

# Inferring Authentication Tags \*

Riccardo Focardi Matteo Maffei Francesco Placella  
Università Ca'Foscari di Venezia  
Dipartimento di Informatica  
Via Torino 155, 30172 Mestre (VE), Italy  
{focardi,maffei,fplacell}@dsi.unive.it

## ABSTRACT

We present PEAR (Protocol Extendable Analyzer), a tool automating the two static analyses for authentication protocols presented in [7, 8]. These analyses are based on a tagging scheme that describes how message components contribute in achieving authentication. The tool provides a tag inference procedure that allows users to analyze untagged protocol specifications. When a protocol is successfully validated, tags give users precise information on *how* and *why* authentication is guaranteed. Notably, the tool receives in input both the protocol specification and the validation rules. Both validation and tag inference are parametric with respect to the validation rules, thus allowing users to easily implement new rules/analyses with no need of modifying the underlying procedures.

## 1. INTRODUCTION

Authentication protocols allow a *claimant* to prove its claimed identity to a *verifier*. Even if authentication protocols are very small pieces of code, in the last years many such protocols have been shown to be flawed (see, e.g., [11, 12, 17, 22, 23]). For this reason, a number of analysis techniques [1, 2, 3, 4, 14, 15, 20] have been proposed, aiming at finding attacks or proving protocol correctness.

Following this line of research, in [7, 8], we proposed two static analysis techniques for authentication protocols. Our main motivation was to characterize authentication at a language-level, thus proving such a property by just inspecting the protocol code. This has been achieved by exploiting a tagging mechanism for messages that makes the interpretation of certain, critical ciphertext components unambiguous. The untagged part of a ciphertext forms the message's payload, while the tagged components include entity identifiers, tagged by *Id*, messages to be authenticated, tagged by *Auth* and nonces. Nonce tags are more elaborate, as they convey information on the role that identities play in the authentication protocol. Moreover, they disambiguate the role of the ciphertext itself, specifying whether it is a challenge or a response. Specifically,

\*Work partially supported by EU Contract IST-FET-2001-32617 "Models and Types for Security in Mobile Distributed Systems" (MyThS).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WITS'05, January 10, 2005, Long Beach, CA, USA.  
Copyright 2005 ACM 1-58113-980-2/05/01 ...\$5.00.

nonces are tagged either by *Claim* or *Verif*, according to the role played by the identity tagged by *Id* in the authentication task. In case the ciphertext constitutes the protocol challenge, the tags are interrogative, denoted *Claim?* or *Verif?*.

As an example, let us consider the two following simple protocols, allowing *Alice* (*A*) to authenticate message *M* with *Bob* (*B*). We assume *k* to be a long-term key shared between *A* and *B*:

$$\begin{array}{ccc} A & & B \\ \leftarrow n & & \leftarrow \{A, n, M\}_k \\ \rightarrow \{B, n, M\}_k & & \rightarrow n \end{array}$$

Following [15, 16], we say that the protocol on the left side exploits a *POSH* (*Public-Out Secret-Home*) nonce-handshake, since the nonce is sent as cleartext (the challenge) and received into a ciphertext (the response). The protocol on the right side, instead, follows a *SOPH* (*Secret-Out Public-Home*) nonce-handshake, as the nonce is sent as ciphertext (the challenge) and received into a plaintext (the response). In the former, *B* represents the verifier of the authentication task, while in the latter *A* is the claimant. The two protocols can be tagged as follows:

$$\begin{array}{ccc} A & & B \\ \leftarrow n & & \leftarrow \{A, n, M\}_k \\ \rightarrow \{B, n, M\}_k & & \rightarrow n \end{array}$$
$$\begin{array}{ccc} A & & B \\ \leftarrow \{Id(B), Verif(n), Auth(M)\}_k & & \rightarrow n \end{array}$$

Tags provide semantics to ciphertexts thus disambiguating their role in the authentication task. Moreover, they make explicit the underlying protocol logic, pointing out the mechanisms used for achieving authentication. In the example above, identifiers are all tagged with *Id* and the message *M* is tagged with *Auth*. In the first protocol, the nonce is tagged with *Verif* to indicate that *B* is the intended verifier of the protocol session. In the second protocol, *Claim?* means that the verifier (*B*) is asking *A* to be the claimant of the protocol session.

The static analyses we developed validate tagged specifications and, basically, the only human effort required is tagging protocol messages. Even though this task is not hard, finding the right tags somehow implies that the user already knows the authentication mechanisms exploited by the protocol, namely *why* the protocol is correct.

In this paper, we present PEAR (Protocol Extendable Analyzer), a tool implementing the static analyses presented in [7, 8]. The most interesting features are summarized below:

**Tag and Type Inference** The analysis can be performed even if users do not provide tagged specifications: tags can be inferred by PEAR, further reducing the human effort. The inference algorithm receives as input the (untagged) protocol specification and yields all the possible taggings allowing the protocol to be validated. To achieve this, the tool also infers types of nonces determining whether they are used in *POSH* or *SOPH* hand-shakes.

Since finding tags and nonce types somehow means understanding *why* the protocol is correct, PEAR is able to validate protocols by also making explicit the underlying logic and authentication mechanisms.

**Flexibility** The tool is parametric with respect to the validation rules: as explained in §4.1, the tool receives as input both the protocol and the validation rules. Rules must adhere to a fixed shape which is, however, fairly expressive. As a matter of fact, both the syntactic analysis of [7] and the type and effect system of [8] can be given as input to the tool. Even the inference algorithm is parametric with respect to the validation rules. This guarantees that the inference procedure works even if the analysis is refined by adding (or modifying the current) validation rules.

**Plan of the paper** The rest of the paper is organized as follows: §2 gives a brief outline of the  $\rho$ -spi calculus. §3 summarizes the type and effect system of [8]. In §4, we present PEAR, focusing on the validation and inference procedures. In §5, we propose some case-studies (a variant of the ISO Two-Pass Unilateral Authentication Protocol [18] and a variant of the CCITT X.509(3) [9, 10]). In §6, we conclude with some final remarks.

## 2. THE $\rho$ -spi calculus

The  $\rho$ -spi calculus, originally proposed in [7] and then extended in [8], derives from the spi calculus [2], and inherits many of the features of *Lysa* [3], a version of the spi calculus proposed for the analysis of authentication protocols.  $\rho$ -spi differs from both calculi in several respects: it incorporates the notion of tagged message exchange from [6], it provides new authentication-specific constructs, and offers primitives for declaring process identities and keys.

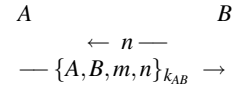
The syntax is reported in Table 1 and described below. We presuppose two countable sets:  $\mathcal{N}$  of names and  $\mathcal{V}$  of variables. We reserve  $k, m, n$  for names and  $x, y, z$  for variables, with  $a$  ranging over both names and variables. Identities  $I, D$ , ranged over by  $I$  and  $J$ , are a subset of names. Identities are further partitioned into *trusted principals*  $I_P$ , ranged over by  $A$  and  $B$ , and *enemies*  $I_E$ , ranged over by  $E$ . Both names and variables can be tagged, noted  $C(a)$ . Tags, denoted by  $C$ , are a special category of names. The pair composed by a public key and the corresponding private one is noted by  $\text{Pub}(m), \text{Priv}(m)$ , similarly to [2].

*Processes* (or *protocols*), ranged over by  $P, Q$  are the parallel composition of principals. Each principal is a sequential process associated with an identity  $I$ , noted  $I \triangleright S$ . The replicated form  $I \triangleright! S$  indicates an arbitrary number of copies of  $I \triangleright S$ . In order to allow the sharing of keys among principals, we provide  $\rho$ -spi with let-bindings:  $\text{let } k = \text{sym-key}(I_1, I_2).P$  declares (and binds) the long-term key  $k$  shared between  $I_1$  and  $I_2$  in the scope  $P$ . Similarly,  $\text{let } k = \text{asym-key}(I).P$  declares, and binds in the scope  $P$ , the key pair  $\text{Pub}(k), \text{Priv}(k)$  associated to  $I$ .

Sequential processes may never fork into parallel components: this assumption helps assign unique identities to (sequential) processes, and involves no significant loss of expressive power as protocol principals are typically specified as sequential processes, pos-

sibly sharing some long-term keys. The sequential process  $\mathbf{0}$  is the null process that does nothing, as usual. Process  $\text{new}(n).S$  generates a fresh name  $n$  local to  $S$ . We presuppose a unique (anonymous) public channel, the network, from/to which all principals, including intruders, read/send messages. Similarly to *Lysa*, our input primitive may (atomically) test part of the message read, by pattern-matching. If the input message matches the pattern, then the variables occurring in the pattern are bound to the remaining subpart of the message; otherwise the message is not read at all. For example, process  $\text{in}(\text{Claim}(x)).P$  may only read messages of the form  $\text{Claim}(M)$ , binding  $x$  to  $M$  in  $P$ . Encryption just binds  $x$  to the encrypted message, while decryption checks whether the message contained in  $x$  matches the form  $\{M_1, \dots, M_n\}_{M_0}$  (or the form  $\{\{M_1, \dots, M_n\}_{M_0}\}$ , i.e., is a tuple of  $n$  messages encrypted with the appropriate key. Only in this case  $x$  is decrypted and the variables in the patterns  $\mathcal{M}_1, \dots, \mathcal{M}_n$  get bound to the decrypted messages. Similarly to the input primitive, decryption also may test part of the decrypted messages by pattern-matching. Finally, the primitives  $\text{run}(I_1, I_2, \mathcal{M}).S$  and  $\text{commit}(I_1, I_2, \mathcal{M}).S$  declare that the sequential process  $I_1$  is starting and committing respectively, a protocol session with  $I_2$  for authenticating message  $\mathcal{M}$ . These constructs are used to check the *correspondence assertions* [24].

*Example 1.* Let us consider the following simple protocol.



The specification of the protocol in  $\rho$ -spi is reported in Table 2; after declaring the long-term key shared between  $A$  and  $B$ , an unbounded number of instances of  $A$  as responder and an unbounded number of instances of  $B$  as initiator are run in parallel.  $B$  generates a new nonce and sends it on the network.  $A$  receives it and generates the message she is willing to authenticate. Finally, she encrypts the nonce and the message together with her own identity and the identity label of the intended receiver and sends the ciphertext on the network.  $B$  receives it, checks the freshness by verifying the presence of the nonce and commits on the received message  $x$ .

**Operational Semantics.** We define the operational semantics of  $\rho$ -spi in terms of *traces*, after [5]. A trace is a possible sequence of *actions* performed by a process. Each process primitive has an associated action and we denote with  $\text{Act}$  the set of all possible actions. The dynamics of the calculus is formalized by means of a transition relation between *configurations*, i.e., pairs  $\langle s, P \rangle$ , where  $s \in \text{Act}^*$  is a trace,  $P$  is a (closed) process. Each transition  $\langle s, P \rangle \longrightarrow \langle s :: \alpha, P' \rangle$  simulates one computation step in  $P$  and records the corresponding action  $\alpha$  in the trace.

Principals do not directly synchronize with each other. Instead, they may receive from the unique channel an arbitrary message  $M$  known by the environment, which models the Dolev-Yao intruder: the environment knows all the identity labels, the messages sent on the network, the content of ciphertexts whose decryption key is known, ciphertexts created by its knowledge and all the keys declared as owned by  $E$  together with all the public keys. Finally, it may create fresh names not appearing in the trace. The transition relation is given in detail in [8].

**Definition 1 (Traces)** *The set  $T(P)$  of traces of process  $P$  is the set of all the traces generated by a finite sequence of transitions from the configuration  $\langle \varepsilon, P \rangle$ :  $T(P) = \{s \mid \exists P'. \langle \varepsilon, P \rangle \longrightarrow^* \langle s, P' \rangle\}$*

The notion of safety (similar to the one in [14]) formalizes the *correspondence* property of [24], also known as *agreement* [19].

**Table 1** The syntax of  $\rho$ -spi calculus.

**Notation:**  $C$  over tags:  $\{\text{Claim, Claim?, Verif, Verif?, Id, Auth}\}$ ,  $a$  over names and variables.

$\mathcal{M} ::= \text{Patterns}$		$P, Q ::= \text{Processes}$	
$m, n, k$	names	$I \triangleright S$	(principal)
$x, y, z$	variables	$I \triangleright! S$	(replication)
$C(a)$	tagged data	$P Q$	(composition)
$\text{key}(a)$	$\text{key}(key \in \{\text{Pub/Priv}\})$	$\text{let } k = \text{sym-key}(I_1, I_2).P$	(symmetric-key assignment)
		$\text{let } k = \text{asym-key}(I).P$	(asymmetric-key assignment)
$S ::= \text{Sequential processes}$			
$\mathbf{0}$			(nil)
$\text{new}(n).S$			(restriction)
$\text{in}(\mathcal{M}_1, \dots, \mathcal{M}_n).S$			(input)
$\text{out}(\mathcal{M}_1, \dots, \mathcal{M}_n).S$			(output)
$\text{encrypt}\{\mathcal{M}_1, \dots, \mathcal{M}_n\}_{\mathcal{M}_0} \text{ as } x.S$			(symmetric encryption)
$\text{encrypt}\{ \mathcal{M}_1, \dots, \mathcal{M}_n \}_{\mathcal{M}_0} \text{ as } x.S$			(asymmetric encryption)
$\text{decrypt } x \text{ as } \{\mathcal{M}_1, \dots, \mathcal{M}_n\}_{\mathcal{M}_0}.S$			(symmetric decryption)
$\text{decrypt } x \text{ as } \{ \mathcal{M}_1, \dots, \mathcal{M}_n \}_{\mathcal{M}_0}.S$			(asymmetric decryption)
$\text{run}(I_1, I_2, \mathcal{M}).S$			(run)
$\text{commit}(I_1, I_2, \mathcal{M}).S$			(commit)

**Table 2** Simple Protocol in  $\rho$ -spi calculus

$\text{Protocol}$	$\triangleq$	$\text{let } k_{AB} = \text{sym-key}(A, B) . (A \triangleright !\text{Responder} \mid B \triangleright !\text{Initiator})$
$\text{Responder}$	$\triangleq$	$\text{in}(x).\text{new}(m).\text{run}(A, B, m).\text{encrypt}\{A, B, m, x\}_{k_{AB}} \text{ as } z.\text{out}(z).$
$\text{Initiator}$	$\triangleq$	$\text{new}(n).\text{out}(n).\text{in}(z).\text{decrypt } z \text{ as } \{A, B, x, n\}_{k_{AB}}.\text{commit}(B, A, x)$

**Definition 2 (Safety)** A trace  $s$  is safe if and only if whenever  $s = s_1 :: \text{commit}(B, A, M) :: s_2$ , then  $s_1 = s'_1 :: \text{run}(A, B, M) :: s''_1$ , and  $s'_1 :: s''_1 :: s_2$  is safe. A process  $P$  is safe if,  $\forall s \in T(P), s$  is safe.

Informally, a trace is safe if every  $\text{commit}(B, A, M)$  is preceded by a distinct  $\text{run}(A, B, M)$ . This guarantees that whenever  $B$  is convinced of the identity of  $A$  sending  $M$ , then  $A$  has indeed started the protocol with  $B$  for authenticating  $M$ .

### 3. THE TYPE AND EFFECT SYSTEM

The type and effect system applies to protocol (tagged) narrations that may be represented as  $\rho$ -spi processes of the form

$$\text{keys}(k_1, \dots, k_n).(I_1 \triangleright !S_1 \mid \dots \mid I_m \triangleright !S_m)$$

where  $\text{keys}(k_1, \dots, k_n)$  represents a sequence of let binding for the long-term keys  $k_1, \dots, k_n$ . The analysis proceeds by examining each process  $\text{keys}(k_1, \dots, k_n).I_i \triangleright S_i$ , and attempts to validate  $S_i$  under the key assignment determined by the prefix  $\text{keys}(k_1, \dots, k_n)$ .

Types regulate the use of terms in the authentication task (nonces, long-term keys, etc.) while effects record the history of a process (input, decryptions, nonce generations, etc.). Processes are typed according to the judgement  $\Gamma \vdash P : e$ . Intuitively,  $\Gamma \vdash P : e$  means that the process  $P$  can be typed under the typing environment  $\Gamma$  and the hypotheses expressed by the effect  $e$ . Judgment  $I; \Gamma \vdash S : e$ , for the sequential process  $S$  executed by the entity  $I$ , has the same intuitive meaning.

To show how the analysis works, we illustrate two of the typing rules, allowing a principal to authenticate a responder in a POSH

nonce-handshake (Table 3). Rule NEW NAME regulates the generation of a nonce  $n$ . The type of  $n$  is  $Un$  (Untrusted) since the nonce has to be sent as cleartext on the network; the freshness of  $n$  is recorded by adding the atomic effect  $\text{fresh}(n)$  to the effect for the continuation process. (An effect is an ordered multiset of atomic effects.)

$A$  can authenticate  $M$  as originated by  $I$ , by asserting an event  $\text{commit}(A, I, M)$ . This primitive is regulated by POSH COMMIT (VERIF). Notice that  $A$  must have previously received a ciphertext providing the evidence that  $I$  is really willing to authenticate  $M$  with  $A$ . Moreover that ciphertext has to contain a fresh nonce  $n$ . When typing the following part of the process the atomic effect  $\text{fresh}(n)$  is removed since the nonce is not fresh anymore. For a full explanation of the type and effect system, please refer to [8].

Our main result states that if a process can be typed with empty effect and empty typing environment, then every trace generated by that process is safe.

**Theorem 1 (Safety)** If  $\emptyset \vdash P : \square$ , then  $P$  is safe.

Interestingly, our analysis is strongly compositional, as stated by the following theorem.

**Theorem 2 (Strong Compositionality)** Let  $P$  be  $\text{keys}(k_1, \dots, k_n).(I_1 \triangleright !S_1 \mid \dots \mid I_n \triangleright !S_n)$ . Then  $\emptyset \vdash P : \square$  iff  $\emptyset \vdash \text{keys}(k_1, \dots, k_n).I_i \triangleright !S_i : \square, \forall i \in [1, n]$ .

Intuitively, a protocol is safe if so are all the protocol participants. In addition, judging a participant safe only requires knowledge of

**Table 3** Rules for *POSH* nonce-handshakes

$$\frac{\text{NEW NAME} \quad A; \Gamma, n : Un \vdash S : e + [\text{fresh}(n)]}{A; \Gamma \vdash \text{new}(n : Un).S : e}$$

$$\frac{\text{POSH COMMIT (VERIF)} \quad J; \Gamma \vdash S : e \quad \Gamma \vdash n, M, \tilde{M} : Un \quad \Gamma \vdash K : \text{key}_{\text{sym}}(A, I) \quad \text{dec}\{\text{ld}(A), \text{Verif}(n), \text{Auth}(M), \tilde{M}\}_K \in e}{A; \Gamma \vdash \text{commit}(A, I, M).S : e + [\text{fresh}(n)]}$$

the long-term keys it shares with other participants. This is a fairly mild assumption as the information conveyed by the keys is relative to identities of the parties sharing them, not to the protocol they are running. Consequently, unlike similar results proved of existing typing systems for authentication, notably [14, 15], Theorem 2 may be directly applied to the verification of multi-protocol systems (see [13, 21] for details).

## 4. PEAR

As mentioned in §1, the tool receives in input both the protocol specification and the validation rules. These must adhere to a common shape which is presented in §4.1. The validation algorithm and the tag inference are discussed in §4.2 and §4.3, respectively.

### 4.1 Validation Rules

As mentioned above, both the validation and the inference procedures are parametric with respect to the validation rules. These are supposed to adhere to the following common shape:

```
rule [RULE]
{
  { instr( $p_1, \dots, p_n$ ) }
  conditions { /* hypothesis definition */ }
  actions { /* thesis definition */ }
}
```

where RULE is the name of the validation rule, instr is the primitive to be validated,  $p_1, \dots, p_n$  are (optional) *parameters*. These may be names, variables, identities and containers and are declared before the validation rules. As an example,

```
declarations
{
  Principal I, J
  Names n, m
  Variables x
  Keys k1, k2
}
```

declares that  $I, J$  range over identities,  $n, m$  over names,  $x$  over variables and  $k1, k2$  over keys.

The conditions are the hypotheses under which the validation rule can be applied, while the actions represent the thesis. Both conditions and actions are defined on containers. Different kinds of containers are defined in PEAR: sets and multisets, both ordered and unordered. The element of a container can possibly be associated with a value. The following are some operations defined on containers:

$e.\text{add}(p)$	adds $p$ into the container $e$
$e.\text{add}(p, \text{val})$	adds $p$ with associated value $\text{val}$ into $e$
$e.\text{contains}(p)$	tests if $p$ is in $e$
$e.\text{contains}(p, \text{val})$	tests if $p$ is associated with $\text{val}$ in $e$
$e.\text{delete}(p)$	removes $p$ from $e$

In the type and effect system case, two containers are needed: one for the typing environment (an unordered set whose elements are associated with their type) and the other one for the effect (a multiset of atomic effects).

As an example, the typing rule POSH COMMIT (VERIF), in Table 3, is implemented as follows:

```
rule [POSH COMMIT (VERIF)](A)
{
  { commit(A, I, M) }
  conditions {
    e.contains(fresh(n)) &
    e.contains dec{ld(A), Verif(n), Auth(M), ...}_K &
    Γ.checktype(n, M, ... : Un) &
    Γ.checktype(K : key_sym(A, I)) }
  actions { e.delete(fresh(n)) }
}
```

The pattern  $\dots$  represents an unbounded number of untagged names and variables. The operation  $\Gamma.\text{checktype}(m_1, \dots, m_n : Un)$  checks that  $m_1, \dots, m_n$  have type  $Un$  in the container  $\Gamma$ . In POSH COMMIT (VERIF),  $\text{fresh}(n)$  is required to be in the effect  $e$  in the thesis, while it is removed when typing the continuation process. Accordingly,  $\text{fresh}(n)$  is required to be in the container  $e$  (the condition  $e.\text{contains}(\text{fresh}(n))$ ) and then it is removed from  $e$  (the action  $e.\text{delete}(\text{fresh}(n))$ ).

### 4.2 Validation Algorithm

The validation algorithm independently analyzes each sequential process and works in a top-down fashion, trying to validate the first primitive, possibly updating the typing environment and effect containers, then moving to the second primitive and so on.

Validation rules are defined on parameters. During the validation, these are instantiated to the terms appearing in the instruction which is going to be analyzed. All the remaining parameters are implicitly universally quantified. As an example, let us suppose to analyze the instruction  $\text{commit}(A, B, x)$  by POSH COMMIT (VERIF) under the following typing environment  $\Gamma$  and effect  $e$ :

$$\begin{aligned} \Gamma &\triangleq \{n1 : Un, n2 : Un, n3 : Un, x : Un, k1 : \text{key}_{\text{sym}}(A, B)\} \\ e &\triangleq [\text{fresh}(n1), \text{fresh}(n2), \\ &\quad \text{dec}\{\text{Verif}(n2), \text{ld}(A), \text{Auth}(x), n1, n3\}_{k1}] \end{aligned}$$

The algorithm creates a substitution function between parameters and patterns. In this case, the following is the only possible substitution:

$$[A = A, I = B, M = x, K = k1, n = n2, \dots = n1, n3]$$

Notice that  $\dots$  is instantiated to  $n1, n3$ . Generally, parameters instantiation might not be unique, resulting in more than one possible substitution. Moreover, more than one validation rule might be successfully applied to a primitive. This means that backtracking might be required when typing a sequential process.

### 4.3 Tag and Type Inference

Tagging protocol messages and typing nonces are the only non-trivial human efforts required by our static analysis. An interesting feature of PEAR is the tag and type inference procedure: given an untagged and untyped protocol, PEAR finds all the possible tags and types that validate the protocol. Notably, the inference algorithm is parametric with respect to the validation rules. This means that it scales up to new analyses and to extensions of the presently implemented ones.

The human effort is thus limited to giving an untagged and untyped specification in  $\rho$ -spi calculus and specifying which encryption corresponds to which decryption.

Typing nonces is quite trivial, since it only amounts to determine whether they are used in *POSH* or *SOPH* hand-shakes. On the contrary, tags might be non-trivial to specify since they require the user to know *why* the protocol is correct and in which extent message components contribute to achieving authentication. Inferring this information, automatically provides a proof of correctness for the protocol (with *unbounded* number of sessions), and explains the user *how* and *why* the protocol guarantees the property of interest.

#### 4.3.1 The Algorithm

The inference algorithm receives as input the (untagged) protocol specification and yields all the possible taggings allowing the protocol to be validated. The algorithm is very simple and is based on the validation one. Basically, before starting the inference procedure, the validation rules are inspected and each protocol instruction is associated with the set of all potential taggings for such instruction. The validation algorithm is modified so that it will backtrack even on these sets of potential taggings.

The algorithm scheme is reported in Table 4. First, sequential processes are ordered (function *reorder\_protocol*) according to some heuristics defined below and then each primitive is inspected in order to find all the taggings possibly allowing its validation (function *find\_tags*). This function requires an inspection of the validation rules. By this function, it is possible to reduce hugely the number of taggings to be considered (for instance, when analyzing an encryption, only the tag combinations allowing at least one encryption validation rule to be applied are considered, while the other ones are discarded). Notice that this function makes the procedure parametric with respect to the validation rules, i.e., changing the rules would change, accordingly, the set of potential taggings. The reordered protocol and this set of potential taggings are given as input to the modified validation procedure which basically tries to validate the protocol using all the possible potential taggings. The *limit* parameter is a way for the user to give an upper bound to the backtrack iterations. At the end, the algorithm returns the (possibly empty) set of taggings allowing the protocol to be validated.

#### 4.3.2 Scalability

The solution adopted is probably not the more efficient way for implementing tag inference. However, our main aim was to develop a flexible and scalable procedure working on every analysis given as input to the tool. This allows us to refine the analysis in order to capture more sophisticated authentication mechanisms without affecting the inference procedure. Indeed, by fixing the set of validation rules, smarter algorithms might be developed, but this would require to change the inference procedure when extending/modifying the analysis.

#### 4.3.3 Decidability and Performance

The inference procedure always terminates since the number of tags and ciphertext components that may be potentially tagged, is

finite. Since the procedure is a simple backtrack scheme, the performance decreases quickly when the protocol size increases. However, protocol specifications are typically very small and we never experienced long termination times in all the example protocols we have considered. Nevertheless, to improve performance, we have implemented some heuristics for deciding the order according to which sequential processes should be inspected by the inference algorithm. These heuristics have been proved effective in improving the algorithm performance.

**Multiplicity** First, the validation rules are scanned in order to find out the one with the minimal number of tags involved. Let us refer to that number with *min*. The number of possible taggings for each encryption and decryption grows with the number of its message components. For reducing the number of backtracking required, encryptions and decryptions with number of message components close to *min* should be inspected before the ones with several message components.

**Distance** The distance between decryption/encryption primitives and the commit one is considered since the more are the primitives in the middle, the more are the analysis steps required for validating the commit. Since each step may introduce a backtrack, this should reduce the backtracking steps needed to validate the protocol.

**Complementarity** When a decryption is tagged, the complementary encryption (belonging to a different sequential process) is also tagged, and this new tagging is immediately checked. As a matter of fact, while decryptions are regulated by a unique typing rule which is not influenced by the tagging, encryptions are regulated by different and subtler rules which fail or validate a primitive according to the tags. It is better to know as soon as possible which taggings cannot be applied because of the effect on sibling processes. This heuristics helps pruning “useless” backtracking.

As shown<sup>1</sup> in Table 5, the inference procedure terminates for most of the protocols we have considered in a few seconds. However, the performance of the algorithm changes appreciably according to the ordering of the sequential processes. Each column is associated with a possible ordering. *I* stands for initiator, *R* for responder, *T* for trusted third party. The first three protocols are not based on a trusted third party, this is why there are only two possible orderings. The ordering selected by our heuristics is in bold font. Notice that, in all the performed tests, the heuristics always selected the ordering that maximize the performance of the inference procedure. We leave as future work a formal study of the complexity of the algorithm.

## 5. EXAMPLES

In this section, we propose two case-studies: a variant of the ISO Two-Pass Unilateral Authentication Protocol [18] and a variant of the CCITT X.509(3) [9, 10].

### 5.1 A simple protocol

The most interesting aspect when analyzing a protocol by PEAR is the tag inference procedure. Indeed, inferred tags explain the logics which the protocol is based on and, as we will see, may suggest simplifications. Let us consider again the simple protocol presented in Table 2. PEAR finds out the two following taggings for this protocol.

<sup>1</sup>Tests were conducted using a PIII 700 Mhz based-system with 320 MB RAM.

**Table 4** Inference Algorithm

*input* : *protocol, limit*  
*output* : *taggings*

*taggings* =  $\emptyset$   
*ordered\_protocol* := *reorder\_protocol(protocol)*  
*possible\_tags* := *find\_tags(ordered\_protocol)*  
*taggings* := *modified\_protocol\_validation(ordered\_protocol, possible\_tags, limit)*

**Table 5** Algorithm Results in Milliseconds

			IR	RI
Amended SPLICE-AS			1693	<b>1682</b>
ISO Two-Pass Unilateral Authentication			1382	<b>1322</b>
ISO Three Steps Mutual Authentication			1332	<b>1282</b>

	IRT	ITR	RIT	RTI	TIR	TRI
ANS Shared Key	13369	28811	11006	<b>5298</b>	40539	18707
Wide Mouthed Frog	10415	7140	<b>3795</b>	4026	14401	9814
Woo-Lam	3815	2413	<b>1772</b>	1873	3915	3084

## Tagging 1

A B

$\leftarrow n \leftarrow$

$\leftarrow \{ \text{Id}(A), B, \text{Auth}(m), \text{Claim}(n) \}_{k_{AB}} \rightarrow$

## Tagging 2

A B

$\leftarrow n \leftarrow$

$\leftarrow \{ A, \text{Id}(B), \text{Auth}(m), \text{Verif}(n) \}_{k_{AB}} \rightarrow$

The first tagging makes explicit that  $A$  is the claimant of the authentication session, while the second tagging indicates that  $B$  is the intended verifier. In both cases  $n$  is correctly detected as a nonce and  $m$  as a message to authenticate. In general, identity labels are needed in order to specify who is the originator and who is the intended receiver of the ciphertext. This breaks the symmetry of shared-key cryptography and avoids dangerous reflection attacks, in which the spy reflects back the nonce to the initiator in order to obtain from him a correct encryption.

This inference result shows that one identity label is enough (since if  $B$  is the verifier, then  $A$ , the other entity owning the encryption key, must be the claimant). Indeed, the protocol might be simplified by deleting the untagged identity from the ciphertext. The two taggings suggest the following two simplifications:

A B

$\leftarrow n \leftarrow$

$\leftarrow \{ A, m, n \}_{k_{AB}} \rightarrow$

A B

$\leftarrow n \leftarrow$

$\leftarrow \{ B, m, n \}_{k_{AB}} \rightarrow$

The latter protocol is known as ISO Two-Pass Unilateral Authenti-

cation Protocol [18].

**5.2 A Variant of CCITT X.509 (3)**

This protocol is a variant of the BAN modified version of CCITT X.509 [9, 10]. We have added the first nonce-handshake, where  $n$  is sent by  $B$  to  $A$  for being encrypted in the second message.

A B

$\leftarrow n \leftarrow$

$\leftarrow \{ |n, n_A, B, x_A, \{ |y_A| \}_{\text{Pub}(k_B)}| \}_{\text{Priv}(k_A)} \rightarrow$

$\leftarrow \{ |n_B, A, n_A, x_B, \{ |y_B| \}_{\text{Pub}(k_A)}| \}_{\text{Priv}(k_B)} \leftarrow$

$\leftarrow A, \{ |B, n_B| \}_{\text{Priv}(k_A)} \rightarrow$

The aim of the protocol is to allow  $A$  to authenticate with  $B$  both  $x_A$  and  $y_A$ . The privacy of  $y_A$  is ensured by the encryption with the public key of  $B$ . Similarly,  $B$  authenticates  $x_B$  and  $y_B$  with  $A$ , preserving the secrecy of  $y_B$ . Although our current analysis cannot deal with secrecy (this is part of the ongoing work), we analyzed the authentication guarantees provided by the protocol. The code of the protocol is in Table 6. We analyze an unbounded number of instances of  $A$  and  $B$  playing both the initiator and the responder roles. As outlined by the run and commit primitives, we check the agreement between  $A$  and  $B$  on  $x_A$  and  $x_B$ . The inference procedure provides the following tagging:

A B

$\leftarrow n \leftarrow$

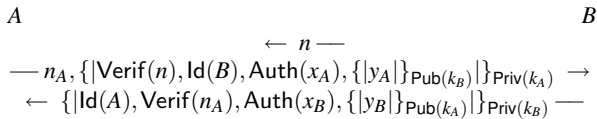
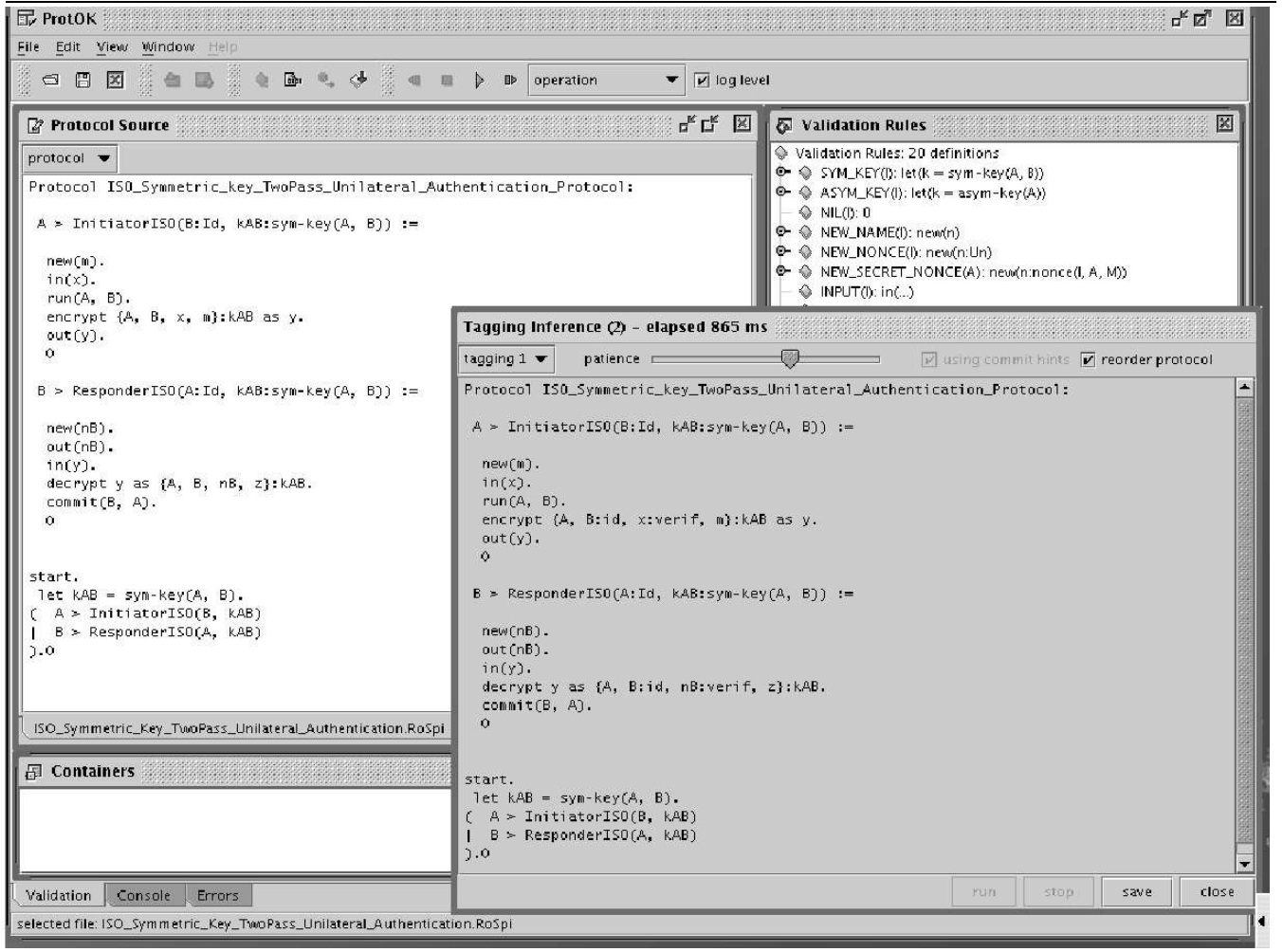
$\leftarrow \{ |\text{Verif}(n), n_A, \text{Id}(B), \text{Auth}(x_A), \{ |y_A| \}_{\text{Pub}(k_B)}| \}_{\text{Priv}(k_A)} \rightarrow$

$\leftarrow \{ |n_B, \text{Id}(A), \text{Verif}(n_A), \text{Auth}(x_B), \{ |y_B| \}_{\text{Pub}(k_A)}| \}_{\text{Priv}(k_B)} \leftarrow$

$\leftarrow A, \{ |B, n_B| \}_{\text{Priv}(k_A)} \rightarrow$

Notice that the nonce  $n_B$  is never tagged. This suggests that the third nonce-handshake becomes redundant in this variant, leading us to the following simplification:

Figure 1 Inference Procedure



This protocol requires a ciphertext exchange less than the (BAN modified version of) CCITT X.509 (3). Nevertheless, it still guarantees to  $B$  that  $A$  is willing to authenticate  $x_A$  with him, and it still guarantees to  $A$  that  $B$  is willing to authenticate  $x_B$  with her. These are the intended requirements of the original CCITT X.509(3). However, the two protocols are somehow different. In the original version  $B$  also has confirmation that  $A$  has in fact received  $x_B$  (and  $y_B$ ). In the simplified one this does not happen since the last message has been removed. Our analysis does not capture this additional authentication property because we are basically checking two independent unilateral authentication guarantees. The analysis might be extended to deal with a form of mutual authentication, where  $A$  and  $B$  are both agreeing on the exchanged values  $x_A$  and  $x_B$ . We are currently studying how to incorporate this in our type and effect system, and, more generally, we are trying to characterize which security properties we can express by run-commit events.

## 6. CONCLUSION AND FUTURE WORK

In this paper we have presented PEAR, a tool automating the static analyses proposed in [7, 8]. The tool provides a tag inference procedure, reducing a lot the human effort required for the analysis. Interestingly, both the validation and the inference algorithm are parametric with respect to the validation rules, allowing the analysis to be refined without involving changes in the tool.

Of course the analysis cannot be extended arbitrarily as Theorems 1 (Safety) and 2 (Strong Compositionality) should still hold for the new set of rules. This amounts to extending the theorem proofs in order to cover the new (or modified) rules. We plan to investigate constraints on the typing rules, which are sufficient conditions for the safety and the compositionality of the type and effect system. This would allow one to easily extend the analysis by just showing that the new set of rules respects such constraints. It would be very useful to study constraints which are simple enough to be directly checked by PEAR so that the extension task becomes completely automated.

We are currently trying to characterize the class of protocols that can be validated by our analysis, i.e., for which the analysis is complete. Examples of authentication mechanisms that we do not consider yet are the *trust-witness* mechanism studied in [14, 15], the use of (messages encrypted with) public keys instead of simple identity labels, and the authentication of encrypted messages (as,

**Table 6** Variant of CCITT X.509 (3) in  $\rho$ -spi calculus

$\begin{aligned} & \text{Protocol}_{CCITT} \triangleq \\ & \text{let } k_A = \text{asym-key}(A). \\ & \text{let } k_B = \text{asym-key}(B). \\ & (  \forall I \in \{A, B\} \\ & \quad \forall J \in \{A, B\}, J \neq I \\ & I \triangleright ! \text{Initiator}_{CCITT}(I, J, k_I, \text{Pub}(k_J))   \\ & I \triangleright ! \text{Responder}_{CCITT}(I, J, k_I, \text{Pub}(k_J))) \\ \\ & \text{Responder}_{CCITT}(I, J, \text{Priv}(k_I), \text{Pub}(k_J)) \triangleq \\ & \text{in}(z). \\ & \text{new}(m_x). \\ & \text{new}(m_y). \\ & \text{new}(n_A). \\ & \text{run}(I, J, m_x). \\ & \text{encrypt } \{ m_y \}_{\text{Pub}(k_J)} \text{ as } z_y. \\ & \text{encrypt } \{z, n_A, J, m_x, z_y\}_{\text{Priv}(k_I)} \text{ as } z_1. \\ & \text{out}(z_1). \\ & \text{in}(z_2). \\ & \text{decrypt } z_2 \text{ as } \{x, I, n_A, x_B, y\}_{\text{Pub}(k_J)}. \\ & \text{decrypt } y \text{ as } \{y_B\}_{\text{Priv}(k_I)}. \\ & \text{encrypt } \{J, x\}_{\text{Priv}(k_I)} \text{ as } z_3. \\ & \text{out}(z_3). \\ & \text{commit}(I, J, x_B) \end{aligned}$	$\begin{aligned} & \text{Initiator}_{CCITT}(I, J, k_I, \text{Pub}(k_J)) \triangleq \\ & \text{new}(n). \\ & \text{out}(n). \\ & \text{in}(z). \\ & \text{decrypt } z \text{ as} \\ & \quad \{ n, z_A, I, x_A, y \}_{\text{Pub}(k_J)}. \\ & \text{decrypt } y \text{ as } \{y_A\}_{\text{Priv}(k_I)} \\ & \text{new}(m_x). \\ & \text{new}(m_y). \\ & \text{new}(n_B). \\ & \text{encrypt } \{ m_y \}_{\text{Pub}(k_J)} \text{ as } z_y. \\ & \text{run}(I, J, m_x). \\ & \text{encrypt } \{n_B, J, z_A, m_x, z_y\}_{\text{Priv}(k_I)} \text{ as } z_1. \\ & \text{out}(z_1). \\ & \text{in}(z_2). \\ & \text{decrypt } z_2 \text{ as } \{I, n_B\}_{\text{Pub}(k_J)}. \\ & \text{commit}(I, J, x_A) \end{aligned}$
---	---

e.g.,  $y_A$  and  $y_B$  of CCITT X.509 (3). discussed in Section 5.2). We are presently studying how to incorporate these (and other) aspects in our type and effect system.

We are also working in the direction of a comparison with the type and effect systems of [14, 15]. Indeed, those analyses are not equipped with type inference. We are studying an encoding between our type and effect system and the above mentioned ones, so that the tags inferred by our tool can be directly translated into types for [14, 15]. Another interesting issue is implementing those analyses directly in PEAR, possibly extending the common rule shape in a way that preserves the inference algorithm. See [7, 8] for a more detailed comparison with related literature.

In case a protocol cannot be validated, PEAR just tells the user where the validation fails without giving much information about how the protocol might be modified to fulfill the analysis requirements. It might be interesting to give hints to users in this direction. For example, by measuring the distance from matching a typing rule, the tool might give users a list of the minimal modifications to implement so that such a typing rule becomes applicable. We leave this as a future work.

The tool is available at <http://www.dsi.unive.it/~maffei/pear>.

*Acknowledgements.* We would like to thank the anonymous referees for their very helpful comments and suggestions.

## 7. REFERENCES

- [1] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 298(3):387–415, 2003.
- [2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
- [3] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 126–140. IEEE Computer Society Press, June 2003.
- [4] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Control flow analysis can find new flaws too. In *Proceedings of the Workshop on Issues on the Theory of Security (WITS'04)*, ENTCS. Elsevier, 2004.
- [5] M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings of ICALP 01*, volume 2076, pages 667–681. LNCS 2076, Springer Verlag, 2001.
- [6] M. Bugliesi, R. Focardi, and M. Maffei. Principles for entity authentication. In *Proceedings of 5th International Conference Perspectives of System Informatics (PSI 2003)*, volume 2890 of *Lecture Notes in Computer Science*, pages 294–307. Springer-Verlag, July 2003.
- [7] M. Bugliesi, R. Focardi, and M. Maffei. Compositional analysis of authentication protocols. In *Proceedings of European Symposium on Programming (ESOP 2004)*, volume 2986 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 2004.
- [8] M. Bugliesi, R. Focardi, and M. Maffei. Authenticity by tagging and typing. In *2nd ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code (FMSE 2004)*. ACM press, October 2004. Affiliated to CCS 2004.
- [9] M. Burrows, M. Abadi, and R. Needham. “A Logic of Authentication”. *Proceedings of the Royal Society of London*, 426(1871):233–271, 1989.
- [10] CCITT. The directory authentication framework. Draft Recommendation X.509, 1987. Version 7.
- [11] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. <http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz>, November 1997.
- [12] R. Focardi, R. Gorrieri, and F. Martinelli. Non interference

- for the analysis of cryptographic protocols. In *Proceedings of ICALP'00*, pages 354–372. Springer LNCS 1853, July 2000.
- [13] R. Focardi and M. Maffei.  $\rho$ -spi calculus at work: Authentication case studies. In *Proceedings of Mefisto Project, Formal Methods for Security and Time*, volume 99, pages 267–293. ENTCS, August 2004.
- [14] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *Proceedings of 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 145–159. IEEE Computer Society Press, June 2001.
- [15] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Proceedings of 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 77–91. IEEE Computer Society Press, 24-26 June 2002.
- [16] J. D. Guttman and F. J. Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, 2002.
- [17] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 255–268. IEEE Computer Society Press, July 2000.
- [18] ISO/IEC. *Information Technology-Security Techniques-Entity Authentication Mechanisms, Part 2:Entity Authentication using Symmetric Techniques*. 1993.
- [19] G. Lowe. “A Hierarchy of Authentication Specification”. In *Proceedings of the 10th Computer Security Foundation Workshop (CSFW'97)*, pages 31–44. IEEE Computer Society Press, 1997.
- [20] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur $\phi$ . In *Proceedings of the 1997 IEEE Symposium on Research in Security and Privacy*, pages 141–153. IEEE Computer Society Press, 1997.
- [21] M. Maffei. Tags for multi-protocol authentication. In *Proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SECCO '04)*. To appear. ENTCS, August 2004.
- [22] R. M. Needham and M. D. Schroeder. Authentication revisited. *ACM SIGOPS Operating Systems Review*, 21(1):7–7, 1987.
- [23] L. C. Paulson. Relations between secrets: Two formal analyses of the yahalom protocol. *Journal of Computer Security*, 9(3):197–216, 2001.
- [24] T. Woo and S. Lam. “A Semantic Model for Authentication Protocols”. In *Proceedings of 1993 IEEE Symposium on Security and Privacy*, pages 178–194, 1993.