Statically Detecting Message Confusions in a Multi-Protocol Setting *

Chiara Bodei[1], Linda Brodo[2], Pierpaolo Degano[1], Han Gao[3]

1: Dipartimento di Informatica, Università di Pisa, Via Pontecorvo, 3, I-56127 Pisa - Italy email: {chiara,degano}@di.unipi.it
2: Dipartimento di Scienze dei Linguaggi, Università di Sassari, via Tempio,9, I-07100 Sassari - Italy email: brodo@uniss.it
3: Informatics and Mathematical Modelling, Technical University of Denmark, R. Petersens Plads bldg 321, DK-2800 Kongens Lyngby - Denmark
email: hg@imm.dtu.dk

Abstract. In a multi-protocol setting, different protocols are concurrently executed, and each principal can participate in more than one. The possibilities of attacks therefore increase, often due to the presence of similar patterns in messages. Messages coming from one protocol can be confused with similar messages coming from another protocol. As a consequence, data of one type may be interpreted as data of another, and it is also possible that the type is the expected one, but the message is addressed to another protocol. In this paper, we shall present an extension of the LySa calculus [7, 4] that decorates encryption with tags including the protocol identifier, the protocol step identifier and the intended types of the encrypted terms. The additional information allows us to find the messages that can be confused and therefore to have hints to reconstruct the attack. We extend accordingly the standard static Control Flow Analysis for LySa, which over-approximates all the possible behaviour of the studied protocols, included the possible message confusions that may occur at run-time. Our analysis has been implemented and successfully applied to small sets of protocols. In particular, we discovered an undocumented family of attacks, that may arise when Bauer-Berson-Feiertag and the Woo-Lam authentication protocols are running in parallel. The implementation complexity of the analysis is low polynomial.

keywords: Multi-protocol attacks, Type Flaw Attacks, Control Flow Analysis, LySa.

1 Introduction

Usually, security protocols are studied and verified in isolation, i.e. under the hypothesis that there is only a single protocol using the network at a time. However, this is not realistic, since protocols share the same network and often

* This work has been partially supported by the project SENSORIA and by the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004.

they also share some cryptographic keys, for economical or practical reasons. Verification of properties is more difficult in this setting, because of possible unintended and unforeseen interactions: new attacks, involving more than one protocol, can arise, called *multi-protocol attacks*. For this reason, protocols that are proved secure in isolation, can be attack-prone when running in parallel: indeed, security properties are not *compositional* in general. The first works on this subject are in [2, 19], where the authors show that given a secure protocol, an *ad hoc* protocol can be used to attack it, by suitably interleaving messages of both. Formal verification in this setting, as put forward in [22], is something to which we intend to contribute. In the multi-protocol setting, one of the main problems is the presence of similar patterns in messages coming from different protocols. As a consequence, attacks may easily occur, due to the confusion between two messages that belong to different protocols. Mainly, these attacks are based on *type flaws*, that occur when a field, originally intended to have one type, is instead interpreted as having another type. To prevent such attacks, the current techniques [17, 18] consist in systematically associating each message field with a tag representing its intended type. Therefore fields with different types cannot be mixed up. Other attacks may arise instead, that depend on the mix-up of two messages with the same structure but that belong to different protocols. Therefore, in general, type tags may not suffice. Following [8] and extending the work done in [5], we then annotate each message with an identifier for the protocol, and the protocol step, and with the type of each message component. Here, we only deal with message confusions arising from messages that include the same number of terms, e.g. in the present framework, it is not possible to confuse a concatenation of terms with a single term (it is possible instead in [13]).

In this paper, we explore these issues and propose a static analysis technique, based on Control Flow Analysis, for detecting potential type flaw attacks in a multi-protocol setting. The same unifying framework has been previously used to deal with a wide range of security properties, including confidentiality [14], freshness [12] and message authentication [7]. To address type flaw attacks, we extend the version of LySA calculus presented in [4] with special tags, that, besides the intended types of the encrypted terms, also include the protocol identifier and the protocol step identifier. This additional information can also be exploited to understand which are the possible message confusions and interferences between different protocols. The Control Flow Analysis approximates the behaviour of protocols in terms of the possibly exchanged messages and potential values of variables. Our framework can be working in either a *prescriptive* way, such that flaws are avoided; or a *descriptive* way, such that flaws are detected and recorded as violations. Furthermore, if no tag violation is found, we can prove that the protocol is free of type flaw attacks at run time. The analysis is fully automated and always terminates. By applying this framework to the multi-protocol setting, we can statically verify if in a small group of protocols tag-violations and the consequent attacks are possible. In particular, we discovered a family of undocumented flaws (to our knowledge), possibly occurring

when the Bauer-Berson-Feiertag and the Woo-Lam authentication protocols are running in parallel. They are based on the classical oracle mechanism.

The paper is organised as follows. In Section 2, we present the LySA calculus with tags. We introduce the Control Flow Analysis in Section 3, which captures any tag-mismatching that may happen. In Section 4, we show how the Control Flow Analysis can detect an attack that may arise in the composition of two protocols. In Section 5, we conclude with a discussion on our approach.

2 Calculus

The LySA calculus [7] is a process algebra, in the tradition of the π- [23] and Spi- [1] calculi. It differs from these essentially in two aspects: (1) the absence of channels: all processes have only access to a single global communication channel, the network; (2) the inclusion of pattern matching *into* the language constructs where values can become bound to variables, i.e. into input and into decryption (while usually there is a separate construct). We use here a dialect of LySA, which presents a more general pattern matching than the one in [7] and that slightly extend the one in [4], thus allowing us to tag encryptions and their components. In this paper, we only show the modelling of symmetric key protocols. Asymmetric key protocols can be dealt with in a very similar way (see e.g., [7]).

Syntax of Terms LySA consists of terms and processes; values correspond to *closed* terms, i.e. terms without free variables. Values are then the basic blocks of the calculus and are used to represent agent names, nonces, keys. Terms may either be *terms* E or *definition terms* M. In fact, we distinguish between *definition* (or binding) occurrences and *use* (or applied) occurrences of variables. A definition occurrence is when a variable gets its binding value, while a use occurrence is an appearance of a variable where its binding value is used. So far, we have that terms – that can be names or variables – are used for modelling outputs and encryptions. Instead, for modelling inputs and decryptions we use definition terms, that include terms and definition variables.

The use/definition distinction is obtained by means of syntax: the definition occurrence of a variable x is denoted by $\natural x$, while in the scope of the declaration, the variable appears as x. This notation allows us to distinguish variables from occurrences of terms in tuples of definition terms, by implicitly partitioning them into terms or variables. In pattern matching, the first are checked for matching, while the others are bound in case of successful matching (see below).

We have two syntactic categories, one for terms E and one for definition terms M. Encryptions are tuples of terms (definition terms) $E_1, \cdots, E_k$ ($M_1, \cdots, M_k$, resp.) encrypted under a term E_0 representing a shared key.

$E ::=$	*terms*	$M ::=$	*definition terms*
n	name ($n \in \mathcal{N}$)	E	terms
x	use variable ($x \in \mathcal{X}_{\mathcal{S}}$)	$\natural x$	definition variable
$\{E_1, \cdots, E_k\}_{E_0}$	symmetric encryption	$\{M_1, \ldots, M_k\}_{E_0}$	definition encryption

Here $\mathcal{N}$ denotes a set of names; the set $\mathcal{X}_S$ contains occurrences of variables, be they use or definition, respectively.

Tagging On the top of this syntax, in [4], an extension was proposed, in order to cope with types, where tags were introduced to represent the intended types of terms. The following formal treatment (syntax, semantics and control flow analysis) is very similar to the one adopted in [4]. Following [17], we indeed assumed to have a tag for each base type, such as *nonce*, *key*, etc.. Moreover, we assume that the attacker is able to change tags, but only those of terms that he can access. In fact, by making the assumption of perfect cryptography, we have that only cleartext can be altered. Attackers can thus only forge an encryption when possessing the key used to cipher it. Actually, we can tag every term we want, but it suffices to check this information *inside* encryptions and decryptions, as shall be shown in Section 3. Here, we further extend the above syntax in order to include in the tagging schema of encryptions, an identifier ID for the protocol, and an identifier for the message step i, in which the encryption is generated. Tag ranges over a given set **Tag** that contains, besides type tags, such as *agent*, *nonce*, *key*, and *enc*, also a string ID_i encoding the *protocol identifier* and the *message step identifier*. Furthermore, there are tag variables, that are to standard variables such as tags are to closed terms (i.e. terms without variables). Similarly to the $\natural$-notation, we syntactically distinguish the definition occurrences of tag variables (in the form $\sharp t$), from the corresponding use occurrences (in the form t). Syntactically, we enrich the previous categories with the following productions, where $\mathcal{X}_T$ denote sets of occurrences of tag variables:

$E ::=$ *terms*		$M ::=$ *definition terms*	
...		...	
Tag	tags ($Tag \in$ **Tag**)	$\sharp t$	definition tag variable ($t \in \mathcal{X}_T$)
t	use tag variable ($t \in \mathcal{X}_T$)		

Intuitively, when specifying a protocol, we substitute each encryption like $\{E_1, ..., E_k\}_{E_0}$ with the tagged form $\{ID_i, (E_1, Tag_1) \cdots, (E_k, Tag_k)\}_{E_0}$. We call Val the set of values, i.e. closed terms. Each term can have a tag associated with it. We do not associate each term with the corresponding tag, though. We just use tags, when necessary. Put in other words, it is like having a jolly tag associated with each other term, that is omitted, because its check is always successful.

Syntax of Processes In addition to the classical constructs for composing processes, our calculus also contains an input and a decryption constructs both with matching. Furthermore, to allow the static analysis to keep track of the decryptions in which a violation may occur, we decorate each decryption with a label l (from an enumerable set $\mathcal{C}$). Labels are mechanically attached to program points in which decryptions occur (actually, they are specific nodes in the abstract syntax tree of processes), so the user/attacker cannot manipulate them. Finally, by overloading the symbol ν, we use a new process construct to declare

the expected tag of a tag variable.

$$P ::= \langle E_1, \ldots, E_k \rangle.P \mid (M_1, \ldots, M_k).P \mid \mathit{decrypt}\ E\ \mathit{as}\ \{M_1, \ldots, M_k\}^l_{E_0}\ \mathit{in}\ P \mid (\nu\ n)P \mid\ (\nu\ \sharp t : Tag)P \mid\ P_1|P_2 \mid\ !P \mid\ \mathsf{0}$$

We use $\mathsf{fv}(\cdot)$ for representing the sets of free variables and tag variables, $\mathsf{fn}(\cdot)$ for free names and tags, $\mathsf{bv}(\cdot)$ for bound variables/tag variables, and $\mathsf{bn}(\cdot)$ for bound names/tags, respectively. As usual, we omit the trailing $\mathsf{0}$ of processes.

Pattern matching is included both in inputs and decryptions. In both cases, our patterns are tuples of definition terms $(M_1, \cdots, M_k)$ that have to be matched against tuples of terms $(E_1, \cdots, E_k)$, when receiving (decrypting, resp.). Note that, at run time, each tuple $(E_1, \cdots, E_k)$ only includes closed terms, i.e. each variable composing each one of the E_i has been bound in the previous steps of the computation. Instead, definition terms M_i can be partitioned into closed terms to be matched and variables to be bound. Intuitively, the matching succeeds when the closed terms, say M_i, pairwise match to the corresponding terms E_i, and its effect is to bind the remaining terms E_j to the remaining variables $\natural x_j$. To exemplify, consider the following processes.

$$\begin{aligned}
R &= (\nu \sharp t_{id} : ID_2)(\nu \sharp t_k : key)\mathit{decrypt}\ \{ID_2, (A, agent), (N_B, nonce), (z, key)\}_K\ \mathit{as} \\
&\qquad \{\sharp t_{id}, (A, agent), (N_B, nonce), (\natural z_k, \sharp t_k)\}^{l_R}_K\ \mathit{in}\ R' \\
\tilde{R} &= (\nu \sharp t_{id} : ID_2)(\nu \sharp t_k : key)\mathit{decrypt}\ \{ID'_3, (A, agent), (N_B, nonce), (z, nonce)\}_K\ \mathit{as} \\
&\qquad \{\sharp t_{id}, (A, agent), (N_B, nonce), (\natural z_k, \sharp t_k)\}^{l_R}_K\ \mathit{in}\ \tilde{R}' \\
S &= \mathit{decrypt}\ \{t_{id}, (A, agent), (N_B, nonce), (z, t)\}_K\ \mathit{as} \\
&\qquad \{ID_2, (A, agent), (N_B, nonce), (\natural z_k, key)\}^{l_S}_K\ \mathit{in}\ S'
\end{aligned}$$

The decryptions in R and $\tilde{R}$ always succeed and result in binding $\natural z_k$ to (the values assumed by) z, $\sharp t_k$ to key and $\sharp t_{id}$ to ID_2, in R, and $\sharp t_k$ to $nonce$ and $\sharp t_{id}$ to ID'_3, in $\tilde{R}$. In particular, in $\tilde{R}$ the decryption succeeds, even though the declared tag for $\sharp t_k$ would be key and $\sharp t_{id}$ should be ID_2. In the decryption in S $\natural z_k$ is bound to z only if t is itself key and t_{id} is ID_2. Note that the principal that decrypts knows which is the protocol he is running and also knows in which step has been generated the encryption.

Operational Semantics To simplify the definition of our Control Flow Analysis in Section 3, we discipline the α-renaming of *bound* values and variables. To do it in a simple and "implicit" way, we partition all the names used by a process into finitely many equivalence classes and we use the names of the equivalence classes instead of the actual names. This partition works in a way that names from the same equivalence class are assigned a common *canonical name* and consequently there are only finitely many canonical names in any execution of a given process. This is enforced by assigning the same canonical name to every name generated by the same restriction. The canonical name $\lfloor n \rfloor$ is for a name n; similarly $\lfloor x \rfloor$ is for a variable x. In this way, we statically maintain the identity of values and variables that may be lost by freely applying α-conversions.

Hereafter, when unambiguous, we shall simply write n (resp. x) for $\lfloor n \rfloor$ (resp. $\lfloor x \rfloor$). Similarly for tag variables.

We give LySA a reduction semantics. We slightly modify the standard *structural congruence* $\equiv$ on LySA processes, to take care of tag declarations. We define $\equiv$ as the least congruence satisfying the following clauses:

- $P \equiv Q$ if P and Q are disciplined α-equivalent (as explained above);
- $(\mathcal{P}/_{\equiv}, |, \mathsf{0})$ is a commutative monoid;
- $(\nu n)\mathsf{0} \equiv \mathsf{0}, \quad (\nu n)(\nu n')P \equiv (\nu n')(\nu n)P, \quad (\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \quad$ if $n \notin \mathsf{fn}(P)$,
- $(\nu \sharp t : Tag)\mathsf{0} \equiv \mathsf{0}, \quad (\nu \sharp t : Tag)(\nu \sharp t' : Tag)P \equiv (\nu \sharp t' : Tag)(\nu \sharp t : Tag)P$, $(\nu \sharp t : Tag)(P \mid Q) \equiv P \mid (\nu \sharp t : Tag)Q \quad$ if $\sharp t \notin \mathsf{bv}(P)$;
- $!P \equiv P \mid !P$

The *reduction relation* $\rightarrow_{\mathcal{R}}$ is the least relation on closed processes that satisfies the rules in Table 1. We consider two variants of *reduction relation* $\rightarrow_{\mathcal{R}}$, graphically identified by a different instantiation of the relation $\mathcal{R}$, which decorates the transition relation. Both semantics use the tag environment Γ, which maps a tag variable to a tag.

$$\Gamma : \mathcal{X}_T \rightarrow \mathbf{Tag}$$

One variant takes advantage of checks on tag associations, while the other one discards them: essentially, the first semantics checks for tag matching, while the other one does not (see below):

- the *reference monitor semantics* $\Gamma \vdash P \rightarrow_{\mathsf{RM}} Q$ takes

$$\mathcal{R}(E, M, \Gamma) = \begin{cases} \text{false} & \text{if } (M = \sharp t) \wedge (E \neq \Gamma(\sharp t)) \\ \text{true} & \text{otherwise} \end{cases}$$

  this function only affects tag variables, i.e. only definition terms M in the form $\sharp t$. It checks whether the tag associated with the variable in the tag environment ($\Gamma(\sharp t)$) is E;
- the *standard semantics* $\Gamma \vdash P \rightarrow Q$ takes, by construction, $\mathcal{R}$ to be universally true (and therefore the index $\mathcal{R}$ is omitted).

Moreover, we define an auxiliary function that handles closed terms and variables to be bound in two different ways. Technically, we implicitly partition the tuples and treat the respective elements differently. The *pattern matching* function $comp(E, M)$ compares E against M only when M is a closed term and not a definition variable, nor a definition tag variable.

$$comp(E, M) = \begin{cases} \text{false} & \text{if } M \notin \{\natural x |\ x \in \mathcal{X}_S\} \cup \{\sharp t |\ t \in \mathcal{X}_T\} \wedge\ (E \neq M) \\ \text{true} & \text{otherwise} \end{cases}$$

We use the standard notion of *substitution* applied to a process P, $P[E/M]$. Note that when used in pattern matching, it has an effect only on definition variables $\natural x$ and definition tag variables $\sharp t$; in the other cases the substitution function

coincides with the identity function. Furthermore, notice that each definition variable occurs at most once in each pattern matching.

The judgement $\Gamma \vdash P \rightarrow_{\mathcal{R}} P'$ means that the process P can evolve into P', given the tag environment Γ. The rule (Com) expresses that an out-

$$(Com)\quad \frac{\wedge_{i=1}^{k} comp(E_i, M_i)}{\Gamma \vdash \langle E_1, \ldots, E_k \rangle.P \mid (M_1, \ldots, M_k).Q \rightarrow_{\mathcal{R}} P \mid Q[E_1/M_1, \ldots, E_k/M_k]}$$

$$(Dec)\quad \frac{E_0 = E_0' \ \wedge \ \wedge_{i=1}^{k} comp(E_i, M_i) \ \wedge \ \wedge_{i=1}^{k} \mathcal{R}(E_i, M_i, \Gamma)}{\Gamma \vdash decrypt\ \{E_1, \ldots, E_k\}_{E_0}\ as\ \{M_1, \ldots, M_k\}^{l}_{E_0'}\ in\ P \rightarrow_{\mathcal{R}} P[E_1/M_1, \ldots, E_k/M_k]}$$

$$(Tag\ Decl)\quad \frac{\Gamma[\sharp t \mapsto Tag] \vdash P \rightarrow_{\mathcal{R}} P'}{\Gamma \vdash (\nu\ \sharp t : Tag)P \rightarrow_{\mathcal{R}} (\nu\ \sharp t : Tag)P'} \qquad (Res)\quad \frac{\Gamma \vdash P \rightarrow_{\mathcal{R}} P'}{\Gamma \vdash (\nu\ n)P \rightarrow_{\mathcal{R}} (\nu\ n)P'}$$

$$(Par)\quad \frac{\Gamma \vdash P_1 \rightarrow_{\mathcal{R}} P_1'}{\Gamma \vdash P_1 \mid P_2 \rightarrow_{\mathcal{R}} P_1' \mid P_2} \qquad (Congr)\quad \frac{P \equiv P' \ \wedge \ \Gamma \vdash P' \rightarrow_{\mathcal{R}} P'' \wedge P'' \equiv P'''}{\Gamma \vdash P \rightarrow_{\mathcal{R}} P'''}$$

Table 1. Operational semantics, $\Gamma \vdash P \rightarrow_{\mathcal{R}} P'$, parameterised on $\mathcal{R}$.

put $\langle E_1, \ldots, E_k \rangle.P$ is matched by an input $(M_1, \ldots, M_k)$ by checking whether the closed terms M_i are pairwise the same with the corresponding E_i (i.e. if $comp(E_i, M_i)$). When the matchings are successful, the remaining E_j are bound to the corresponding M_j (that are use variables or tag variables).

Similarly, the rule (Decr) expresses the result of matching an encryption $\{E_1, \ldots, E_k\}_{E_0}$ against $\mathsf{decrypt}\ E\ \mathsf{as}\{M_1, \ldots, M_k\}^{l}_{E_0'}\ \mathsf{in}\ P$. As was the case for communication, the closed terms M_i must match the corresponding E_i, and additionally the keys must be the same. Note that the key cannot be a definition variable: it has to be matched in order to decrypt. When the matching is successful the remaining terms E_j are bound to the corresponding M_j (that are definition variables or definition tag variables). Recall that in the *reference monitor semantics* we ensure that the components of the decrypted message have the tags expected, by checking whether tag variables, e.g. $\sharp t$, are bound to the tags included in the corresponding tag environment, e.g. $\Gamma(\sharp t)$. In the *standard semantics* the condition $\mathcal{R}(E, M, \Gamma)$ is universally true and thus can be ignored.

Back to our example processes R, $\tilde{R}$, and S, using the *reference monitor semantics*, we have that in R $comp(ID_2, \sharp t_{id}) = comp(z, \natural z_k) = comp(key, \sharp t_k) = true$ and $\mathcal{R}(key, \sharp t_k, \Gamma) = true$ and $\mathcal{R}(ID_2, \sharp t_{id}, \Gamma) = true$ (because $\Gamma(\sharp t_k) = key$ and $\Gamma(\sharp t_{id}) = ID_2$), while in $\tilde{R}$ $comp(z, \natural z_k) = comp(nonce, \sharp t_k) = true$, but $\mathcal{R}(nonce, \sharp t_k, \Gamma) = false$ (because $nonce \neq \Gamma(\sharp t_k)$) and also $\mathcal{R}(ID_3', \sharp t_{id}, \Gamma) = false$ (because $nonce \neq \Gamma(\sharp t_{id})$). Finally, in S $comp(t, key) = true$ only if

$t = key$, $comp(t_{id}, ID_2) = true$ only if $t_{id} = ID_2$, if both are true then $\natural z_k$ is bound to z.

The rule (Tag Decl) records the new association between the tag variable $\sharp t$ and the tag Tag in the tag environment Γ. The updating of Γ is indicated as $\Gamma[\sharp t \mapsto Tag]$. The rules (Repl), (Par) and (Congr) are standard.

Dynamic Property As for the dynamic property of processes, we call a process *tag coherent*, if the process respects the declared tags and therefore is free of tag violations. This amounts to saying that each computation possible in the standard semantics is also possible in the reference monitor semantics. In turn, the reference monitor will never stop any execution step, when in all computations, each tag variable is bound to the expected tag, more precisely when each term has the expected type and the decryption is performed on the encryption generated in the correct protocol and in the correct step of the protocol. Actually, we assume that any attacker cannot modify the contents of an encryption, unless possessing the encryption key. Thus, we only consider tag violations arising inside encryptions and decryptions. As usual, $*$ stands for the transitive and reflexive closure of the transition relation.

Definition 1. *A process P is* tag coherent *if for all executions $\Gamma \vdash P \rightarrow^* P' \rightarrow P''$, then $\Gamma \vdash P \rightarrow^*_{\mathsf{RM}} P' \rightarrow_{\mathsf{RM}} P''$.*

3 Static Analysis

We develop a Control Flow Analysis for tagged LySA processes that safely over-approximates all the possible protocol behaviour. The result of the analysis of a process P also permits to safely approximate when the reference monitor may abort the computation of P. The approximation is represented by a tuple $(\Gamma, \rho, \kappa, \psi)$ (resp. a pair (ρ, ϑ) when analysing a term E), called *estimate* for P (resp. or E), that satisfies the judgements defined by the axioms and rules of Table 2. In particular, the analysis records which value tuples may flow over the network and which values may be bound to each *definition variable* (e.g. $\natural x$) and *definition tag variable* (e.g. $\sharp t$). Moreover, at each decryption place, the analysis checks whether a *tag* (e.g. *nonce* or ID_i) bound to each *definition tag variable* is the intended one, or a violation is reported. The analysis is defined in the flavour of Flow Logic [25].

Analysis of Terms The judgement for analysing terms is $\rho \models E : \vartheta$. The analysis keeps track of the potential values of *variables* x or *tag variables* t, by recording them into the global *abstract environment* ρ that maps variables and tag variable to the sets of values that they may be bound to: $\rho : \mathcal{X}_{\mathcal{S}} \cup \mathcal{X}_{\mathcal{T}} \rightarrow \wp(Val)$. The judgement is defined by the axioms and rules in the upper part of Table 2. The rules describe the analysis of terms which approximates the set of values ϑ that a term may evaluate to. A name n (a tag Tag, resp.) evaluates to the set ϑ, provided that n (Tag, resp.) belongs to ϑ. Similarly for a variable x (a

tag variable t, resp.), provided that ϑ includes the set of values $\rho(x)$ ($\rho(t)$, resp.) to which x (t, resp.) is associated with. To reduce the number of rules, we use the parameter N that stands for both the generic name n and for the generic tag Tag, and the parameter X that stands for both the generic variable x and for the generic tag variable t.

To produce the set ϑ, the rule for (Encr) (i) finds the sets ϑ_i for each term E_i, (ii) collects all k-tuples of values $(v_0, \cdots, v_k)$ taken from $\vartheta_0 \times \cdots \times \vartheta_k$ into values of the form $\{v_1, \cdots, v_k\}^l_{v_0}$ (iii) requires these values to belong to ϑ.

Analysis of Processes In the analysis of processes, the information on the possible values, that may flow over the network, is collected into the *abstract network environment* component $\kappa \subseteq \wp(Val^*)$ that includes all the value-tuples forming a message that may flow on the network.

The judgement for processes takes the form: $\rho, \kappa, \Gamma \models P : \psi$, where the components ρ, κ, and Γ are as above (recall that $\Gamma : \mathcal{X}_T \rightarrow \mathbf{Tag}$), while the component $\psi \subseteq \mathcal{C}$, is the (possibly empty) set of "error messages" that take the form of decryption labels l: $l \in \psi$ means that a tag-mismatching (or violation) may happen inside the decryption labelled l. The judgement is defined by the axioms and rules in the lower part of Table 2 (where $X \Rightarrow Y$ means that Y is only evaluated when X is *True*) and are explained later on.

For keeping the analysis component finite, as said before, we have partitioned all the names used by a process into finitely many equivalence classes and we have used the names of the equivalence classes instead of the actual names.

Before commenting on the analysis rules, we describe three auxiliary functions that generate some logic formulas to be used in the analysis rules.

The first one is the *matching* function, that takes care of pattern matching, by checking whether a value v corresponds to a term M. Remember that pattern matching cannot be performed on either $\natural x$ or $\sharp t$. If this is not the case, matching succeeds when v is a possible evaluation of the term M.

$$match(v, M, \rho) = \begin{cases} \text{false} & \text{if } M \notin \{\natural x |\ x \in \mathcal{X}_S\} \cup \{\sharp t |\ t \in \mathcal{X}_T\} \wedge (v \notin \vartheta) \\ & \text{where } \vartheta \text{ is s.t. } (\rho \models M : \vartheta) \\ \text{true} & \text{otherwise} \end{cases}$$

For instance, when $\rho \models x : \vartheta$, $match(n, x, \rho)$, is true if $n \in \vartheta$, while is false if $n \notin \vartheta$; $match(m, \natural x, \rho)$ is true, instead.

The second one is a *substitution* function, that takes care of variable binding. Intuitively, it only makes sense to bind a value or a tag to either a *definition variable* or a *definition tag variable*, respectively. So the substitution function binds the value v to M only when M is a variable $\natural x$ or a tag variable $\sharp t$.

$$sub(v, M, \rho) = \begin{cases} \text{false if } (v \notin \rho(M)) \text{ with } M \in \{\natural x |\ x \in \mathcal{X}_S\} \cup \{\sharp t |\ t \in \mathcal{X}_T\} \\ \text{true otherwise} \end{cases}$$

For example, $sub(m, \natural x, \rho)$ is true if $(m \in \rho(x))$, while $sub(m, m, \rho)$ is true.

The last function is about *tag checking*. Given a tag environment Γ, it checks whether v is the expected tag of a *definition tag variable* $\sharp t$. If it is not the case,

the decryption labeled l, is recorded in the error component ψ. Note that in order to let the tag checking work, M has to be a definition tag.

$$chk(v, M, \Gamma, l, \psi) = \begin{cases} \text{false if } M \in \{\sharp t |\ t \in \mathcal{X}_T\} \ \wedge\ (v \neq \Gamma(M)) \ \wedge\ (l \in \psi) \\ \text{true otherwise} \end{cases}$$

For instance, if $m \neq \Gamma(\sharp t)$, then $chk(m, \sharp t, \Gamma, l, \psi)$ is false and $l \in \psi$.

$$(Const)\ \frac{N \in \vartheta}{\rho \models N : \vartheta}\ (N = Tag \text{ or } n) \qquad (Var)\ \frac{\rho(X) \subseteq \vartheta}{\rho \models X : \vartheta}\ (X = x \text{ or } t)$$

$$(Encr)\ \frac{\wedge_{i=0}^{k}\ \rho \models E_i : \vartheta_i \ \wedge \quad \forall v_0, \ldots, v_k : \wedge_{i=0}^{k}\ v_i \in \vartheta_i \Rightarrow \{v_1, \ldots, v_k\}_{v_0} \in \vartheta}{\rho \models \{E_1, \ldots, E_k\}_{E_0} : \vartheta}$$

$$(Out)\ \frac{\wedge_{i=1}^{k}\ \rho \models E_i : \vartheta_i \ \wedge \quad \forall v_1, \ldots, v_k : \wedge_{i=1}^{k} v_i \in \vartheta_i \Rightarrow \langle v_1, \ldots, v_k \rangle \in \kappa \ \wedge \quad \rho, \kappa, \Gamma \models P : \psi}{\rho, \kappa, \Gamma \models \langle E_1, \ldots, E_k \rangle.P : \psi}$$

$$(In)\ \frac{\forall \langle v_1, \ldots, v_k \rangle \in \kappa \ \wedge_{i=1}^{k} (match(v_i, M_i, \rho) \Rightarrow sub(v_i, M_i, \rho) \ \wedge \quad \rho, \kappa, \Gamma \models P : \psi)}{\rho, \kappa, \Gamma \models (M_1, \ldots, M_k).P : \psi}$$

$$(Dec)\ \frac{\begin{array}{c} \rho \models E : \vartheta \ \wedge\ \rho \models E_0 : \vartheta_0 \ \wedge \\ \forall \{v_1, \ldots, v_k\}_{v_0} \in \vartheta : v_0 \in \vartheta_0 \Rightarrow \\ \wedge_{i=1}^{k} (match(v_i, M_i, \rho) \Rightarrow (sub(v_i, M_i, \rho) \ \wedge\ chk(v_i, M_i, \Gamma, l) \ \wedge \\ \rho, \kappa, \Gamma \models P : \psi) \end{array}}{\rho, \kappa, \Gamma \models decrypt\ E\ as\ \{M_1, \ldots, M_k\}^{l}_{E_0}\ in\ P : \psi}$$

$$(TNew)\ \frac{(\sharp t, Tag) \in \Gamma \ \wedge\ \rho, \kappa, \Gamma \models P : \psi}{\rho, \kappa, \Gamma \models (\nu\ \sharp t : Tag)P : \psi} \qquad (Par)\ \frac{\rho, \kappa, \Gamma \models P_1 : \psi \ \wedge\ \rho, \kappa, \Gamma \models P_2 : \psi}{\rho, \kappa, \Gamma \models P_1 \mid P_2 : \psi}$$

$$(Res)\ \frac{\rho, \kappa, \Gamma \models P : \psi}{\rho, \kappa, \Gamma \models (\nu\ n)P : \psi} \qquad (Rep)\ \frac{\rho, \kappa, \Gamma \models P : \psi}{\rho, \kappa, \Gamma \models !P : \psi} \qquad (Nil)\ \ \rho, \kappa, \Gamma \models 0 : \psi$$

Table 2. Analysis of tagged Lysa Terms: $\rho \models E : \vartheta$, and Processes: $\rho, \kappa, \Gamma \models P : \psi$

We now briefly comment on the rules for analysing processes. In the premises of the rule for k-ary *output* (Out), we require that all the terms are abstractly evaluated, and that all the combinations of these values are recorded in κ. Indeed these are the values that may be communicated. Finally, the continuation process must be analysed.

The rule (In) describes the analysis of pattern matching *input* and uses both the *match* and *substitution* functions. The idea is to examine all the sequences of $\langle v_1, ..., v_k \rangle$ in the κ component and to pointwise compare them against the tuple of definition terms $(M_1, ..., M_k)$. The *matching* function selects only the closed terms (names or tags) and for each of them, say M_i, checks whether the corresponding v_i is included in ϑ_i, i.e. the result of the analysis for M_i. If the matching succeeds for all the closed terms, then, the substitution function

takes care of binding the remaining values v_j to the corresponding *definition variables* or *definition tag variables* M_j. Moreover, the continuation process must be analysed.

The rule for *decryption* (Dec) is quite similar to the rule for *input*: matching and substitution are handled in the same way. The values to be matched are those obtained by evaluating the term E and the definition ones are the terms inside the decryption. If the matching succeeds for all closed terms, then the substitution is applied to the remaining values that are bound to the corresponding definition variables or definition tag variables. When processing the *substitution*, *tag checking* is also performed to capture possible violations. These occur when a *definition tag variable* is bound to an unexpected tag. In this case, the label l of the decryption is recorded in the error component ψ. In the case of both input and decryption the continuation process P is analysed only when the input or decryption could indeed succeed.

The rule for *tag declaration* (TNew) requires that the declared tag is recorded in the tag environment Γ. The rule for the *inactive process* (Nil) does not restrict the analysis result, while the rules for *parallel composition* (Par), *restriction* (Res), and *replication* (Rep) ensure that the analysis also holds for the immediate subprocesses.

Semantic properties Our analysis is semantically correct regardless of the way the semantics of LySA is parameterised. More precisely, we proved a subject reduction theorem for both the standard and the reference monitor semantics: $(\rho, \kappa, \psi, \Gamma)$ for P is a valid estimate also for all the states passed through in a computation of P, i.e. for all the derivatives of P.

Theorem 1 (Subject reduction). *If* $\Gamma \vdash P \rightarrow Q$ *and* $\rho, \kappa, \Gamma \models P : \psi$ *then* $\rho, \kappa, \Gamma \models Q : \psi$. *Furthermore, if* $\psi = \emptyset$ *then* $P \rightarrow_{\mathsf{RM}} Q$

In addition, when analysing a process P if the error component ψ is empty then the reference monitor *cannot stop* the execution of P. This means that our analysis correctly predicts when we can safely do without the reference monitor. We shall say that the reference monitor RM *cannot abort* a process P whenever there exist no Q, Q' such that $P \rightarrow^* Q \rightarrow Q'$ and $P \rightarrow^*_{\mathsf{RM}} Q \not\rightarrow_{\mathsf{RM}}$, where $Q \not\rightarrow_{\mathsf{RM}}$ stands for $\neg\exists Q' : Q \rightarrow_{\mathsf{RM}} Q'$. We then have:

Theorem 2 (Static check for reference monitor). *If* $\rho, \kappa, \Gamma \models P : \psi$ *and* $\psi = \emptyset$ *then RM cannot abort* P.

Example As shown in [4], our analysis acts in a descriptive way: it describes which violations may occur. In the same setting, our approach also offers a prescriptive usage: we can impose a tag discipline, by forcing some data to correspond to the expected tags. At this point, the analysis may statically check that tag violations are not possible any longer. In other words, we can instrument the code with the only checks necessary to enforce tag security. We can illustrate it on our example processes R, $\tilde{R}$, and S. For ***detecting*** possible tag violations, in $\tilde{R}$,

we assume that the t_k should be *key*, but we do not impose it in the protocol specification, we do not check the tag associated with the received value. The analysis, instead, correctly reveals that the tag received, may be *nonce*. For ***preventing*** such a tag violation and the consequent attack from arising, the protocol has to be modelled differently, by explicitly requiring that t_k has to be a key. More precisely, we check whether the tag associated with the value received and decrypted is *key* as expected. As a consequence, the decryption fails and the analysis result does not find potential tag violations any longer.

Modelling the Attacker In a protocol execution, several principals exchange messages over an open network, which is accessible to the attackers and therefore vulnerable to malicious behaviour. We assume an active Dolev-Yao attacker [11]: it can eavesdrop, and replay, encrypt, decrypt, generate messages providing that the necessary information is within his knowledge, that it increases while interacting with the network.

This scenario can be modelled in LySA as a process running in parallel with the protocol process. Formally, we shall have $P_{sys} \mid P_\bullet$, where P_{sys} represents the protocol process and $P_\bullet$ is some *arbitrary* attacker. To get an account of the infinitely many attackers, the overall idea is to find a formula $\mathcal{F}$ (for a similar treatment see [7]) that characterizes all $P_\bullet$: this means that whenever an estimate $(\rho, \kappa, \Gamma, \psi)$ satisfies $\mathcal{F}$, then $(\rho, \kappa, \Gamma) \models P_\bullet : \psi$ for all attackers $P_\bullet$. Intuitively, the formula has to mimic how all the $P_\bullet$ are analysed. The attacker process is parameterised on some attributes of P_{sys}, e.g. the length of all the encryptions occurred and all the messages sent over then network. In the formula, the names and variables the attacker uses are apart from the ones used by P_{sys}. We can then postulate a new distinguished name $n_\bullet$ and a new distinguished variable $z_\bullet$, in which the names and variables, resp., of the attacker are coalesced; therefore $n_\bullet$ may represent any name generated by the attacker, while $\rho(z_\bullet)$ represents the attacker knowledge. It is possible to prove that if an estimate of a process P with $\psi = \emptyset$ satisfies the attacker formula than RM does not abort the execution of $P \mid Q$, regardless of the choice of the attacker Q. Further details can be found in [7].

Implementation and Complexity The overall goal of the implementation of the analysis is to compute the analysis result $(\rho, \kappa, \Gamma, \psi)$ for a given process. This is done in two phases. In the first phase, construct a function using Standard ML that translates a LySA process into a logic formula in Alternation-free Least Fixed Point logic [24], which is regarded as an extension of Horn clauses, and in the second phase use Succinct Solver to compute interpretations of predicates that satisfy the formula. As the Succinct Solver is used to compute a finite representation of the analysis result, according to the Proposition 2 in [24], it is easy to draw a conclusion that: a finite representation of an analysis result for $\rho, \kappa, \Gamma \models P : \psi$ may be computed in *low polynomial time* in the size of the process P.

4 Multi-protocol tagging

We are able to statically detected a family of similar attacks in a setting where the (secure in isolation) Bauer-Berson-Feiertag (BBF) [3] and Woo-Lam Authentication Π_f (WL) [27] protocols are running in parallel, using the same long-term keys (they both use symmetric encryption). They are all based on the fact that the initiator of the WL can be exploited as an oracle to produce an encryption that can be confused with the encryption including the new session key in the BBF protocol, thus attacking the secrecy and authentication of BBF. For lack of space, we only show one of these attacks. In this setting, each principal can participate in different protocols and the trusted server S serves in both protocols. In BBF, S is used to generate and distribute a new session key, while in WL S acts as intermediary between I and R.

In BBF, the initiator, I, sends its identifier and a new fresh nonce to R which forwards them to the trusted server, together with its own identifier and nonce. The server creates the new session key and replays back to R two encrypted messages; R decrypts the one encrypted with the long-term key, K_{RS}, verifies that the nonce is the same generated earlier in the session, and forwards the other encryption to I, that can decrypt it and check for the nonce. At the end, both should obtain the new session key.

The initiator I of the WL protocol sends R its identifier and receives back from him its nonce. Then I sends to R its nonce and both the principal identifiers, encrypted with its long-term key K_{IS}. The responder contacts the server S by sending him an encryption containing the I's message together with its nonce and the principal identifiers. Eventually, S confirms the identity of I to R by sending R an encryption containing the fresh nonce generated in the second step of WL. The narration of the two protocols follows.

BBF	WL
1. $I \rightarrow R : I, N_I$	1. $I \rightarrow R : I$
2. $R \rightarrow S : I, N_I, R, N_R$	2. $R \rightarrow I : N_R$
3. $S \rightarrow R : \{K_{IR}, I, N_R\}_{K_{RS}}, \{K_{IR}, N_I, R\}_{K_{IS}}$	3. $I \rightarrow R : \{I, R, N_R\}_{K_{IS}}$
4. $R \rightarrow I : \{K_{IR}, N_I, R\}_{K_{IS}}$	4. $R \rightarrow S : \{I, R, N_R, \{I, R, N_R\}_{K_{IS}}\}_{K_{RS}}$
	5. $S \rightarrow R : \{I, R, N_R\}_{K_{IS}}$

In Table 3 we show the specification of the two protocols, where each message begins with the pair of roles involved in the exchange. For the sake of simplicity, we only specify the single roles in each protocol. Actually, each agent can participate in both protocols and can play in both the initiator and responder roles.

The attack arises in a scenario where the principal B begins a run of WL as initiator with A as a responder. His messages are intercepted by the attacker ($M(X)$ stands for the attacker impersonating the principal X). The principal B is then involved as responder in a run of the protocol BBF, apparently with A as initiator, but actually with $M(A_I)$. The attacker exploits the fact that the long-term key K_{BS} between B and S is the same in both the protocols.

/*Role I in BBF*/	$(\nu\ N_I)(\nu\ \sharp tx_{k1} : KEY)(\nu\ \sharp txp : BBF_3)$
/*1*/	$\langle I, R, I, N_I \rangle.$
/*4*/	$(R, I, \flat\ x_{e1}).decrypt\ x_{e1}\ as\{\sharp txp, (\sharp tx_{k1}, \flat x_{k1}), (nonce, N_I), (agent, R)\}^{\ell_1}_{K_{IS}}\ in\ 0$
/*Role I in WL*/	
/*1*/	$\langle I, R, I \rangle.$
/*2*/	$(R, I, \flat x_n).$
/*3*/	$\langle I, R\{WL_3, (agent, I), (agent, R), (nonce, x_n)\}_{K_{IS}} \rangle.0$
/*Role R in BBF*/	$(\nu\ N_R)(\nu\ \sharp ty_p : WL_2)(\nu\ \sharp ty_k : KEY)$
/*1*/	$(I, R, \flat y_1, \flat y_2).$
/*2*/	$\langle R, S, y_1, y_2, R, N_R \rangle.$
/*3*/	$(S, R, \flat y_{e1}, \flat y_{e2}).decrypt\ y_{e1}\ as\{\sharp ty_p, (\sharp ty_k, \flat y_k), (agent, y_1), (nonce, N_R)\}^{\ell_2}_{K_{RS}}\ in$
/*4*/	$\langle R, I, y_{e2} \rangle.0$
/*Role R in WL*/	$(\nu\ N'_I)(\nu\ \sharp ty_{p1} : WL_5)(\nu\ \sharp ty_{a2} : AGENT)$
/*1*/	$(I, R, \flat y_{a1}).$
/*2*/	$\langle R, I, N'_R \rangle.$
/*3*/	$(I, R, \flat y_{e3}).$
/*4*/	$\langle R, S, \{(agent, y_{a1}), (agent, R), (nonce, N'_R), y_{e3}\}_{K_{RS}} \rangle.$
/*5*/	$(S, R, \flat y_{e4}).decrypt\ y_{e4}\ as\{\sharp ty_{p1}, (agent, y_{a1}), (agent, R), (nonce, N'_R)\}^{\ell_3}_{K_{BS}}\ in\ 0$
/*Server in BBF*/	$(\nu\ K1)$
/*2*/	$(R, S, I, \flat s_{n1}, R, \flat s_{n2}).$
/*3*/	$\langle S, R, \{BBF_3, (key, K1), (agent, I), (nonce, s_{n2}\}_{K_{RS}},$ $\{BBF_3, (key, K1), (nonce, s_{n1}), (agent, R)\}_{K_{IS}} \rangle.0$
/*Server in WL*/	$(\nu\ \sharp tz_{p1} : WL_4)(\nu\ \sharp tz_{p2} : WL_3)(\nu\ \sharp tz_{n1} : NONCE)$
/*4*/	$(R, S, \flat z_{e1}).decrypt\ z_{e1}\ as\{\sharp tz_{p1}, (agent, I), (agent, R), (\sharp tz_{n1}, \flat z_{n1}), \flat\ z_{e2}\}^{\ell_4}_{K_{RS}}\ in$ $decrypt\ z_{e2}\ in\{\sharp tz_{p2}, (agent, I), (agent, R), (tz_{n1}, z_n)\}^{\ell_5}_{K_{IS}}\ in$
/*5*/	$\langle S, R, \{WL_5, (agent, I), (agent, R), (nonce, z_{n1})\}_{K_{RS}} \rangle.0$

Table 3. Specification of the two protocols

More precisely, the intruder suitably mixes the nonces of the two protocols and then exploits B, in the role of initiator in WL, as an oracle to get a message composed with three terms and encrypted with K_{BS} ($\{B, A, N_B\}_{K_{BS}}$), that can be wrongly accepted by B, in the role of responder in BBF, as the encryption containing the new session key, the name of the initiator and the nonce generated in the second step of BBF. As a consequence B, in the BBF session, believes to have communicated with A and that the new session key is B.

$/*1.WL*/\quad B \rightarrow M(A) : B$
$/*1.BBF*/\ M(A) \rightarrow B : A, N_A$
$/*2.BBF*/\ B \rightarrow M(S) : A, N_A, B, N_B$
$/*2.WL*/\quad M(A) \rightarrow B : N_B$
$/*3.WL*/\quad B \rightarrow M(A) : \{B, A, N_B\}_{K_{BS}}$ *$/*$ B acts as an oracle for the intruder $*/$*
$/*3.BBF*/\ M(S) \rightarrow B : \{B, A, N_B\}_{K_{BS}}, \{B, A, N_B\}_{K_{BS}}$ *$/*$ B accepts a wrong key $*/$*
$/*4.BBF*/\ B \rightarrow M(A) : \{B, A, N_B\}_{K_{BS}}$

In the LySA specification, a system sufficient to capture the above attack, is defined as the parallel composition of three processes A, B, and S, running in parallel within the scope of the shared keys: $System = (\nu K_{AS})(\nu K_{BS})A\|B\|S$, where A and B can play as initiator and responder in both protocols, and the actions in the two roles in the two protocols run in parallel. In particular, we can focus on B in the role of initiator in WL and B in the role of responder in BBF.

Our analysis correctly detects the message confusion that occurs at step 3 of the BBF protocol. In fact, the results, limited to the variables of interest, listed below, shows that there has been a tag violation ($l_2 \in \psi$) in the encryption, where an agent identifier, B has been accepted as a key ($\{nonce\} \in \rho(ty_k)$, but $\Gamma(ty_k) = key$). Furthermore, from $\{WL_3\} \in \rho(ty_p)$ and $\Gamma(ty_p) = BBF_3$ we can say that a message created in WL at step 3 has been instead accepted in BBF at step 3.

$$\begin{array}{l} \{\{BBF_3, (key, K1), (agent, A), (nonce, N_B)\}_{K_{BS}}, \\ \{WL_3, (agent, B), (agent, A), (nonce, N_B)\}_{K_{BS}}\} \in \rho(y_{e1}) \end{array}$$
$$\{BBF_3, WL_3\} \in \rho(ty_p) \quad \{key, nonce\} \in \rho(ty_k) \quad \{K_1, B\} \in \rho(ty_k) \quad \{l_2\} \in \psi$$

Note that the above results can be obtained by the specification in Table 3, by replacing I and R in the third step of BBF with A and B, resp.

Furthermore, if we replaced ty_p with BBF_3, i.e. if we forced the decrypted value to match the tag BBF_3, as in

$$decrypt\ y_{e1}\ as\ \{BBF_3, (\sharp ty_k, \natural ty_k), (agent, y_1), (nonce, N_B)\}^{\ell_2}_{K_{BS}}\ in\ \ldots$$

then the attack could be prevented at run time. The analysis of the new specification would be indeed such that $\psi = \emptyset$. Intuitively, in the third step of BBF, B could still receive $\{B, A, N_B\}_{K_{BS}}$, but the decryption would fail, because of pattern matching, in fact $comp(BBF_3, WL_3) = false$. This is an example of instrumentation of the code, used to introduce the only necessary checks on tags.

5 Conclusion

In the multi-protocol setting, harmful interactions among different protocols often are based on the presence of similar patterns in messages that can be exploited by attackers. As an example, we modelled and analysed an undocumented family of attacks that occur when the Bauer-Berson-Feiertag and the Woo-Lam authentication protocols are concurrently executed. We can also statically detect some of the attacks reported in [10], that we do not present here for lack of space (an example can be found in [5]).

We extended the process calculus LySA with tags, which represent the intended types of terms, the intended protocol and the message step. The semantics uses a *reference monitor* to capture tag-mismatching at run time. We developed a Control Flow Analysis to check at each decryption place whether the received, secret data has the right type and the message has been generated in the same

protocol, in the correct step. Our tagging schema let us therefore detect possible multi-protocol attacks. In addition, we can impose a prescriptive tag discipline, by using tags at run time to semantically force some data – *only* the ones that can be confused – to be accepted only if they correspond to the expected ones: in this case, we aim at *preventing* tag violation to arise. Attaching type tags and protocol identifiers within encrypted messages is a classical countermeasure to prevent this kind of attacks. It is often unnecessary and redundant to tag everything. The suggestions given by the analysis on how to instrument the code with the necessary checks can thus lead to useful insights on the optimal usage of tags at run time, identifying the redundant ones.

The Control Flow Analysis presented here is based on a particular kind of tags. There are a lot of different tagging schemes in literature that can be included in our framework as well, e.g. the just referred [20] or the ones proposed in [10].

In the last years, LySA has been given different kinds of annotations for checking some classic security properties, e.g. confidentiality [14], freshness [12] and message authentication [7]. It is very easy to combine tags with the above annotations, thus obtaining a more general form of analysis. We therefore obtain a single unifying framework for studying protocols. Our analysis has been implemented and can be computed in low polynomial time in the size of the process under consideration.

Studies on multi-protocol systems are usually focused on analysing possible interactions between two, or a few more, specific protocols running in parallel, as in [10]. There, verification techniques exploiting the Scyther Tool [9] are presented, in order to check the security properties of small groups of protocols, and to discover new multi-protocol attacks. Scyther can either compute a fixed number of runs or simulate an unbounded number of runs: if the studied property is not verified an attack is reconstructed. The termination is always guaranteed. Our static approach guarantes that the over-approximated results covers all the attacks both for a bounded or unbounded number of runs. We have termination, because ours is a static analysis: the price is a loss in precision, that dynamic techniques avoid. Model checking has been instead used in [26] to discover new attacks. This technique has the disadvantage of state-explosion, not suffered by Control Flow Analysis, again because of approximation. Also [21], presents a certain number of new multi-protocol attacks. Similarly to [20], we apply a uniform tagging scheme for all the protocols in the system. We remark that our tags just identify the expected types of message components and protocol identifiers, instead of roles, *identity, authentication, verification*, as in [20]. There is also a line of research devoted to establishing conditions for composition of protocols. In [15], authors define two protocols independent, when the achievement of a security goal of one protocol does not depend on the execution of the other. Using the Strand Space formalism [16], they prove that protocol independence is obtainable if encryption is used in non-overlapping way. In the same line of research is [6], where it is shown that secrecy-preserving protocols can be safely composed, if each encryption comes with a tag identifying the protocol name.

References

1. M. Abadi and A.D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation, 148(1)*, pp.1-70, 1999.
2. J. Alves-Foss. Multi-Protocol Attacks and the Public Key Infrastructure. *In Proceedings of National Information System Security Conference*, 566–576, 1998.
3. R.K. Bauer, T.A. Berson, and R.J. Feiertag. A key distribution protocol using event markers. *ACM Transactions on Computer Systems*, 1(3): 249-255, 1983.
4. C. Bodei, P. Degano, H. Gao, L. Brodo. Detecting and Preventing Type Flaws: a Control Flow Analysis with tags. *Proc. of 5th International Workshop on Security Issues in Concurrency (SecCO)*. ENTCS 194, pp. 3-22, 2007.
5. C. Bodei, P. Degano, H. Gao, L. Brodo. Detecting and Preventing Type Flaws at static time. Submitted.
6. V. Cortier, J. Delaitre and S. Delaune. Safely Composing Security Protocols. *Proc. of FSTTCS'07*, LNCS 4855, pp. 352-363
7. C. Bodei, M. Buchholtz, P. Degano, F. Nielson and H.R. Nielson. Static Validation of Security Protocols. *Journal of Computer Security*, 13(3), pp. 347 - 390, 2005.
8. U. Carlsen. Cryptographic Protocols Flaws. *Proc. of CSFW*: pp. 192-200, 1994.
9. C.J.F. Cremers. The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. To appear in CAV 2008.
10. C.J.F. Cremers. Feasibility of multi-protocol attacks. *Proc. of International Conference on Availability, Reliability and Security (ARES)*, pp. 287–294. IEEE Computer Society Press, 2006.
11. D. Dolev and A.C. Yao. On the Security of Public Key Protocols. IEEE TIT, IT-29(12):198-208, 1983.
12. H. Gao, C. Bodei, P. Degano and H.R. Nielson. A Formal Analysis for Capturing Replay Attacks in Cryptographic Protocols. *Proc. of the Asian Computing Science Conference (ASIAN'07)*. LNCS 4846, pp. 150-165, Springer, 2007.
13. H. Gao, C. Bodei, P. Degano. A Formal Analysis of Complex Type Flaw Attacks on Security Protocols. *Proc. of Algebraic Methodology and Software Technology (AMAST'08)*. LNCS, 2008.
14. H. Gao and H.R. Nielson. Analysis of LySa-calculus with explicit confidentiality annotations. *Proc. of Advanced Information Networking and Applications (AINA)*, IEEE Computer Society, 2006.
15. D. Guttman and F. Javier Thayer. Protocol independence through disjoint encryption. *Proc. of the 13th Computer Security Foundations Workshop*, pp. 24-34. IEEE Computer Society Press, 2000.
16. D. Guttman and F. Javier Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 238(2), pp. 24-34, 2002.
17. J. Heather, G. Lowe and S. Schneider. How to prevent type flaw attacks on security protocols. *Proc. of the 13th Computer Security Foundations Workshop (CSFW)*, IEEE Computer Society Press, 2000.
18. Y. Li, W. Yang and J. Huang. Preventing type flaw attacks on security protocols with a simplified tagging scheme. *Journal of Information Science and Engineering*, 21(1):59-84, 2005.
19. J. Kelsey, B. Schneier and D. Wagner. Protocol Interactions and and the Chosen Protocol Attack. *Proc. of Security Protocols Workshop*, LNCS 1361, Springer, 1998.
20. M. Maffei. Tags for Multi-Protocol Authentication. ENTCS 128(5), pp. 55-63, 2005.
21. A. Mathuria, A.R. Singh, P.V. Sharavan, R. Kirtankar. Some New Multi-Protocol Attacks. *In Proc. of Advanced Computing and Communications (ADCOM)*, 2007.

22. C. Meadows. Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trend. IEEE Journal of Selected Areas in Communications, 21(1), pp. 44-54, 2003.
23. R. Milner. Communicating and mobile systems: the π-calculus. Cambridge University Press, 1999.
24. F. Nielson, H. Riis Nielson, and H. Seidl. A succinct solver for ALFP. *Nordic Journal of Computing*, 9:335-372, 2002.
25. H.R. Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. *The Essence of Computation: Complexity, Analysis, Transformation* LNCS 2566: 223-244, Springer Verlag, 2002.
26. M. Panti, L. Spalazzi, S. Tacconi, R Pagliarecci, R. Model checking the security of multi-protocol systems. *Proc. of the 2005 International Symposium on Collaborative Technologies and Systems*, 2005.
27. T.Y.C. Woo and S.S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24-37, 1994.