplus \mathcal{P};\phi;\sigma;i) & \xrightarrow{\tau} & (\mathcal{P};\phi;\sigma;i)\\
\textsc{Par} & (i:(P \mid Q) \uplus \mathcal{P};\phi;\sigma;i) & \xrightarrow{\tau} & (i:P\uplus i:Q\uplus\mathcal{P};\phi;\sigma;i)\\
\textsc{Rep} & (i:!P \uplus \mathcal{P};\phi;\sigma;i) & \xrightarrow{\tau} & (i:P'\uplus i:!P\uplus\mathcal{P};\phi;\sigma;i)\\
& \multicolumn{3}{r}{\text{with } P' \text{ a copy of } P \text{ where bound variables are renamed.}}\\
\textsc{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)\\
& \multicolumn{3}{r}{\text{where } c'' \text{ is a fresh channel name.}}\\
\textsc{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)\\
& \multicolumn{3}{r}{\text{where } j_0 \text{ is the smallest index such that } x\sigma \text{ and } u_{j_0}\sigma \text{ unify and } \sigma_0 = mgu(x\sigma, u_{j_0}\sigma).}
\end{array}$$

$$\begin{array}{lrcll}
\textsc{Move} & (\mathcal{P};\phi;\sigma;i) & \xrightarrow{\mathsf{phase}\ i'} & (\mathcal{P};\phi;\sigma;i') & \text{with } i' > i.\\
\textsc{Phase} & (i:i':P\uplus\mathcal{P};\phi;\sigma;i) & \xrightarrow{\tau} & (i':P\uplus\mathcal{P};\phi;\sigma;i) & \\
\textsc{Clean} & (i:P\uplus\mathcal{P};\phi;\sigma;i') & \xrightarrow{\tau} & (\mathcal{P};\phi;\sigma;i') & \text{when } i' > i.
\end{array}$$

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:*

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

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

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

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*
- $\delta_{\mathsf{DS}}(\mathrm{xx}) = \tau_{\mathrm{xx}}$ *for* $\mathrm{xx} \in \{ska, ekb, ekc\}$.

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

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, $\tau = \mathsf{f}(\tau_1, \dots, \tau_k)$ and E is defined as:*

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

Example 14. *Let $\tau_{\mathsf{msg}} \overset{\mathsf{def}}{=} \mathsf{aenc}(\mathsf{sign}(\tau_k, \tau_{ska}), \mathsf{pk}(\tau_{ekb}))$. 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:*

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

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,

$$\begin{aligned}\{a_1,\dots,a_k\}\otimes\{b_1,\dots,b_l\} \overset{\mathsf{def}}{=}& \\ &\{a_1\uplus b_1, a_1\uplus b_2,\dots,a_1\uplus b_l,\\ &\ a_2\uplus b_1, a_2\uplus b_2,\dots,a_2\uplus b_l,\\ &\qquad\vdots\\ &\ a_k\uplus b_1, a_k\uplus b_2,\dots,a_k\uplus b_l\}\end{aligned}$$

where $a_1,\dots,a_k,b_1,\dots,b_l$ 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, $i\geq 0$:

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

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 $\mathsf{f}(\tau_1,\dots,\tau_k)$.

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 $A_i\subseteq B_i$.

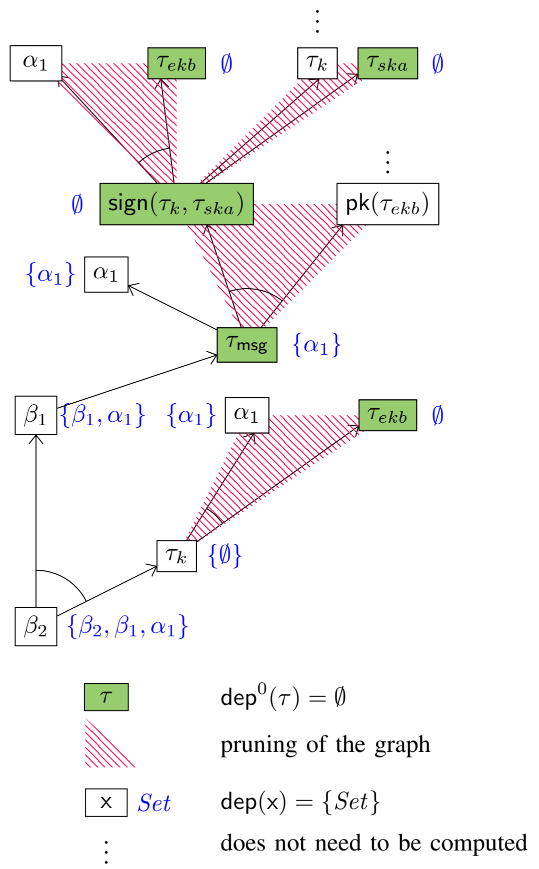

Fig. 2. Computation of $\mathsf{dep}(\beta_2)$.

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 $S\otimes\emptyset=\emptyset$.

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*
- $\begin{array}{rl}\mathsf{dep}(\mathsf{sign}(\tau_k, \tau_{ska})) & = \{\emptyset\} \otimes \mathsf{dep}(\tau_k) \otimes \mathsf{dep}(\tau_{ska}) \\ & \quad \cup\ S_{\mathsf{out}}(\mathsf{sign}(\tau_k, \tau_{ska})) \\ & = \emptyset\end{array}$

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.

[1]A formal defintion is given in Appendix

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.

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 $u \in \mathcal{T}_0(\Sigma_{\mathsf{c}}, \Sigma_0 \cup \mathcal{N} \cup \mathcal{X})$, $\alpha \in \mathcal{L}$, 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

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

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

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

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

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

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).

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 $\perp$ 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 $\perp$ 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 $\perp$ 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.

	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

$\star$: 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

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 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, 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, 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, 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, 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, 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, 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, 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. Some preliminaries

Our technical development relies 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 II, 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) *$\ell'_{\mathsf{g}} = \mathsf{g}(x_1, \dots, x_n)$ 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 $\ell_{\mathsf{g}}|_1$ 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 $\mathsf{g}(t_1, \dots, t_n) \twoheadrightarrow t$ in $\mathcal{R}_f$ such that $u|_p = \mathsf{g}(t_1, \dots, t_n)\theta$ for some substitution θ, and $v = u[t\theta]_p$. We denote $u\downarrow$ the forced normal form of 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 $1 \dots 1$ in R. A recipe R is *simple* when $R = C[R_1, \dots, R_n]$ for some context C built using symbols in $\Sigma_{\mathsf{c}} \cup \Sigma_0$, and each R_i is almost destructor-only. Note that an almost destructor-only recipe is a simple recipe.

B. Key result

When considering recipes for a witness of non static inclusion, we can not assume 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. 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. The definition of $S^+_{\mathsf{out}}(\tau)$ (which contains a bit of marking) is given in Section V. In addition, we define

$$\mathsf{dep}^+(\tau) = \mathsf{dep}^0(\tau) \cup (\{\emptyset\} \otimes \mathsf{dep}(\tau_1) \otimes \dots \otimes \mathsf{dep}(\tau_k)) \cup S^+_{\mathsf{out}}(\tau)$$

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

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 11. *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 of P 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 $vars(R) = \{\mathsf{w}_{i_1}, \dots, \mathsf{w}_{i_k}\}$, we denote $exec|_R$ the execution obtained by pruning D w.r.t. $\{v_1, \dots, v_k\}$ 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. in $exec$), and given a node v of a dag (coming from an execution trace), we denote $\mathsf{Label}(v)$ its label.

Then, we are able to state this key result whose proof can be done by induction on the length of the execution.

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 $\mathsf{Label}(exec|_R) \subseteq A$.*

C. Marking

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.

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 $\mathsf{g}(t_1, \dots, t_n) \to r_{\mathsf{g}} \in \mathcal{R}_{\mathsf{d}}$ 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 12. *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).*

Lemma 1. *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 $u \in \mathcal{T}(\Sigma_\mathsf{c}, \Sigma_0)$.*

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

Lemma 2. *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.*

Our second criterion is a *precedence criterion*. 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.

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 3. *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.

D. The case of equivalence

Regarding trace equivalence, 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. A similar result has already been established in [21] considering a fixed set of primitives.
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. More precisely, we establish that we may consider a witness tr of the non-inclusion $P \sqsubseteq_t Q$ for which there exists $A \in \mathsf{dep}(P)$ such that $\mathsf{Label}(\mathsf{tr}) \subseteq A$. Remenber that

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

   where S_reach, S_test, and S_check are given in Figure 3.

Our alternative definition of static inclusion is formally defined below.

Definition 13. *Let ϕ, ψ be such that $dom(\phi) = dom(\psi)$. We write $\phi \sqsubseteq^\mathsf{simple}_s \psi$ 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 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 is actually equivalent to the original one.

Lemma 4. *Let ϕ and ψ be two frames having the same domain. We have that: $\phi \sqsubseteq_s \psi \Leftrightarrow \phi \sqsubseteq^\mathsf{simple}_s \psi$.*

We now briefly explain the purpose of the different components in the definition of $\mathsf{dep}(P)$. The case $S_\mathsf{reach}(P)$ corresponds to the case where the witness tr exists in P but not in Q. The case $S_\mathsf{check}(P)$ corresponds to a failure of static inclusion, and more precisely to the item 4 of Definition 13.

The case $S_\mathsf{test}(P)$ corresponds to a failure of static inclusion, and actually covers the items 1, 2, and 3 of Definition 13.

1) Assuming that the recipe R is such that $\delta_0(R\phi\downarrow) = \tau$, relying on Proposition 1, we get that it is enough to consider a set A in $S_\mathsf{out}(\tau)$. Actually, $S_\mathsf{out}(\tau) \subseteq S^+_\mathsf{out}(\tau) \subseteq S_\mathsf{test}(P)$ since $\tau \in St(\delta_0(P))$.
2) This case can be handled in a similar way.
3) This case is a bit more complex and we have to consider different cases depending on whether the simple recipe is the asap one or not. Let us assume that $\delta_0(R\phi\downarrow) = \tau$. Relying on Proposition 1, in case the simple recipe is also the asap one, we get it is enough to consider a set A in $\mathsf{dep}(\tau) \otimes S^+_\mathsf{out}(\tau)$; whereas we obtain a set A in $\mathsf{dep}^+(\tau) \otimes S_\mathsf{out}(\tau)$ otherwise. Anyway, we can show that $\mathsf{dep}^+(\tau) \otimes S_\mathsf{out}(\tau) \subseteq \mathsf{dep}(\tau) \otimes S^+_\mathsf{out}(\tau) \subseteq S_\mathsf{test}(P)$ since $\tau \in St(\delta_0(P))$.