Detecting and Preventing Type flaws: a Control Flow Analysis with tags [1]

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

[1] *Dipartimento di Informatica, Università di Pisa, Via Pontecorvo, I-56127 Pisa - Italia -* {chiara,degano}@di.unipi.it

[2] *Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads bldg 321, DK-2800 Kongens Lyngby - Denmark -* hg@imm.dtu.dk

[3] *Dipartimento di Scienze dei Linguaggi, Università di Sassari, via Tempio,9, I-07100 Sassari - Italia -* brodo@uniss.it

Abstract

A *type flaw attack* on a security protocol is an attack where an honest principal is cheated on interpreting a field in a message as the one with a type other than the intended one. In this paper, we shall present an extension of the LySa calculus with tags attached to each field, indicating the intended types. We developed a control flow analysis for analysing the extended LySa, which over-approximates all the possible behaviour of a protocol and hence is able to capture any type confusion that may happen during the protocol execution. The control flow analysis has been applied to a number of security protocols, either subject to type flaw attacks or not. The results show that it is able to capture type flaw attacks on those security protocols.

Keywords: Security Protocol, Control Flow Analysis, Type Flaw Attacks

1 Introduction

A *type flaw attack* on a security protocol arises when a field, originally intended to have one type, is instead interpreted as having another type. To prevent such attacks, the current techniques [11,12] consist in systematically associating each message field with a tag representing its intended type. Therefore fields with different types cannot be mixed up. Nevertheless, these may result in requiring extra and somehow unnecessary computational power and network transmission band. This is particularly heavy, when resources are limited such as in battery-powered embedded systems like PDAs, cell phones, laptops, etc.

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 the presence of a Dolev-Yao attacker [7]. The proposed approach abstracts the fields of protocol messages to a lower level, such that the misinterpretation can be formally

[1] This work has been partially supported by the project SENSORIA.

modeled. To this end, we extend the LySa calculus [2,3] with special tags, which represent the type of terms. The Control Flow Analysis approximates the behaviour of protocols in terms of the possibly exchanged messages and potential values of variables. The analysis can be working in either a *prescriptive* way, such that type flaws are avoided; or a *descriptive* way, such that type flaws are detected and recorded as violations of the intended types. Furthermore, if no type 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. It has been successfully applied to a number of protocols, such as Woo-Lam π_1[19] and Andrew Secure RPC [17].

LySa has been given different kinds of annotations for checking other security properties, e.g. confidentiality [9] and freshness [8]. It is very easy to combine tags with those techniques, thus giving a more comprehensive results of analysing security protocols.

The paper is organised as follows. In Section 2, we present the LySa calculus with tags for type flaw attacks, both the syntax and semantics are defined. We introduce the Control Flow Analysis in Section 3, which captures any type-mismatching that may happen. In Section 4, we show how the Control Flow Analysis works on two example protocols that are subject to type flaw attacks. In Section 5, we conclude with an assessment of our approach and a comparison with related work.

2 Calculus

The LySa calculus [2,3] is a process algebra, in the tradition of the π- [14] and Spi- [1] calculi. It differs from these essentially in two aspects. The first is the absence of channels: all processes have only access to a single global communication channel, the ether. The second aspect concerns the inclusion of pattern matching into the language constructs where values can become bound to values, i.e. into input and into decryption. This is different from having a separate matching construct, usually an if-then construct as in other process calculi and lead to more succinct specifications of protocols. We use here a dialect of LySa, which presents a more general pattern matching than the one in [2,3]. See also [5,16] for an alternative treatment.

Syntax of Terms

The basic blocks of LySa are values, used to represent agent names, nonces, keys. Syntactically, they are described by terms that may either be standard terms E or *matching terms* M. Standard terms – that can be names or variables – are used for modeling outputs and encryptions. Instead, for modeling inputs and decryptions we use matching terms, that, in turn, can be standard terms, or variables. 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 an use occurrence is an appearance of a variable where its binding value is used.

The 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. Furthermore, this notation distinguishes variables from occurrences of standard terms in tuples of matching terms, by implicitly partitioning them into standard terms or variables. In pattern matching, the first are checked for matching, while the others are bound in case of successful matching (see below).

S ::=	*standard terms*	$\mathcal{S}$ ::=	*matching standard terms*
n	name ($n \in \mathcal{N}$)	S	standard terms
x	use variable ($x \in \mathcal{X}_\mathcal{S}$)	$\natural x$	definition variable ($x \in \mathcal{X}_S$)

Here $\mathcal{N}$, $\mathcal{X}_S$, denote sets of names and of applied occurrences of variables, respectively. The name n is used to represent keys, nonces and names of principals.

Type Tagging We extend the syntax of standard LySa to cope with types, by using tags to represent the types of terms. Following [11], we assume to have a tag for each base type, such as *nonce*, *key*, etc. Moreover, we assume that the attacker is able to change only the types of terms that he can access. For malleability reasons, we choose to tag only encryptions and decryptions. In fact, by making the assumption of perfect cryptography, we have that only cleartext can be altered. Attackers can only forge an encryption when possessing the key used to cipher it.

$$\mathbf{Tag} \ni Tag ::= agent \mid nonce \mid key \mid \ \ldots$$

There are type 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 defining occurrences of type variables (in the form $\sharp t$), from the corresponding use occurrences (in the form t). Syntactically, we have the following two new categories, where $\mathcal{X}_T$ denote sets of applied occurrences of type variables.

T ::=	*type terms*	$\mathcal{T}$ ::=	*matching type terms*
Tag	type tags ($Tag \in \mathbf{Tag}$)	T	type terms
t	use type variable ($t \in \mathcal{X}_\mathcal{T}$)	$\sharp t$	defining type variable ($t \in \mathcal{X}_\mathcal{T}$)

Furthermore, we can merge the above syntactic categories with the ones for standard terms in order to obtain the two more general syntactic categories for terms E and matching terms M. Encryptions are tuples of terms $E_1, \cdots, E_k$ encrypted under a term E_0 representing a shared key.

E ::= *terms*		M ::=	*matching terms*
S	standard terms	$\mathcal{S}$	matching standard terms
T	type terms	$\mathcal{T}$	matching type terms
$\{E_1, \cdots, E_k\}_{E_0}$	symmetric encryption	$\{M_1, \ldots, M_k\}_E$	matching encryptions

We call Val the set of values, i.e. closed terms. Each value can have a type tag associated with it. From here on, for readability, we usually associate standard terms and type terms in encryptions and decryptions.

Syntax of Processes

In addition to the classical constructs for composing processes, our calculus also contains an input construct with matching and a decryption operation with matching. Furthermore, to keep track of the decryptions in which a violation occurs, we decorate each decryption with a label l (from a numerable set $\mathcal{C}$). Labels are mechanically attached to program points in which decryptions occur (they are nodes in the abstract syntax tree of processes). Finally, by overloading the symbol ν, we use a new process construct to declare the expected type of a type variable.

$P ::=$ *processes*

$\langle E_1, \ldots, E_k \rangle.P$	output
$(M_1, \ldots, M_k).P$	input
$\mathit{decrypt}\ E\ \mathit{as}\ \{M_1, \ldots, M_k\}^l_{E_0}\ \mathit{in}\ P$	decryption with matching
$(\nu\ n)P$	restriction
$(\nu\ \sharp t : \mathit{Tag})P$	type declaration
$P_1 \mid P_2$	parallel composition
$!P$	replication
$\mathsf{0}$	nil

The sets of free variables, resp. free names, and of bound variables and names, of a term or a process are written $\mathsf{fv}(\cdot)$, $\mathsf{fn}(\cdot)$, $\mathsf{bv}(\cdot)$, $\mathsf{bn}(\cdot)$, respectively. they are defined in the standard way. As usual, we omit the trailing $\mathsf{0}$ of processes.

Our patterns – in the form $(M_1, \cdots, M_k)$ – are matched against tuples of terms $(E_1, \cdots, E_k)$. Note that, at run time, each $(E_1, \cdots, E_k)$ only includes closed terms, i.e. each variable composing each one of the E_i has been bound in the previous computations. Instead, matching terms M_i can be partitioned in closed terms 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 two processes, where only standard terms are present.

$$P = \mathit{decrypt}\ \{A, w_n\}_K\ \mathit{as}\ \{\natural x_a, N_B\}^{l_P}_K\ \mathit{in}\ P' \qquad Q = \mathit{decrypt}\ \{A, N_B\}_K\ \mathit{as}\ \{\natural x_a, \natural y_n\}^{l_Q}_K\ \mathit{in}\ Q'$$

The decryption in P succeeds only if $w_n = N_B$: in this case $\natural x_a$ will be bound to A. Instead, the second decryption in Q always succeeds, and results in binding $\natural x_a$ to A, and $\natural y_n$ to N_B.

The roles played by tags and type variables in the pattern matching are the same played by terms and variables. Suppose, e.g. to have the following processes:

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

The decryptions in R and $\tilde{R}$ always succeed and result in binding $\natural z_k$ to (the values assumed by) z, and $\sharp t_k$ to *key* or to *nonce*. In particular, in $\tilde{R}$ the decryption succeeds, even though the declared type for $\sharp t_k$ is *key*. In the decryption in S only if t successfully matches with *key* then $\natural z_k$ is bound to z.

Operational Semantics

Below we slightly modify the standard *structural congruence* $\equiv$ on LySa processes, also to take care of type declarations. It is the least congruence satisfying the following clauses:

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

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 assume that values and variables are "stable", i.e. that for each value $n \in \mathcal{N}$ there is a canonical representative $\lfloor n \rfloor$ for the set $\{n, n_0, n_1, \cdots\}$ and similarly, for each variable $x \in \mathcal{X}_S \cup \mathcal{X}_T$ there is a canonical representative $\lfloor x \rfloor$ for the set $\{x, x_0, x_1, \cdots\}$. Then, we discipline α-conversion as follows: two values (resp. variables) are α-convertible only when they have the same canonical value (resp. variable). In this way, we statically maintain the identity of values and variables that may be lost by freely applying α-conversions. Hereafter, we shall simply write n (resp. x) for $\lfloor n \rfloor$ (resp. $\lfloor x \rfloor$).

Following the tradition of the π-calculus, we shall give LySa a reduction semantics. The *reduction relation* $\rightarrow_{\mathcal{R}}$ is the least relation on closed processes that satisfies the rules in Table 1. It uses structural congruence, as defined above, and the disciplined treatment of α-conversion. 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 type environment Γ, which maps a type variable in a set of tags.

$$\Gamma : \mathcal{X}_{\mathcal{T}} \rightarrow \wp(\mathbf{Tag})$$

One variant ($\rightarrow_{\mathsf{RM}}$) takes advantage of checks on type associations, while the other one ($\rightarrow$) discards them: essentially, the first semantics checks for type matching, while the other one does not (see below):

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

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

  This function affects only type variables, i.e. only matching terms M in the form $\sharp t$. It checks whether the type $(\Gamma(\sharp t))$ associated with the variable includes 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 two auxiliary functions that handle the difference between closed terms and variables to be bound, by implicitly partitioning the tuples and treating the respective elements differently. We use a slightly modified notion of *substitution* applied to a process P, $P[E/M]$, where M can be either $\natural x$ or $\sharp t$.

$$P[E/M] = \begin{cases} P[M \mapsto E] \text{ if } M \in \{\natural x |\ x \in \mathcal{X}_S\} \cup \{\sharp t |\ t \in \mathcal{X}_T\} \\ P \qquad\qquad\quad \text{otherwise} \end{cases}$$

The *pattern matching function* $comp(E, M)$ compares E against M only when M is a closed term and not a variable.

$$comp(E, M) = \begin{cases} \text{false} \ \text{ if } E \neq M \wedge \mathsf{fv}(M) \cup \mathsf{bv}(M) = \emptyset \\ \text{true} \ \text{ otherwise} \end{cases}$$

The judgement $\Gamma \vdash P \rightarrow_{\mathcal{R}} P'$ means that the process P can evolve into P', given the type environment Γ. The rule (Com) expresses that an output $\langle E_1, \ldots, E_k \rangle.P$ 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 variables or type variables).

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

$$(Dec) \quad \frac{\wedge_{i=0}^{k} comp(E_i, M_i) \ \wedge \ \wedge_{i=1}^{k} \mathcal{R}(E_i, M_i)}{\Gamma \vdash decrypt\ \{E_1, \ldots, E_k\}_{E_0}\ as\ \{M_1, \ldots, M_k\}^{l}_{E_0}\ in\ P \rightarrow_{\mathcal{R}} P[E_1/M_1, \ldots, E_k/M_k]}$$

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

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

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

Similarly, the rule (Decr) expresses the result of matching an encryption $\{E_1, \ldots, E_k\}_{E_0}$ with $\mathsf{decrypt}\ E\ \mathsf{as}\{M_1, \ldots, M_k\}^{l}_{E_0'}\ \mathsf{in}\ P$. As it was the case for communication, the closed terms M_i must match with the corresponding E_i, and additionally the keys must be the same. When the matching is successful the remaining terms E_j are bound to the corresponding M_i (that are definition variables or definition type variables). Recall that in the *reference monitor semantics* we ensure that the components of the decrypted message have the types expected, by checking whether the $\sharp t$ are bound to a type tag that is included in $\Gamma(\sharp t)$. In the *standard semantics* the condition $\mathcal{R}(E, M)$ is universally true and thus can be ignored. Back to our example processes R, $\tilde{R}$, S, we have that in R, $comp(z, \natural z_k) = comp(key, \sharp t) = true$ and $\mathcal{R}(key, \sharp t) = true$ (because $key \in \Gamma(\sharp t)$), while in $\tilde{R}$, $comp(z, \natural z_k) = comp(nonce, \sharp t) = true$, but $\mathcal{R}(nonce, \sharp t) = false$ (because $nonce \notin \Gamma(\sharp t)$). Note also that in S, $comp(t, key) = true$ only if $t = key$, and, in this case $P[z/\natural z_k] = P[\natural z_k \mapsto z]$.

The rule (Type Decl) records the new association between the type variable $\sharp t$ and the type Tag in the type 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 the process, we shall consider a process free of type flaw attack, when in all computations, each type variable is bound to the expected type. Consequently, the reference monitor will never stop any execution step. Note that we only consider the type flaws occurring inside encryptions and decrytpions.

Definition 2.1 A process P is *free of type flaw attacks* when for each step $\Gamma \vdash P \rightarrow^* P' \rightarrow P''$, we always have $\Gamma \vdash P \rightarrow^* P' \rightarrow_{\mathsf{RM}} P''$.

3 Static Analysis

We develop a control flow analysis for analysing tagged LySa processes. The aim of the analysis is to safely over-approximate all the possible protocol behaviour which permits to safely approximate when the reference monitor may abort the computation of a process 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. for 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 type variable* (e.g. $\sharp t$). Moreover, at each decryption place, the analysis checks whether a *type tag* (e.g. Tag) bound to each *definition type variable* is the intended one, or a violation is reported. The analysis is defined in the flavor of Flow Logic [15].

Analysis of Terms

The judgement for analysing terms is $\rho \models E : \vartheta$. The analysis keeps track of the potential values of *variables* or *type variables*, e.g. x or t, by recording them into the global *abstract environment* ρ:

- $\rho : \mathcal{X}_{\mathcal{S}} \cup \mathcal{X}_{\mathcal{T}} \rightarrow \wp(Val)$ maps variables and type variable to the sets of values that they may be bound to.

The judgement is defined by the axioms and rules in the upper part of Table 2. Basically, the rules amount to demanding that ϑ contains all the values associated with the components of a term, e.g. a name n evaluates to the set ϑ, provided that n belongs to ϑ; similarly for a variable x, provided that ϑ includes the set of values $\rho(x)$ to which x is associated with.

Analysis of Processes

In the analysis of processes, the information on the possible values, that may flow over the network, is collected into the component κ:

- $\kappa \subseteq \wp(Val^*)$: the *abstract network environment* 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 $\Gamma : \mathcal{X}_T \rightarrow \wp(\mathbf{Tag})$), while $\psi \subseteq \mathcal{C}$, is the (possibly empty) set of "error messages" of the form l, indicating that a type-mismatching (or violation) may happen at 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.

Before commenting on the analysis rules, we introduce three auxiliary functions, all of which generate some logic formulas to be used in the analysis rules. See some examples below.

The first one is the *matching* function, which takes care of pattern matching a value v to a matching term M. Remember that pattern matching cannot be performed on either $\natural x$ or $\sharp t$, requiring that M has to be some S or T. If this is the case, matching succeeds when v is an evaluation of the value of S or T.

$$match(v, M, \rho) = \begin{cases} \text{false} & \text{if } M \in \{S, T\} \wedge v \notin \vartheta \text{ where } \vartheta \text{ is s.t. } (\rho \models M : \vartheta) \\ \text{true} & \text{otherwise} \end{cases}$$

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

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

The last function is about *type checking*. Given a type environment Γ, it checks whether v is the expected type of a *definition type 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 type checking work, M has to be a definition type.

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

$$
\begin{array}{llll}
\mathit{match}(m, n) & = (\rho \models n : \vartheta \ \wedge \ m \in \vartheta) & \mathit{match}(m, \natural x) & = \mathit{true} \\
\mathit{sub}(m, \natural x) & = (m \in \rho(x)) & \mathit{sub}(m, n) & = \mathit{true} \\
\mathit{chk}(m, \sharp t, \Gamma, l) & = (m \neq \Gamma(t) \Rightarrow l \in \psi) & \mathit{chk}(m, n, \Gamma, l) & = \mathit{true}
\end{array}
$$

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

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

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

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

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

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

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

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, since they 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* function and *substitution*. The idea is to examine all the sequences of $\langle v_1, ..., v_k \rangle$ in the κ component and to point-wise compare it against the tuple of matching terms $(M_1, ..., M_k)$. The *matching* function selects only the closed terms 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 type 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 matching 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 type variables. When processing the *substitution*, *type checking* is also performed to capture violations. These occur when a *definition type variable* is bound to an unexpected type. In this case, the label l of the decryption is recorded in the error component ψ. Both in the case of input and decryption we make sure only to analyse the continuation process P in those cases where the input or decryption could indeed succeed.

The rule for *type declaration* (TNew) requires that the declared type is recorded in the type 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: if $(\rho, \kappa, \Gamma) \models P : \psi$, then the same tuple $(\rho, \kappa, \psi, \Gamma)$ is a valid estimate for all the states passed through in a computation of P, i.e. for all the derivatives of P.

Lemma 3.1 (Substitution result in Terms) *$\rho \models E : \vartheta$ and $E' \in \rho(x)$ imply $\rho \models E[E'/x] : \vartheta$*

Proof. The proof proceeds by structural induction over term by regarding each of the rules in the analysis.

Case (Const). Assume that $E = N$ and $\rho \models N : \vartheta$. For arbitrary choices of x and E' it holds that $N[E'/x] = N$ so it is immediate that also $\rho \models N[E'/x] : \vartheta$.

Case (Var). Assume that $E = X$ and $\rho \models X : \vartheta$, i.e. that $\rho(X) \subseteq \vartheta$. Then there are two cases. Either $X \neq x$ in which case $X[E'/x] = X$ so clearly $\rho \models X[V/x] : \vartheta$. Alternatively, $X = x$ in which case $X[E'/x] = E'$. Furthermore assume that $E' \in \rho(x)$ and because $\rho(x) \subseteq \vartheta$, it holds that $\rho \models E' : \vartheta$ in which case $\rho \models X[E'/x] : \vartheta$ by the analysis.

Case (Encr). Follow directly from the induction hypothesis □

Lemma 3.2 (Substitution result in Processes) *$\rho, \kappa, \Gamma \models P : \psi$ and $E \in \rho(x)$ imply $\rho, \kappa, \Gamma \models P[E/x] : \psi$*

Proof. The proof is done by straightforward induction applying the induction hypothesis on any sub-process and lemma 3.1 on any sub-terms. □

Lemma 3.3 (Predicates Equivalence 1) *For any arbitrary* v *and* M, $match(v, M) \Rightarrow (v, M)$

Lemma 3.4 (Predicates Equivalence 2) *For any arbitrary* v,M,Γ *and* l, $chk(v, M) \Rightarrow \mathcal{R}(v, M)$

Lemma 3.5 (Invariance of Structural Congruence) *If* $P \equiv Q$ *and* $\rho, \kappa, \Gamma \models P : \psi$ *then* $\rho, \kappa, \Gamma \models Q : \psi$

Proof. The proof amounts to a straightforward inspection of each of the clauses defining $P \equiv Q$.
□

Theorem 3.6 (Subject reduction) *If* $\Gamma \vdash P \rightarrow Q$ *and* $\rho, \kappa, \Gamma \models P : \psi$ *then also* $\rho, \kappa, \Gamma \models Q : \psi$. *Furthermore, if* $\psi = \emptyset$ *then* $P \rightarrow_{\mathsf{RM}} Q$

Proof. By induction on the inference of $P \rightarrow Q$.
In case (Com) we assume
$\rho, \kappa, \Gamma \models \langle E_1, \ldots, E_k\rangle.P \mid (M_1, \ldots, M_k).Q : \psi$ which amounts to:

(a) $\wedge_{i=1}^{k} \rho \models E_i : \vartheta_i$

(b) $\forall v_1, \ldots, v_k : \wedge_{i=1}^{k} v_i \in \vartheta_i \Rightarrow \langle v_1, \ldots, v_k\rangle \in \kappa$

(c) $\rho, \kappa, \Gamma \models P : \psi$

(d) $\forall \langle v_1, \ldots, v_k\rangle \in \kappa : \wedge_{i=1}^{k} match(v_i, M_i) \Rightarrow \wedge_{i=1}^{k} sub(v_i, M_i) \;\wedge\; \rho, \kappa, \Gamma \models Q : \psi$

Moreover we assume that $\wedge_{i=1}^{k} comp(E_i, M_i)$ because $\langle E_1, \ldots, E_k\rangle.P \mid (M_1, \ldots, M_k).Q \rightarrow P \mid Q[E_1/M_1, \ldots, E_k/M_k]$ and we have to prove $\rho, \kappa, \Gamma \models P \mid Q[E_1/M_1, \ldots, E_k/M_k] : \psi$. From (a) we have $\wedge_{i=1}^{k} E_i \in \vartheta_i$ since $\wedge_{i=1}^{k} \mathsf{fv}(E_i) = \emptyset$ and then (b) gives $\langle E_1, \ldots, E_k\rangle \in \kappa$.

From the assumption $\wedge_{i=1}^{k} comp(E_i = M_i)$ we get $\wedge_{i=1}^{k} match(E_i, M_i)$. Now (d) gives $\wedge_{i=1}^{k} sub(E_i, M_i)$ and $\rho, \kappa, \Gamma \models Q : \psi$. The substitution result then gives $\rho, \kappa, \Gamma \models Q[E_1/M_1, \ldots, E_k/M_k]$ and together with (c) this gives the required result. The second part is trivial: when $\psi = \emptyset$, obviously
$\langle E_1, \ldots, E_k\rangle.P \mid (M_1, \ldots, M_k).Q \rightarrow_{\mathsf{RM}} P \mid Q[E_1/M_1, \ldots, E_k/M_k]$

In case (Dec) we assume
$\rho, \kappa, \Gamma \models decrypt\ \{E_1, \ldots, E_k\}_{E_0}\ as\ \{M_1, \ldots, M_k\}_{E'_0}^{l}\ in\ P : \psi$
which amounts to:

(f) $\wedge_{i=0}^{k}\ \rho \models E_i : \vartheta_i$

(g) $\forall v_0, \ldots, v_k : \wedge_{i=0}^{k} v_i \in \vartheta_i \Rightarrow \{v_1, \ldots, v_k\}_{v_0} \in \vartheta$

(h) $\rho \models E'_0 : \vartheta'_0$

(i) $\forall \{v_1, \ldots, v_k\}_{v_0} \in \vartheta : v_0 \in \vartheta_0$

$$\Rightarrow \wedge_{i=1}^{k} match(v_i, M_i) \Rightarrow (\wedge_{i=1}^{k}(sub(v_i, M_i) \wedge$$

$$chk(v_i, M_i, \Gamma, l) \;\wedge\; \rho, \kappa, \Gamma \models P : \psi)$$

Furthermore we assume that $\wedge_{i=0}^{k} comp(E_i, M_i)$ because $decrypt\ \{E_1, \ldots, E_k\}_{E_0}\ as\ \{M_1, \ldots, M_k\}^{l}_{E'_0}\ in\ P \rightarrow$ $P[E_1/M_1, \ldots, E_k/M_k]$ and we have to prove $\rho, \kappa, \Gamma \models P[E_1/M_1, \ldots, E_k/M_k] : \psi$. From (f) and $\wedge_{i=0}^{k}\mathsf{fv}(E_i) = \emptyset$, we get $\wedge_{i=0}^{k} E_i \in \vartheta_i$ and then (g) gives $\{E_1, \ldots, E_k\}_{E_0} \in \vartheta$. From (h) and the assumption $\wedge_{i=0}^{k} comp(E_i, M_i)$ we get $v_0 \in \vartheta_0$ and $\wedge_{i=1}^{k} match(v_i, M_i)$. Now (i) gives $\wedge_{i=1}^{k}(sub(v_i, M_i) \wedge chk(v_i, M_i, \Gamma, l)$ and $\rho, \kappa, \Gamma \models P : \psi$. Using Lemma 3.2 we get the required result $\rho, \kappa, \Gamma \models P[E_1/M_1, \ldots, E_k/M_k]$

For the second part of the result we observe that $\wedge_{i=1}^{k} chk(v_i, M_i, \Gamma, l)$ follows from (i) and since $\psi = \emptyset$ it must be the case that $\wedge_{i=1}^{k}$ if $M_i \in \{\sharp t | t \in \mathcal{X}_T\}$ then $v = \Gamma(\sharp t)$. Thus the condition of the rule (Dec) are fulfilled for $\rightarrow_{\mathsf{RM}}$.

In case (Tyep Decl) we assume $\rho, \kappa, \Gamma \models (\nu\ \sharp t : Tag)P : \psi$, which amounts to:

$(a)\quad (\sharp t, Tag) \in \Gamma$

$(b)\quad \rho, \kappa, \Gamma \models P : \psi$

Furthermore we assume that $\Gamma[\sharp t \mapsto Tag] \vdash P \rightarrow Q$. By applying the induction hypothesis on (b), we have $\rho, \kappa, \Gamma \models Q : \psi$, which together with (a) gives the expected result that $\rho, \kappa, \Gamma \models (\nu\ \sharp t : Tag)Q : \psi$.

In cases (Par) and (Rep) follow directly from the induction hypothesis.

The case (Congr) also uses the congruence result.

□

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.

Theorem 3.7 (Static check for reference monitor) *If* $\rho, \kappa, \Gamma \models P : \psi$ *and* $\psi = \emptyset$ *then RM cannot abort* P.

Proof. Suppose *per absurdum* that such Q and Q' exist. A straightforward induction extends the subject reduction result to $P \rightarrow^* Q$ giving $\rho, \kappa, \Gamma \models Q : \psi$ and $\psi = \emptyset$. The part 2 of the subject reduction result applied to $Q \rightarrow Q'$ gives $Q \rightarrow_{\mathsf{RM}} Q'$ which is a contradiction. □

Example

Consider a scenario in which a principal A sends out an encrypted nonce onto the network and another principal B is expecting an encrypted key receiving from the network. Assume both encryptions use the same key K, obviously, B could be cheated on accepting the nonce as the key.

$$A \rightarrow \quad : \{N\}_K$$

$$\rightarrow B : \{K'\}_K$$

Our control flow analysis can work in two ways depending on how the protocol is modelled: either detecting what B received is a wrong one or preventing B from accepting it.

• In case the goal is to ***detect*** any type flaw attack may happen to the protocol, we can model it as follows,

$$\begin{aligned}&\langle A, \{(N, nonce)\}_K\rangle.0\\ &\mid (\nu\ \sharp tx_n : key)\ \ (A, \natural x_{enc}).\ \ decrypt\ x_{enc}\ as\ \{(\natural x_n, \sharp tx_n)\}^l_K\ in\ 0\end{aligned}$$

where the type of the encrypted message that B received, i.e. $\sharp tx_n$, is declared to be *key*. The analysis then gives rise to the analysis components ρ, κ, Γ and ψ with the following entries:

$$\begin{array}{lll}\langle A, \{(N, nonce)\}_K\rangle \in \kappa & (\sharp tx_n, key) \in \Gamma & l \in \psi\\ \{(N, nonce)\}_K \in \rho(x_{enc}) & N \in \rho(x_n) & nonce \in \rho(tx_n)\end{array}$$

which show that the attack is captured by $l \in \psi$

• In case one wants to ***prevent*** such a type flaw attack from happening, the protocol can be modelled as,

$$\begin{aligned}&\langle A, \{(N, nonce)\}_K\rangle.0\\ &\mid (A, \natural x_{enc}).\ \ decrypt\ x_{enc}\ as\ \{(\natural x_n, key)\}^l_K\ in\ 0\end{aligned}$$

It requires that the message inside the encryption that B got has to be a key. In this case, the analysis result becomes:

$$\begin{array}{lll}\langle A, \{(N, nonce)\}_K\rangle \in \kappa & \Gamma = \emptyset & \psi = \emptyset\\ \{(N, nonce)\}_K \in \rho(x_{enc}) & \rho(x_n) = \emptyset & \end{array}$$

Now $\rho(x_n) = \emptyset$ shows that no value binds to the variable x_n, i.e. the type flaw attack is successfully prevented.

Modelling the Attacker

In our work, the protocol and the attacker are formally modelled as two parallel processes, $P_{sys} \mid P_\bullet$, where P_{sys} represents the protocol process and $P_\bullet$ is some arbitrary attacker. The attacker considered here is the Dolev-Yao attacker [7], who is an active attacker and assumed to have the overall control of the network, over which principals exchange messages. Therefore he has access to messages transmitted over the network and is able to eavesdrop or replay messages sending over the network but also to encrypt, decrypt or generate messages provided that the necessary information is within his knowledge. Instead, secret messages and keys, e.g. (νK_{AB}), are restricted to their scope in P_{sys} and thus not immediately accessible to the attacker. To deal with types, we require that the attacker is able to change types of terms that are accessible to him. Due to space limitations, we shall not go further into details here, rather we refer to [3] for a description about modelling the attacker in a similar setup, as well as for a similar treatment of semantic correctness.

4 Validation

To verify the usefulness of our Control Flow Analysis, a number of experiments have been performed on security protocols from the literature. In this section, we shall show the analysis results of some example protocols, which are subject to type flaw attacks, namely the Woo and Lam protocol, version π_1 and the Andrew Secure RPC protocol (both the original version and the BAN version with type flaw corrected). The analysis results show that those type flaw attacks are successfully captured. Furthermore, it proves that after BAN's correction, the Andrew Secure RPC protocol does not suffer from type flaw attacks any longer.

Woo and Lam Protocol π_1

Woo and Lam [19] introduced a protocol that ensures one-way authentication of the initiator of the protocol, A, to a responder, B. The protocol uses symmetric-key cryptography and a trusted third-party server, S, with whom A and B share long-term symmetric keys. The protocol uses a fresh nonce N_B produced by B. The protocol narration is listed in the left part of the figure below, where K_{AS} and K_{BS} represent the long-term keys that A and B share with the trusted server S.

1. $A \rightarrow B : A$	1. $M(A) \rightarrow B : A$
2. $B \rightarrow A : N_B$	2. $B \rightarrow M(A) : N_B$
3. $A \rightarrow B : \{A, B, N_B\}_{K_{AS}}$	3. $M(A) \rightarrow B : N_B$
4. $B \rightarrow S : \{A, B, \{A, B, N_B\}_{K_{AS}}\}_{K_{BS}}$	4. $B \rightarrow M(S) : \{A, B, N_B\}_{K_{BS}}$
5. $S \rightarrow B : \{A, B, N_B\}_{K_{BS}}$	5. $M(S) \rightarrow B : \{A, B, N_B\}_{K_{BS}}$
the protocol narration	*the type flaw attack*

The Woo-Lam protocol is prown to a type flaw attack, which is shown in the right part of the figure. The attacker replays the nonce N_B to B in step 3, which B accepts as being of the form $\{A, B, N_B\}_{K_{AS}}$. B then encrypts whatever he received and then sends it out in step 4. The attacker intercepts it and replays it to B in step 5 and therefore fools B to believe that he has authenticated A, whereas A has not even participated in the run.

In LySa, the Woo-Lam protocol is modelled as three processes, A, B and S, running in parallel within the scope of the shared keys, say $P_{WL} = (\nu\ K_{AS})(\nu\ K_{BS})(A \mid B \mid S)$, each of which represents the sequence of actions of one principal as listed below. For clarity, each message begins with the pair of principals involved in the exchange.

$$\begin{array}{lll}
\text{Principal } A : & & (\nu\ \sharp tx_{nb} : nonce) \\
& /*1*/ & \langle A, B, A \rangle. \\
& /*2*/ & (B, A, (\natural x_{nb}, \sharp tx_{nb})). \\
& /*3*/ & \langle A, B, (\{A, B, (x_{nb}, tx_{nb})\}_{K_{AS}}, \{agent, agent, nonce\}_{key}) \rangle.0
\end{array}$$

$$
\begin{array}{lll}
\text{Principal } B: & /*1*/ & (A, B, A). \\
 & /*2*/ & (\nu\ N_B)\ \ \langle B, A, (N_B, \mathit{nonce})\rangle. \\
 & /*3*/ & (A, B, (\natural ya_{enc}, \sharp tya_{enc})). \\
 & /*4*/ & \langle B, S, \{A, B, (ya_{enc}, tya_{enc})\}_{K_{BS}}\rangle. \\
 & /*5*/ & (S, B, (\natural ys_{enc}, \sharp tys_{enc})). \\
 & & \mathit{decrypt}\ ys_{enc}\ \mathit{as}\ \{A, B, (N_B, \mathit{nonce})\}^{l_1}_{K_{BS}}\ \mathit{in}\ 0 \\
\text{Server } S: & & (\nu\ \sharp tza_{enc} : \mathit{enc})\ (\nu\ \sharp tz_{nb} : \mathit{nonce}) \\
 & /*4*/ & (B, S, (\natural zy_{enc}, \sharp tzy_{enc})). \\
 & & \mathit{decrypt}\ zy_{enc}\ \mathit{as}\ \{A, B, (\natural za_{enc}, \sharp tza_{enc})\}^{l_2}_{K_{BS}}\ \mathit{in} \\
 & & \mathit{decrypt}\ za_{enc}\ \mathit{as}\ \{A, B, (\natural z_{nb}, \sharp tz_{nb})\}^{l_3}_{K_{AS}}\ \mathit{in} \\
 & /*5*/ & \langle S, B, \{A, B, (z_{nb}, tz_{nb})\}_{K_{BS}}\rangle.0
\end{array}
$$

For the Woo and Lam protocol, we have $(\rho, \kappa, \Gamma) \models P_{WL} : \psi$, where ρ, κ and Γ have the following non-empty entries (we only list here the interesting ones):

$$\rho(za_{enc}) = \{\{A, B, (N_B, \mathit{nonce})\}_{K_{AS}}, N_B\} \qquad (\sharp tza_{enc}, \mathit{enc}) \in \Gamma$$

$$\rho(tza_{end}) = \{\{\mathit{agent}, \mathit{agent}, \mathit{nonce}\}_{\mathit{key}}, \mathit{nonce}\}\ \ \{l_2\} \in \psi$$

The error component has a non-empty set, $\psi = \{l_2\}$, showing that a violation may happen in the decryption marked with label l_2 (the second line of step 4 in S). This is the place where S is trying to decrypt and bind values to the variable za_{enc} and its type variable tza_{enc}, which, as indicated by Γ, can only be $\{A, B, (N_B, \mathit{nonce})\}_{K_{AS}}$. However, $\rho(za_{enc})$ and $\rho(tza_{enc})$ suggest that za_{enc} may also have the value N_B and tza_{enc} may have the value *nonce*. This violates the type assertion and amounts to the fact that, in step 4, S receives the message $\{A, B, N_B\}_{K_{BS}}$ instead of the expected one $\{A, B, \{A, B, N_B\}_{K_{AS}}\}_{K_{BS}}$. This exactly corresponds to the type flaw shown before.

Andrew Secure RPC protocol

The goal of the Andrew Secure RPC protocol is to exchange a fresh, authenticated, secret key between two principals sharing a symmetric key K. In the first message, the initiator A sends a nonce N_A, the responder B increments and returns it as the second message together with his nonce N_B. A accepts the value and returns the $N_B + 1$, B receives and checks the third message and if it contains the nonce incremented, then he sends a new session key, K' to A together with a new value N'_B to be used in subsequent communications.

1. $A \rightarrow B : A, \{N_A\}_K$	1. $A \rightarrow B :\quad A, \{N_A\}_K$
2. $B \rightarrow A : \{N_A + 1, N_B\}_K$	2. $B \rightarrow A :\quad \{N_A + 1, N_B\}_K$
3. $A \rightarrow B : \{N_B + 1\}_K$	3. $A \rightarrow M(B) : \{N_B + 1\}_K$
4. $B \rightarrow A : \{K', N'_B\}_K$	4. $M(B) \rightarrow A : \{N_A + 1, N_B\}_K$
the protocol narration	*the type flaw attack*

Also, the Andrew Secure RPC protocol [17] is subject to type flaw attack as shown above in the right part of the figure: by replaying the message from step 2 to B in step 4, the attacker can successfully force A to accept $N_A + 1$ as the new session key. The protocol makes use of an operation to increment N_A, in step 2, and N_B, in the third step (see [3] for the possible model of SUCC)).

The protocol can be modelled as $P_{Andrew} = (\nu\ K)(A \mid B)$, where K is the shared key and A and B are defined as follows (we only list the relevant steps).

$$\begin{array}{lrl}
\text{Principal } A: & & (\nu\ N_A)\ \ (\nu\ \sharp tx_k : key)\ \ (\nu\ \sharp tx_{nb'} : nonce) \\
 & /*1*/ & \langle A, B, A, \{(N_A, nonce)\}_K\rangle.\ldots \\
 & /*4*/ & (B, A, \natural x_{enc}). \\
 & & decrypt\ x_{enc}\ as\ \{(\natural x_k, \sharp tx_k), (\natural x_{nb'}, \sharp tx_{nb'})\}_K^{l_{x1}}\ in\ 0 \\
\text{Principal } B: & & (\nu\ N_B)(\nu\ N'_B)(\nu\ K')(\nu\ \sharp ty_{na} : nonce) \\
 & /*1*/ & (A, B, A, \natural y_{enc}). \\
 & & decrypt\ y_{enc}\ as\ \{(\natural y_{na}, \sharp ty_{na})\}_K^{l_{y1}}\ in \\
 & /*2*/ & \langle B, A, \{(y_{na} + 1, ty_{na}), (N_B, nonce)\}_K\rangle.\ldots
\end{array}$$

For the Andrew Secure RPC protocol, we have $(\rho, \kappa, \Gamma) \models P_{Andrew} : \psi$, where ρ, κ and Γ have the following non-empty entries (we only list here the interesting ones):

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle B, A, \{(N_A + 1, nonce), (N_B, nonce)\}_K\rangle \in \kappa} \\
\rho(x_k) = \{K', N_a + 1\} & \rho(tx_k) = \{key, nonce\} \\
(\sharp tx_k, key) \in \Gamma & (l_{x1}) \in \psi
\end{array}$$

The component κ collects all the messages potentially flowing over the network, including the one sent by B in step 2, namely $\langle B, A, \{(N_A+1, nonce), (N_B, nonce)\}_K\rangle$. This message could be received by A in his fourth step (e.g. replayed by an attacker) and consequently binding $N_A + 1$ to $\natural x_k$ and *nonce* to $\sharp tx_k$, which can be verified by examining the content of ρ (i.e. $N_A + 1 \in \rho(x_k)$ and $nonce \in \rho(tx_k)$). However, as suggested by Γ, the expected type of the type variable tx_k can *only* be *key* (by $\Gamma(tx_k) = \{key\}$) but not *nonce*. This violation is captured by the analysis by recording the label l_{x1} in the error component ψ (by $l_{x1} \in \psi$).

Andrew Secure RPC protocol with type flaw corrected

An improved version of Andrew Secure RPC protocol is suggested in [6] in order to prevent the above mentioned type flaw attack. The fixing amounts to inserting another component N_A into the encryption in the fourth message, as shown below,

$$4'.\ \ B \to A :\ \ \{K', N'_B, N_A\}_K$$

Now the encryption in step 2 has two fields and in step $4'$, A is expecting an encryption of 3 fields, therefore the attacker is no longer able to replay the message from step 2 and consequently make A accept nonce as a fresh key. This claim is verified by applying our analysis, which gives an empty error component, i.e. $\psi = \emptyset$.

5 Conclusion and Related Work

A type flaw attack happens when a field in a message is interpreted as having a type other than the originally intended one. In this paper, we extended the syntax of the process calculus, LySa, with tags, which represent the intended types of terms. The semantics of the tagged LySa makes use of a *reference monitor* to capture type-mismatching at run time.

On the static side, we developed a control flow analysis for the tagged LySa processes to check at each decryption place that whether the received, secret data has the right type. The static analysis ensures that, if each component of an encryption received by a principal is of the intended type, then the process is not subject to a type flaw attack at execution time. Actually, for malleability reasons, we only consider type flaws attacks occurring inside encryptions and decryptions. As far as the attacker is concerned, we adopted the notion from Dolev-Yao threat model and extended it with tags in order to fit it into our setting. The control flow analysis has been applied to a number of protocols, e.g. Woo-Lam π_1 and Andrew Secure RPC as shown in Section 4, and has confirmed that we can successfully detect type flaw attacks on the protocols.

Type flaw attacks on security protocols have been studied for some years, e.g. [11] also adopted the technique of tagging each message field with intended type, and later on, [12] simplified the tag structure for encryption. However these works aim at preventing type flaw attacks in the protocol execution stage by attaching some extra bits, representing types, to the messages transmitted over the network, and consequently the size of each message is increased, which results in raising unnecessary burden to the underlying network. Other works on type flaw attacks include applying type and effect system to security protocols, e.g. [10], such that a protocol is free of type flaw attacks if it is type checked. Type Systems are normally prescriptive(i.e. they infer types and impose the well-formedness conditions at the same time), while Control Flow Analysis is normally descriptive (i.e. it merely infers the information and then leave it to a separate step to actually impose demands on when programs are well-formed). Our approach offers a mix of both ways. Indeed, it can be either *descriptive*, i.e. it describes when the protocol does not respect the typing (via binding of type variables) or *prescriptive*, i.e. some flaws are avoided (via matching of tag terms). Under this regard, launching the tool implementing our analysis can then correspond to a sort of approximate type checking. More specifically, our control flow analysis can be used to 1) detect type flaw attacks: it can be applied in the protocol design stage: once a tagged protocol process is analyzed to be free of type flaw attacks, it can be used untagged while still ensures security; or 2) prevent type flaw attacks: the tags work in a way such that fields with different types cannot be mixed up. Therefore, it offers flexibility in satisfying different needs.

LySa has been developed to be decorated by several kinds of annotations and successfully applied for checking different security properties, e.g. confidentiality [9] and freshness [8]. It is very easy to combine tags with those techniques, thus obtaining a more general form of analysis. The core analysis can remain the same: different inspections of a solution permit to check different security properties of a protocol, with no need of re-analysing it several times.

The control flow analysis presented here is designed to capture *simple* type flaw attacks, i.e. one field is confused with another single field. Future work will extends the analysis to deal with more *complex* ones [18], as considered in [13], e.g. when a single field in a message is confused with a concatenation of fields. Furthermore, we can think about more complex kinds of tags.

Acknowledgments. We are grateful to Hanne Riis Nielson and Terkel K. Tolstrup for their helpful discussions and comments.

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] C. Bodei, M. Buchholtz, P. Degano, F. Nielson and H.R. Nielson. Automatic Valication of Protocol Narration. *In Proc. of CSFW'03*, IEEE Press.

[3] 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.

[4] C. Bodei, P. Degano, H. Gao, L. Brodo. Detecting and Preventing Type flaws: a Control Flow Analysis with tags. TR-07-16, Dipartimento di Informatica, Università di Pisa, 2007.

[5] M. Buchholtz, F. Nielson and H.R. Nielson. A Calculus for Control Flow Analysis of Security Protocols. *International Journal of Information Security*, 2(3-4), pp.145-167, 2004.

[6] M. Burrows and M. Abadi and R. Needham. A Logic of Authentication. ACM. *Transactions in Computer Systems*, 8(1), pp. 18-36, 1990.

[7] D. Dolev and A.C. Yao. On the Security of Public Key Protocols. IEEE TIT, IT-29(12):198-208, 1983.

[8] H. Gao, P. Degano, C. Bodei and H.R. Nielson. Detecting Replay Attacks by Freshness Annotations. *In Proc. of International Workshop on Issues in the Theory of Security (WITS 2007).*

[9] H. Gao and H.R. Nielson. Analysis of LySa-calculus with explicit confidentiality annotations. *In Proc. of Advanced Information Networking and Applications (AINA 2006)*, IEEE Computer Society.

[10] A.D. Gordon and A. Jeffrey. Types and Effects for Asymmetric Cryptographic Protocols. *In Proc. of 15th Computer Security Foundations Workshop*, pp. 77-91, IEEE Computer Society, 2002.

[11] J. Heather, G. Lowe and S. Schneider. How to prevent type flaw attacks on security protocols. *In Proc. of the 13th Computer Security Foundations Workshop*, IEEE Computer Society Press, 2000.

[12] Y. Li, W. Yang and J. Huang. *Journal of Information Science and Engineering*, 21:59-84, 2005.

[13] C. Meadows. Identifying potential type confusion in authenticated messages. *In. Proc. of Workshop on Foundation of Computer Security*, pp. 75-84, 2002.

[14] R. Milner. Communicating and mobile systems: the π-calculus. Cambridge University Press, 1999.

[15] 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.

[16] C.R. Nielsen, F. Nielson, H.R. Nielson. Cryptographic Pattern Matching. ENTCS 168, pp. 91-107, 2007.

[17] M. Satyanarayanan. *Integrating security in a large distributed system.* ACM Transactions on Computer Systems, 7(3):247–280, 1989.

[18] E. Snekkenes. Roles in cryptographic protocols. *In Proc. of the 1992 IEEE Computer Security Symposium on Research in Security and Privacy*, pp.105-119. IEEE Computer Society Press, 1992.

[19] T.Y.C. Woo and S.S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24-37, 1994.