Electronic Notes in Theoretical Computer Science 194 (2007) 3–22 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Detecting and Preventing Type flaws: a Control Flow Analysis with Tags [1](#_bookmark0)

# Chiara Bodei1 Pierpaolo Degano1 Han Gao2 Linda Brodo3

1 *Dipartimento di Informatica, Universita` 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](mailto:hg@imm.dtu.dk)

3 *Dipartimento di Scienze dei Linguaggi, Universita` di Sassari, via Tempio,9, I-07100 Sassari - Italia -*

[brodo@uniss.it](mailto: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

# 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](#_bookmark14),[12](#_bookmark15)] consist in systematically associating each message field with a tag representing its intended type. Therefore fields with dif- ferent 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

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

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

doi:10.1016/j.entcs.2007.09.010

presence of a Dolev-Yao attacker [[7](#_bookmark10)]. The proposed approach abstracts the fields of protocol messages to a lower level, such that the misinterpretation can be formally modelled. To this end, we extend the LySa calculus [[2](#_bookmark6),[3](#_bookmark7)] 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](#_bookmark22)] and Andrew Secure RPC [[17](#_bookmark16)].

LYSA has been given different kinds of annotations for checking other security properties, e.g. confidentiality [[9](#_bookmark12)] and freshness [[8](#_bookmark11)]. 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 intro- duce 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.

# Calculus

The LySa calculus [[2](#_bookmark6),[3](#_bookmark7)] is a process algebra, in the tradition of the *π*- [[14](#_bookmark17)] and Spi- [[1](#_bookmark5)] 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](#_bookmark6),[3](#_bookmark7)]. See also [[5](#_bookmark8),[16](#_bookmark18)] 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 modelling outputs and encryptions. Instead, for modelling inputs and de- cryptions we use matching terms, that, in turn, can be standard terms, or variables. We distinguish between *deﬁnition* (or binding) occurrences and *use* (or applied) oc- currences 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.

The distinction is obtained by means of syntax: the definition occurrence of a variable *x* is denoted by *qx*, 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*

*n* name (*n* ∈ N )

*x* use variable (*x* ∈ XS )

S ::= *matching standard terms*

*S* standard terms

*qx* definition variable (*x* ∈ X*S*)

Here N , X*S*, denote sets of names and of applied occurrences of variables, respec- tively. 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](#_bookmark14)], 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. 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. Actually, we can tag whatever we want, but we only check inside encryptions and decryptions, as shall be shown in Section 3.

Tag e *Tag* ::= *agent* | *nonce* | *key* | *...*

There are type variables, that are to standard variables such as tags are to closed terms (i.e. terms without variables). Similarly to the *q*-notation, we syntactically distinguish the defining occurrences of type variables (in the form *t*), from the corresponding use occurrences (in the form *t*). Syntactically, we have the following two new categories, where X*T* denote sets of applied occurrences of type variables.

*T* ::= *type terms*

*Tag* type tags (*Tag* ∈ Tag)

*t* use type variable (*t* ∈ XT )

T ::= *matching type terms*

*T* type terms

*t* defining type variable (*t* ∈ XT )

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*,* ··· *, Ek* encrypted under a term *E*0 representing a shared key.

*E* ::= *terms*

1. standard terms
2. type terms

*M* ::= *matching terms*

1. matching standard terms
2. matching type terms

{*E*1*,* ·· · *, Ek*}*E*0 symmetric encryption {*M*1*,. .., Mk*}*E* matching encryption

We call *V al* 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 both 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 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*

|  |  |  |
| --- | --- | --- |
| ⟨*E*1*,... , Ek*⟩*.P* |  | output |
| (*M*1*,... , Mk*)*.P*  *decrypt E as* {*M*1*,... , Mk*}*l* | *in P* | input  decryption with matching |
| (*ν n*)*P* |  | restriction |
| (*ν t* : *Tag*)*P* |  | type declaration |
| *P*1 | *P*2 |  | parallel composition |
| !*P* |  | replication |
| 0 |  | nil |

*E*0

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

Our patterns – in the form (*M*1*,* ··· *, Mk*) – are matched against tuples of terms (*E*1*,* ··· *, Ek*). Note that, at run time, each (*E*1*,* ··· *, Ek*) only includes closed terms,

i.e. each variable composing each one of the *Ei* has been bound in the previous computations. Instead, matching terms *Mi* can be partitioned in closed terms and variables to be bound. Intuitively, the matching succeeds when the closed terms, say *Mi*, pairwise match to the corresponding terms *Ei*, and its effect is to bind the remaining terms *Ej* to the remaining variables *qxj*. To exemplify, consider the following two processes, where only standard terms are present.

*P* = *decrypt* {*A, wn*}*K as Q* = *decrypt* {*A, NB*}*K as*

{*qxa, NB*}*lP in P* ' {*qxa, qyn*}*lQ in Q*'

*K*

*K*

The decryption in *P* succeeds only if *wn* = *NB*: in this case *qxa* will be bound to

*A*. Instead, the second decryption in *Q* always succeeds, and results in binding *qxa*

to *A*, and *qyn* to *NB*.

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:

*R* = (*ν tk* : *key*)*decrypt* {(*A, agent*)*,* (*NB , nonce*)*,* (*z, key*)}*K as*

{(*A, agent*)*,* (*NB , nonce*)*,* (*qzk, tk*)}*lR in R*' *R*˜ = (*ν tk* : *key*)*decrypt* {(*A, agent*)*,* (*NB , nonce*)*,* (*z, nonce*)}*K as*

*K*

{(*A, agent*)*,* (*NB , nonce*)*,* (*qzk, tk*)}*lR in R*˜'

*K*

*S* = *decrypt* {(*A, agent*)*,* (*NB , nonce*)*,* (*z, t*)}*K as*

{(*A, agent*)*,* (*NB , nonce*)*,* (*qzk, key*)}*lS*

*K*

*in S*'

The decryptions in *R* and *R*˜ always succeed and result in binding *qzk* to (the values

assumed by) *z*, and *tk* to *key* or to *nonce*. In particular, in *R*˜

the decryption

succeeds, even though the declared type for *tk* is *key*. In the decryption in *S* only if *t* successfully matches with *key* then *qzk* is bound to *z*.

Operational Semantics

Below we slightly modify the standard *structural congruence* ≡ on LySa pro- cesses, also to take care of type declarations. It is the least congruence satisfying the following clauses:

* *P* ≡ *Q* if *P* and *Q* are disciplined *α*-equivalent (as explained below);
* (P*/*≡*,* |*,* 0) is a commutative monoid;
* (*νn*)0 ≡ 0, (*νn*)(*νn*')*P* ≡ (*νn*')(*νn*)*P* , (*νn*)(*P* | *Q*) ≡ *P* | (*νn*)*Q* if *n* /∈ fn(*P* ),

(*ν t* : *Tag*)0 ≡ 0, (*ν t* : *Tag*)(*ν t*' : *Tag*)*P* ≡ (*ν t*' : *Tag*)(*ν t* : *Tag*)*P* , (*ν t* : *Tag*)(*P* | *Q*) ≡ *P* | (*ν t* : *Tag*)*Q* if *t* /∈ bv(*P* );

* !*P* ≡ *P* | !*P*

To simplify the definition of our control flow analysis in Section 3, we disci- pline 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 equiva- lence 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 re- striction. The canonical name [*n*♩ is for a name *n*; similarly [*x*♩ is for a variable

*x*. For example, a process, that may generate infinitely many names, is !(*ν n*)*P* , as shown in the following chain of equivalences: !(*ν n*)*P* ≡ (*ν n*')*P* ' | !(*ν n*)*P* ≡

(*ν n*')*P* ' | (*ν n*'')*P* '' | !(*ν n*)*P* ≡ *...* Furthermore, the names *n*, *n*' and *n*'' are generated by the same restriction and hence have the same canonical name, i.e. [*n*♩ = [*n*'♩ = [*n*''♩. In this way, we statically maintain the identity of values and variables that may be lost by freely applying *α*−conversions. Hereafter, when un- ambiguous, we shall simply write *n* (resp. *x*) for [*n*♩ (resp. [*x*♩).

Following the tradition of the *π*-calculus, we shall give LySa a reduction se- mantics. The *reduction relation* →R is the least relation on closed processes that satisfies the rules in Table [1](#_bookmark1). It uses structural congruence, as defined above, and the disciplined treatment of *α*-conversion. We consider two variants of *reduction relation* →R, graphically identified by a different instantiation of the relation R, which decorates the transition relation. Both semantics use the type environment Γ, which maps a type variable in a set of tags.

Γ: XT → *℘*(Tag)

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

* the *reference monitor semantics* Γ ▶ *P* →RM *Q* takes

R(*E, M* )= ⎧⎨ false if *M* = *t* ∧ *E* /∈ Γ( *t*)

⎩ true otherwise

This function affects only type variables, i.e. only matching terms *M* in the form

*t*. It checks whether the type (Γ( *t*)) associated with the variable includes *E*.

* the *standard semantics* Γ ▶ *P* → *Q* takes, by construction, R to be universally true (and therefore the index 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 *qx* or *t*.

*P* [*E/M* ]= ⎧⎨ *P* [*M* '→ *E*] if *M* ∈ {*qx*| *x* ∈ X*S*}∪ { *t*| *t* ∈ X*T* }

⎩ *P* otherwise

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* )= ⎧⎨ false if *E* /= *M* ∧ (fv(*M* ) ∪ bv(*M* )= ∅)

⎩ true otherwise

The judgement Γ ▶ *P* →R *P* ' means that the process *P* can evolve into *P* ', given the type environment Γ.

(*Com*)

∧*k* 1 *comp*(*Ei, Mi*)

*i*=

Γ ▶ ⟨*E*1*,.. ., Ek*⟩*.P* | (*M*1*, ... , Mk*)*.Q* →R *P* | *Q*[*E*1*/M*1*, ... , Ek/Mk*]

(*Dec*)

∧*k*

*i*=0

*comp*(*Ei, Mi*) ∧ ∧*k* R(*Ei, Mi*)

*i*=1

Γ ▶ *decrypt* {*E , ... , E* }

1 *k E*0

*as* {*M ,. .. , M* } *in P* →R *P* [*E*1*/M*1*,.. ., Ek/Mk*]

1

*l*

*k*

*E*0

(*Type Decl*) (*Res*)

Γ[ *t* '→ *T ag*] ▶ *P* →R *P* ' Γ ▶ *P* →R *P* '

Γ ▶ (*ν t* : *T ag*)*P* →R (*ν t* : *T ag*)*P* ' Γ ▶ (*ν n*)*P* →R (*ν n*)*P* '

(*P ar*)

Γ ▶ *P*1 →R *P* '

1

Γ ▶ *P*1 | *P*2 →R *P* ' | *P*2

1

(*Congr*)

*P* ≡ *P* ' ∧ Γ ▶ *P* ' →R *P* '' ∧ *P* '' ≡ *P* '''

Γ ▶ *P* →R *P* '''

Table 1 '

Operational semantics, Γ ▶ *P* →R *P* , parameterised on R.

The rule (Com) expresses that an output ⟨*E*1*,... , Ek*⟩*.P* is matched by an input (*M*1*,... , Mk*) by checking whether the closed terms *Mi* are pairwise the same with the corresponding *Ei* (i.e. if *comp*(*Ei, Mi*)). When the matchings are successful, the remaining *Ej* are bound to the corresponding *Mj* (that are variables or type variables).

Similarly, the rule (Decr) expresses the result of matching an encryption

{*E*1*,... , Ek*}*E*0 with decrypt *E* as{*M*1*,... , Mk*}*l* '

*E*0

in *P* . As it was the case for com-

munication, the closed terms *Mi* must match with the corresponding *Ei*, and ad- ditionally the keys must be the same. When the matching is successful the re- maining terms *Ej* are bound to the corresponding *Mi* (that are definition vari- ables or definition type variables). Recall that in the *reference monitor seman- tics* we ensure that the components of the decrypted message have the types ex- pected, by checking whether the *t* are bound to a type tag that is included in Γ( *t*). In the *standard semantics* the condition R(*E, M* ) is universally true and

thus can be ignored. Back to our example processes *R,*

*R*˜*, S*, we have that in *R*,

*comp*(*z, qzk*)= *comp*(*key, t*)= *true* and RM(*key, t*)= *true* (because *key* ∈ Γ( *t*)), while in *R*˜, *comp*(*z, qzk*) = *comp*(*nonce, t*) = *true*, but RM(*nonce, t*) = *f alse* (because *nonce* /∈ Γ( *t*)). Note also that in *S*, *comp*(*t, key*)= *true* only if *t* = *key*, and, in this case *P* [*z/qzk*]= *P* [*qzk* '→ *z*].

The rule (Type Decl) records the new association between the type variable *t* and the type *Tag* in the type environment Γ. The updating of Γ is indicated as Γ[ *t* '→ *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 Γ ▶

*P* →∗ *P* ' → *P* '', we always have Γ ▶ *P* →∗ *P* ' →RM *P* ''.

# 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 (Γ*, ρ, κ, ψ*) (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](#_bookmark2). 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. *qx*) and *deﬁnition type variable* (e.g.

*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](#_bookmark19)].

Analysis of Terms

The judgement for analysing terms is *ρ* |= *E* : *ϑ*. 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*:

* *ρ* : XS ∪ XT → *℘*(*V al*) 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](#_bookmark2). 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 *ρ*(*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*:

* *κ* ⊆ *℘*(*V al*∗): 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: *ρ, κ,* Γ |= *P* : *ψ*, where the com- ponents *ρ, κ*, and Γ are as above (recall that Γ : X*T* → *℘*(Tag)), while *ψ* ⊆ 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](#_bookmark2) (where *X* ⇒ *Y* means that *Y* is only evaluated when *X* is *True*) and are explained later.

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 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 *qx* or *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, ρ*)= ⎧⎨ false if *M* ∈ {*S, T* }∧ *v* /∈ *ϑ* where *ϑ* is s.t. (*ρ* |= *M* : *ϑ*)

⎩ true otherwise

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 *qx* or a type variable *t*.

*sub*(*v, M* )= ⎧⎨ false if *v* /∈ *ρ*(*M* ) with *M* ∈ {*qx*| *x* ∈ X*S*}∪ { *t*| *t* ∈ X*T* }

⎩ true otherwise

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 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,* Γ*, l, ψ*)= ⎧⎨ false if *M* ∈ { *t*| *t* ∈ X*T* } ∧ *v* /= Γ( *t*) ∧ *l* ∈ *ψ*

⎩ true otherwise

*match*(*n, m, ρ*) = (*ρ* |= *n* : *ϑ* ∧ *m* ∈ *ϑ*) *match*(*m, qx*) = *true sub*(*m, qx*) = (*m* ∈ *ρ*(*x*)) *sub*(*m, n*) = *true chk*(*m, t,* Γ*, l, ψ*) = (*m* /= Γ(*t*) ⇒ *l* ∈ *ψ*) *chk*(*m, n,* Γ*, l*) = *true*

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

⟨*v*1*, ..., vk*⟩ in the *κ* component and to point-wise compare it against the tuple of matching terms (*M*1*, ..., Mk*). The *matching* function selects only the closed terms

(*Dec*)

*i*=1

*ρ, κ,* Γ |= *P* : *ψ*

*ρ, κ,* Γ |= (*M*1*, ... , Mk*)*.P* : *ψ*

*ρ* |= *E* : *ϑ* ∧ *ρ* |= *E*0 : *ϑ*0 ∧

∀{*v*1*,.. ., vk*}*v*0 ∈ *ϑ* : *v*0 ∈ *ϑ*0 ⇒

(*match*(*vi, Mi*) ⇒ (*sub*(*vi, Mi*) ∧ *chk*(*vi, Mi,* Γ*, l*))) ∧

*ρ, κ,* Γ |= *P* : *ψ*

*i*=1

∧*k*

(*In*)

0 *k i*=0 *i i* 1 *k v*0

(*Encr*)

*ρ* |= {*E*1*, ... , Ek*}*E*0 : *ϑ*

*k*

∀*v ,.. .,v* : ∧ *v* ∈ *ϑ* ⇒ {*v ,. .., v* } ∈ *ϑ*

*ρ* |= *Ei* : *ϑi* ∧

=0

*i*

∧*k*

(*X* = *x* or *t*)

(*V ar*)

(*N* = *T ag* or *n*)

(*Const*)

(*Nil*) *ρ, κ,* Γ |=0 : *ψ*

*N* ∈ *ϑ*

*ρ* |= *N* : *ϑ*

(*Rep*)

*ρ*(*X*) ⊆ *ϑ*

*ρ* |= *X* : *ϑ*

(*Res*)

*ρ, κ,* Γ |= *decrypt E as* {*M*1*,.. ., Mk*}*l in P* : *ψ*

*k*

1

*i*

*k i*=1*vi*

*k*

1

*ρ, κ,* Γ |= *P* : *ψ*

(*Out*)

*ρ, κ,* Γ |= ⟨*E*1*,. .., Ek*⟩*.P* : *ψ*

∀⟨*v*1*,. .., vk*⟩ ∈ *κ* ∧*k* (*match*(*vi, Mi*) ⇒ *sub*(*vi, Mi*)) ∧

*ρ* |= *Ei* : *ϑi* ∧

=1

*i*

∧*k*

*E*0

( *t, T ag*) ∈ Γ ∧ *ρ, κ,* Γ |= *P* : *ψ ρ, κ,* Γ |= *P*1 : *ψ* ∧ *ρ, κ,* Γ |= *P*2 : *ψ*

(*TNew*) (*P ar*)

*ρ, κ,* Γ |= (*ν t* : *T ag*)*P* : *ψ ρ, κ,* Γ |= *P*1 | *P*2 : *ψ*

∀*v ,. .., v* : ∧ ∈ *ϑ* ⇒ ⟨*v ,. .., v* ⟩ ∈ *κ* ∧

Table 2

*ρ, κ,* Γ |= *P* : *ψ*

*ρ, κ,* Γ |= (*ν n*)*P* : *ψ*

*ρ, κ,* Γ |= *P* : *ψ*

*ρ, κ,* Γ |=!*P* : *ψ*

Analysis of tagged Lysa Terms: *ρ* |= *E* : *ϑ*, and Processes: *ρ, κ,* Γ |= *P* : *ψ*

and for each of them, say *Mi*, checks whether the corresponding *vi* is included in *ϑi*, i.e. the result of the analysis for *Mi*. If the matching succeeds for all the closed terms, then, the substitution function takes care of binding the remaining values *vj* to the corresponding *deﬁnition variables* or *deﬁnition type variables Mj*. 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 vari- ables or definition type variables. When processing the *substitution*, *type checking* is also performed to capture violations. These occur when a *deﬁnition type vari- able* 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 (*ρ, κ,* Γ) |= *P* : *ψ*, then the same tuple (*ρ, κ, ψ,* Γ) 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 for Terms) *ρ* |= *E* : *ϑ and E*' ∈ *ρ*(*x*) *imply ρ* |=

*E*[*E*'*/x*]: *ϑ*

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 *ρ* |= *N* : *ϑ*. For arbitrary choices of *x* and *E*' it holds that *N* [*E*'*/x*]= *N* so it is immediate that also *ρ* |= *N* [*E*'*/x*]: *ϑ*.

Case (Var). Assume that *E* = *X* and *ρ* |= *X* : *ϑ*, i.e. that *ρ*(*X*) ⊆ *ϑ*. Then there are two cases. Either *X* /= *x* in which case *X*[*E*'*/x*] = *X* so clearly *ρ* |= *X*[*V /x*] : *ϑ*. Alternatively, *X* = *x* in which case *X*[*E*'*/x*] = *E*'. Furthermore assume that *E*' ∈ *ρ*(*x*) and because *ρ*(*x*) ⊆ *ϑ*, it holds that *ρ* |= *E*' : *ϑ* in which case *ρ* |= *X*[*E*'*/x*]: *ϑ* by the analysis.

Case (Encr). Follow directly from the induction hypothesis

Lemma 3.2 (Substitution for Processes) *ρ, κ,* Γ |= *P* : *ψ and E* ∈ *ρ*(*x*) *imply*

*ρ, κ,* Γ |= *P* [*E/x*]: *ψ*

Proof. The proof is done by straightforward induction applying the induction hy- pothesis on any sub-process and lemma [3.1](#_bookmark3) on any sub-terms.

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

Lemma 3.4 (Predicates Equivalence 2) *For any arbitrary v,M,*Γ *and l, chk*(*v, M* ) ⇒ R(*v, M* )

Lemma 3.5 (Invariance of Structural Congruence) *If P* ≡ *Q and ρ, κ,* Γ |=

*P* : *ψ then ρ, κ,* Γ |= *Q* : *ψ*

Proof. The proof amounts to a straightforward inspection of each of the clauses defining *P* ≡ *Q*.

Theorem 3.6 (Subject reduction) *If* Γ ▶ *P* → *Q and ρ, κ,* Γ |= *P* : *ψ then also*

*ρ, κ,* Γ |= *Q* : *ψ. Furthermore, if ψ* = ∅ *then P* →RM *Q*

Proof. By induction on the inference of *P* → *Q*.

In case (Com) we assume

*ρ, κ,* Γ |= ⟨*E*1*,... , Ek*⟩*.P* | (*M*1*,... , Mk*)*.Q* : *ψ* which amounts to:

1. ∧*k ρ* |= *Ei* : *ϑi*

*i*=1

1. ∀*v*1*,... , vk* : ∧*k*

*i*=1

1. *ρ, κ,* Γ |= *P* : *ψ*

*vi* ∈ *ϑi* ⇒ ⟨*v*1*,... , vk*⟩∈ *κ*

1. ∀⟨*v*1*,... , vk*⟩∈ *κ* : ∧*k*

*i*=1

*match*(*vi, Mi*) ⇒ ∧*k*

*sub*(*vi, Mi*) ∧ *ρ, κ,* Γ |= *Q* : *ψ*

Moreover we assume that ∧*k comp*(*Ei, Mi*) because

*i*=1

*i*=1

⟨*E*1*,... , Ek*⟩*.P* | (*M*1*,... , Mk*)*.Q* → *P* | *Q*[*E*1*/M*1*,... , Ek/Mk*] and we have to prove

*ρ, κ,* Γ |= *P* | *Q*[*E*1*/M*1*,... , Ek/Mk*] : *ψ*. From (*a*) we have ∧*k*

*i*=1

*Ei* ∈ *ϑi* since

*k i*=1

∧

fv(*Ei*)= ∅ and then (*b*) gives ⟨*E*1*,... , Ek*⟩∈ *κ*.

¿From the assumption ∧*k*

*i*=1

*comp*(*Ei* = *Mi*) we get ∧*k*

*match*(*Ei, Mi*). Now

(*d*) gives ∧*k sub*(*Ei, Mi*) and *ρ, κ,* Γ |= *Q* : *ψ*. The substitution result then gives

*i*=1

*i*=1

*ρ, κ,* Γ |= *Q*[*E*1*/M*1*,... , Ek/Mk*] and together with (*c*) this gives the required result. The second part is trivial: when *ψ* = ∅, obviously

⟨*E*1*,... , Ek*⟩*.P* | (*M*1*,... , Mk*)*.Q* →RM *P* | *Q*[*E*1*/M*1*,... , Ek/Mk*]

In case (Dec) we assume

*ρ, κ,* Γ |= *decrypt* {*E*1*,... , Ek*}*E*0 *as* {*M*1*,... , Mk*}*l* '

*E*0

*in P* : *ψ*

which amounts to:

1. ∧*k ρ* |= *Ei* : *ϑi*

*i*=0

1. ∀*v*0*,... , vk* : ∧*k*

*i*=0

*vi* ∈ *ϑi* ⇒ {*v*1*,... , vk*}*v*0 ∈ *ϑ*

1. *ρ* |= *E*' : *ϑ*'

0 0

1. ∀{*v*1*,... , vk*}*v*0 ∈ *ϑ* : *v*0 ∈ *ϑ*0

*k i*=1

⇒ ∧

*match*(*vi, Mi*) ⇒ (∧*k*

(*sub*(*vi, Mi*)∧

*chk*(*vi, Mi,* Γ*, l*) ∧ *ρ, κ,* Γ |= *P* : *ψ*)

*i*=1

Furthermore we assume that ∧*k comp*(*Ei, Mi*) because

*i*=0

*decrypt* {*E*1*,... , Ek*}*E*0 *as* {*M*1*,... , Mk*}*l* '

*E*0

*in P* →

*P* [*E*1*/M*1*,... , Ek/Mk*] and we have to prove *ρ, κ,* Γ |= *P* [*E*1*/M*1*,... , Ek/Mk*] : *ψ*.

From (*f* ) and ∧*k*

*i*=0

fv(*Ei*)= ∅, we get ∧*k*

*Ei* ∈ *ϑi* and then (*g*) gives

{*E*1*,... , Ek*}*E*0 ∈ *ϑ*. From (*h*) and the assumption ∧*k*

*i*=0

*i*=0

*comp*(*Ei, Mi*) we get

*v*0 ∈ *ϑ*0 and ∧*k*

*i*=1

*match*(*vi, Mi*). Now (*i*) gives ∧*k*

(*sub*(*vi, Mi*) ∧ *chk*(*vi, Mi,* Γ*, l*)

and *ρ, κ,* Γ |= *P* : *ψ*. Using Lemma [3.2](#_bookmark4) we get the required result

*i*=1

*ρ, κ,* Γ |= *P* [*E*1*/M*1*,... , Ek/Mk*]

For the second part of the result we observe that

*i*=1

*k i*=1

∧

*chk*(*vi, Mi,* Γ*, l*) follows from (*i*) and since *ψ* = ∅ it must be the case that ∧*k*

if *Mi* ∈ { *t*|*t* ∈ X*T* } then *v* = Γ( *t*). Thus the condition of the rule (Dec) are

fulfilled for →RM.

In case (Tyep Decl) we assume

*ρ, κ,* Γ |= (*ν t* : *Tag*)*P* : *ψ*, which amounts to:

1. ( *t, T ag*) ∈ Γ
2. *ρ, κ,* Γ |= *P* : *ψ*

Furthermore we assume that Γ[ *t* '→ *Tag*] ▶ *P* → *Q*. By applying the induction hypothesis on (*b*), we have *ρ, κ,* Γ |= *Q* : *ψ*, which together with (*a*) gives the expected result that *ρ, κ,* Γ |= (*ν t* : *Tag*)*Q* : *ψ*.

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 ρ, κ,* Γ |= *P* : *ψ and*

*ψ* = ∅ *then RM cannot abort P.*

Proof. Suppose *per absurdum* that such *Q* and *Q*' exist. A straightforward in- duction extends the subject reduction result to *P* →∗ *Q* giving *ρ, κ,* Γ |= *Q* : *ψ* and *ψ* = ∅. The part 2 of the subject reduction result applied to *Q* → *Q*' gives *Q* →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* → : {*N* }*K*

→ *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,

⟨*A,* {(*N, nonce*)}*K* ⟩*.*0

| (*ν txn* : *key*) (*A, qxenc*)*. decrypt xenc as* {(*qxn, txn*)}*l*

*K*

*in* 0

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

⟨*A,* {(*N, nonce*)}*K* ⟩∈ *κ* ( *txn, key*) ∈ Γ *l* ∈ *ψ*

{(*N, nonce*)}*K* ∈ *ρ*(*xenc*) *N* ∈ *ρ*(*xn*) *nonce* ∈ *ρ*(*txn*) which show that the attack is captured by *l* ∈ *ψ*

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

⟨*A,* {(*N, nonce*)}*K* ⟩*.*0

| (*A, qxenc*)*. decrypt xenc as* {(*qxn, key*)}*l*

*K*

*in* 0

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

⟨*A,* {(*N, nonce*)}*K* ⟩∈ *κ* Γ= ∅ *ψ* = ∅

{(*N, nonce*)}*K* ∈ *ρ*(*xenc*) *ρ*(*xn*)= ∅

Now *ρ*(*xn*) = ∅ shows that no value binds to the variable *xn*, 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, *Psys* | *P*•, where *Psys* represents the protocol process and *P*• is some arbitrary attacker. The attacker considered here is the Dolev-Yao attacker [[7](#_bookmark10)], 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. (*νKAB*), are restricted to their scope in *Psys* 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. We refer to [[3](#_bookmark7)] for a description about modelling the attacker in a similar setup.

# 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](#_bookmark22)] introduced a protocol that ensures one-way authentication of the initiator of the protocol, *A*, toa 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 *NB* produced by *B*. The protocol narration is listed in the left part of the figure below, where *KAS* and *KBS* represent the long-term keys that *A* and *B* share with the trusted server *S*.

* 1. *A* → *B* : *A*
  2. *B* → *A* : *NB*
  3. *A* → *B* : {*A, B, NB*}*KAS*
  4. *B* → *S* : {*A, B,* {*A, B, NB*}*KAS* }*KBS*
  5. *S* → *B* : {*A, B, NB*}*KBS*

*the protocol narration*

1. *M* (*A*) → *B* : *A*
2. *B* → *M* (*A*): *NB*
3. *M* (*A*) → *B* : *NB*
4. *B* → *M* (*S*): {*A, B, NB*}*KBS*
5. *M* (*S*) → *B* : {*A, B, NB*}*KBS*

*the type flaw attack*

The Woo-Lam protocol is subject to a type flaw attack, which is shown in the right part of the figure. The attacker replays the nonce *NB* to *B* in step 3, which *B* accepts as being of the form {*A, B, NB*}*KAS* . *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 the parallel composition of three processes, *A*, *B* and *S*, running within the scope of the shared keys, say *PWL* = (*ν KAS*)(*ν KBS*)(*A* | *B* | *S*). Each process 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.

Principal *A* : (*ν txnb* : *nonce*)

*/* ∗ 1 ∗ */* ⟨*A, B, A*⟩*.*

*/* ∗ 2 ∗ */* (*B, A,* (*qxnb, txnb*))*.*

*/* ∗ 3 ∗ */* ⟨*A, B,* ({*A, B,* (*xnb, txnb*)}*KAS ,* {*agent, agent, nonce*}*key*)⟩*.*0

Principal *B* : */* ∗ 1 ∗ */* (*A, B, A*)*.*

*/* ∗ 2 ∗ */* (*ν NB*) ⟨*B, A,* (*NB, nonce*)⟩*.*

*/* ∗ 3 ∗ */* (*A, B,* (*qyaenc, tyaenc*))*.*

*/* ∗ 4 ∗ */* ⟨*B, S,* {*A, B,* (*yaenc, tyaenc*)}*KBS* ⟩*.*

*/* ∗ 5 ∗ */* (*S, B,* (*qysenc, tysenc*))*.*

*decrypt ysenc as* {*A, B,* (*NB, nonce*)}*l*1

*KBS*

*in* 0

Server *S* : (*ν tzaenc* : *enc*) (*ν tznb* : *nonce*)

*/* ∗ 4 ∗ */* (*B, S,* (*qzyenc, tzyenc*))*.*

*decrypt zyenc as* {*A, B,* (*qzaenc, tzaenc*)}*l*2 *in*

*KBS*

*decrypt zaenc as* {*A, B,* (*qznb, tznb*)}*l*3 *in*

*KAS*

*/* ∗ 5 ∗ */* ⟨*S, B,* {*A, B,* (*znb, tznb*)}*KBS* ⟩*.*0

For the Woo and Lam protocol, we have (*ρ, κ,* Γ) |= *PWL* : *ψ*, where *ρ*, *κ* and Γ have the following non-empty entries (we only list here the interesting ones):

{{*A, B,* (*NB , nonce*)}*KAS , NB*}⊆ *ρ*(*zaenc*) ( *tzaenc, enc*) ∈ Γ

{{*agent, agent, nonce*}*key, nonce*}⊆ *ρ*(*tzaenc*) {*l*2}∈ *ψ*

The error component has a non-empty set, *ψ* = {*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 *zaenc* and its type variable *tzaenc*, which, as indicated by Γ, can only be {*A, B,* (*NB, nonce*)}*KAS* . However, *ρ*(*zaenc*) and *ρ*(*tzaenc*) suggest that *zaenc* may also have the value *NB* and *tzaenc* 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, NB*}*KBS* instead of the expected one {*A, B,* {*A, B, NB*}*KAS* }*KBS* . 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, authenti- cated, secret key between two principals sharing a symmetric key *K*. In the first message, the initiator *A* sends a nonce *NA*, the responder *B* increments and returns it as the second message together with his nonce *NB*. *A* accepts the value and returns the *NB* + 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* → *B* : *A,* {*NA*}*K*
2. *B* → *A* : {*NA* + 1*, NB*}*K*
3. *A* → *B* : {*NB* + 1}*K*
4. *B* → *A* : {*K*'*,N* ' }*K*

*B*

*the protocol narration*

* 1. *A* → *B* : *A,* {*NA*}*K*
  2. *B* → *A* : {*NA* + 1*, NB*}*K*
  3. *A* → *M* (*B*): {*NB* + 1}*K*
  4. *M* (*B*) → *A* : {*NA* + 1*, NB*}*K*

*the type flaw attack*

Also, the Andrew Secure RPC protocol [[17](#_bookmark16)] 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 *NA* +1 as the new session key. The protocol makes use of an operation to increment *NA*, in step 2, and *NB*, in the third step (see [[3](#_bookmark7)] for the possible model of SUCC)).

The protocol can be modelled as *PAndrew* = (*ν K*)(*A* | *B*), where *K* is the shared key and *A* and *B* are defined as follows ( we only list the relevant steps).

Principal *A* : (*ν NA*) (*ν txk* : *key*) (*ν txnb*' : *nonce*)

*/* ∗ 1 ∗ */* ⟨*A, B, A,* {(*NA, nonce*)}*K* ⟩*....*

*/* ∗ 4 ∗ */* (*B, A, qxenc*)*.*

*decrypt xenc as* {(*qxk, txk*)*,* (*qxnb*' *, txnb*' )}*lx*1 *in* 0

*K*

Principal *B* : (*ν NB*)(*ν N* ' )(*ν K*')(*ν tyna* : *nonce*)

*B*

*/* ∗ 1 ∗ */* (*A, B, A, qyenc*)*.*

*decrypt y*

*enc*

*as* {(*qyna*

*, tyna*

)}*ly*1 *in*

*/* ∗ 2 ∗ */* ⟨*B, A,* {(*yna* + 1*, tyna*)*,* (*NB, nonce*)}*K* ⟩*....*

*K*

For the Andrew Secure RPC protocol, we have (*ρ, κ,* Γ) |= *PAndrew* : *ψ*, where *ρ*, *κ*

and Γ have the following non-empty entries (we only list here the interesting ones):

⟨*B, A,* {(*NA* + 1*, nonce*)*,* (*NB, nonce*)}*K* ⟩∈ *κ*

{*K*'*, Na* + 1}⊆ *ρ*(*xk*) {*key, nonce*}⊆ *ρ*(*txk*) ( *txk, key*) ∈ Γ (*lx*1) ∈ *ψ*

The component *κ* collects all the messages potentially flowing over the network, in- cluding the one sent by *B* in step 2, namely ⟨*B, A,* {(*NA*+1*, nonce*)*,* (*NB, nonce*)}*K* ⟩. This message could be received by *A* in his fourth step (e.g. replayed by an attacker) and consequently binding *NA* +1 to *qxk* and *nonce* to *txk*, which can be verified by examining the content of *ρ* (i.e. *NA* +1 ∈ *ρ*(*xk*) and *nonce* ∈ *ρ*(*txk*)). How- ever, as suggested by Γ, the expected type of the type variable *txk* can *only* be *key* (by Γ(*txk*) = {*key*}) but not *nonce*. This violation is captured by the analysis by recording the label *lx*1 in the error component *ψ* (by *lx*1 ∈ *ψ*).

Andrew Secure RPC protocol with type flaw corrected

An improved version of Andrew Secure RPC protocol is suggested in [[6](#_bookmark9)] in order to prevent the above mentioned type flaw attack. The fixing amounts to inserting another component *NA* into the encryption in the fourth message, as shown below,

4'*. B* → *A* : {*K*'*,N* ' *, NA*}*K*

*B*

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. *ψ* = ∅.

# 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 pro- cesses 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](#_bookmark14)] also adopted the technique of tagging each message field with intended type, and later on, [[12](#_bookmark15)] 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](#_bookmark13)], 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](#_bookmark12)] and freshness [[8](#_bookmark11)]. It is very easy to combine tags with those techniques, thus ob- taining 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](#_bookmark21)], as considered in [[13](#_bookmark20)], 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.

# Acknowledgement

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.

1. 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, Universit`a di Pisa, 2007.
2. 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.

1. M. Burrows and M. Abadi and R. Needham. A Logic of Authentication. ACM. *Transactions in Computer Systems*, 8(1), pp. 18-36, 1990.
2. D. Dolev and A.C. Yao. On the Security of Public Key Protocols. IEEE TIT, IT-29(12):198-208, 1983.
3. 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)*.

1. 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.
2. 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.
3. 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.
4. Y. Li, W. Yang and J. Huang. *Journal of Information Science and Engineering*, 21:59-84, 2005.
5. C. Meadows. Identifying potential type confusion in authenticated messages. *In. Proc. of Workshop on Foundation of Computer Security*, pp. 75-84, 2002.
6. R. Milner. Communicating and mobile systems: the *π*-calculus. Cambridge University Press, 1999.
7. 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.
8. C.R. Nielsen, F. Nielson, H.R. Nielson. Cryptographic Pattern Matching. ENTCS 168, pp. 91-107, 2007.
9. M. Satyanarayanan. *Integrating security in a large distributed system*. ACM Transactions on Computer Systems, 7(3):247–280, 1989.
10. 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.
11. T.Y.C. Woo and S.S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24-37, 1994.