In *Proceedings of DIMACS Workshop on Design and Formal Verification of Security Protocols*,
September 3-5, 1997 DIMACS Center, Rutgers University

Using Non Interference for the Analysis of Security Protocols

Riccardo Focardi[†] Anna Ghelli[‡] Roberto Gorrieri[‡]

[†]Dipartimento di Matematica Applicata ed Informatica,
Università Ca' Foscari di Venezia
Via Torino 155, I - 30173 Venezia Mestre (Italy)
e_mail: focardi@dsi.unive.it

[‡]Dipartimento di Scienze dell'Informazione,
Università di Bologna
Mura Anteo Zamboni 7, I - 40127 Bologna (Italy)
e_mail: {ghelli,gorrieri}@cs.unibo.it

Abstract

Non interference [GM82] was originally proposed as a means for analyzing the security of computer systems. In this paper we show that its theory can be profitably exploited also for the analysis of security protocols. As a simple case study, we consider the paradigmatic example of the Needham-Schroeder public-key protocol, that was already shown incorrect and fixed in [Low96]. As expected, our analysis discovers the same bug; what is really new is the methodology, based on the idea of proving the absence of information flow between the protocol partners and potential external attackers.

1 Introduction

Non-Interference (NI for short) [GM82] was introduced in order to circumvent difficulties in classic approaches to computer security. In order to limit, and possibly avoid, the damages produced by malicious programs (often called "Trojan Horses") which try to leak secret information, it was suggested to impose some access control rules which limit the action of these programs. However, it is often difficult to detect the so-called "covert channels", i.e., indirect ways for transmitting information with no explicit reading of confidential information. This kind of difficulties induced some researchers to re-think from scratch the whole problem of guaranteeing security in computer systems.

NI is indeed an alternative, more general approach to computer security. It requires to control directly the whole flow of information, rather than the accesses of subjects to objects (as in classical access control rules). In general, NI-like properties are based on a formal model of system behaviour and on a definition of information flow on such a model. By imposing some information flow rules, we can control both the direct and indirect transmission channels.

Although the basic motivation for NI properties was the analysis of information flows inside systems, the basic ideas behind them are very general and potentially applicable to other related security areas. The aim of this paper is precisely to show that the theory of NI properties can be profitably applied in typical problems of network security. As a simple case study, we consider the paradigmatic example of the Needham-Schroeder public-key protocol, that was already shown incorrect and fixed in [Low96]. As expected, our analysis discovers the same bug; what is really new is the methodology, based on the idea of proving the absence of information flow between the protocol partners and potential external attackers. This methodology is general and is not tailored for authentication protocols. In [FG] the same technique presented here is used to check a simple session key distribution protocol based on public-key (see Section 7 for more details on this issue).

The basic idea which permits to express authentication requirements using NI-based properties is very simple: we want to check if an enemy is able to interfere with the correct execution of the protocol; so, what we do is to check if the actions executed by the enemy interfere with the actions which characterize the successful authentication. As an example, in the attack reported in [Low96] we have that B is convinced to communicate with A but A has never sent an authentication request to B. This anomalous behaviour is caused by the enemy, which is thus able to interfere on two particular actions which we introduce to model the protocol behaviour: the first represents the authentication request from A to B (identified by the first message of the protocol) and the second represents the successful completion of the protocol by B (identified by the second message of the protocol). This enemy interference on the protocol can be detected by NI-based properties.

The method is general, since we do not give a definition of authentication. We simply need to select particular events that we want to observe and which represent the correct execution of the protocol. One typical choice is: one action representing the start of the protocol and two actions representing the correct termination of the protocol for the two partners.

The formal verification of these properties is fully automatized thanks to the CoSeC tool [FG, FG96] that accepts finite-state protocol specifications and check over them the NI-based information flow properties.

The paper is organized as follows. In Section 2 we present the model and the NI-based properties we use in the protocol specification and analysis. In Section 3 we recall the Needham-Schroeder public-key protocol and in Section 4 we specify it using our language. Section 5 is about the attacker specification. In Section 6 we analyze the protocol using NI-based properties. Finally, Section 7 is devoted to the comparison with related work.

2 The model

In this Section we introduce the model and the language we use for the specification of protocols and we also present the generalization of NI we use in the protocol analysis.

2.1 SPA and Semantic Equivalences

In the following, protocols will be specified using the *Security Process Algebra*, an extension of Milner's CCS [Mil89]. SPA has one additional operator, namely the hiding operator E/L of CSP [Hoa85], which is useful in characterizing some security properties in an algebraic style. Moreover the set of visible actions is partitioned into high and low level actions in order to specify multilevel systems.

SPA syntax is based on the following elements: a set $I = \{a, b, \ldots\}$ of *input* actions, a set $O = \{\bar{a}, \bar{b}, \ldots\}$ of *output* actions, a set $\mathcal{L} = I \cup O$ of *visible* actions, ranged over by α, and the usual function $\bar{\cdot} : \mathcal{L} \to \mathcal{L}$ such that $a \in I \Longrightarrow \bar{a} \in O$ and $\bar{a} \in O \Longrightarrow \bar{\bar{a}} = a \in I$; two sets Act_H and Act_L of high and low level actions such that $\overline{Act_H} = Act_H$, $\overline{Act_L} = Act_L$, $Act_H \cup Act_L = \mathcal{L}$ and $Act_H \cap Act_L = \emptyset$ where $\bar{L} \stackrel{\text{def}}{=} \{\bar{a} : a \in L\}$; a set $Act = \mathcal{L} \cup \{\tau\}$ of actions (τ is the internal, invisible action), ranged over by μ; a set K of constants, ranged over by Z. The syntax of SPA *agents* is defined as follows:

$$E \quad ::= \quad \underline{0} \mid \mu.E \mid E + E \mid E|E \mid E \setminus L \mid E/L \mid E[f] \mid Z$$

where $L \subseteq \mathcal{L}$ and $f : Act \to Act$ is such that $f(\bar{\alpha}) = \overline{f(\alpha)}$, $f(\tau) = \tau$. Moreover, for every constant Z there must be the corresponding definition: $Z \stackrel{\text{def}}{=} E$. The meaning of $\underline{0}$, $\mu.E$, $E+E$, $E|E$, $E \setminus L$, $E[f]$ and $Z \stackrel{\text{def}}{=} E$ is as for CCS [Mil89]. Intuitively, $\underline{0}$ is the empty process, which cannot do any action; $\mu.E$ can do an action μ and then behaves like E; $E_1 + E_2$ can alternatively choose [1] to behave like E_1 or E_2; $E_1|E_2$ is the parallel composition of E_1 and E_2, where the executions of the two systems are interleaved, possibly synchronized on complementary input/output actions, producing an internal τ. $E \setminus L$ can execute all the actions E is able to do, provided that they do not belong to $L \cup \bar{L}$; E/L turns all the actions in L into internal τ's; if E can execute action μ, then $E[f]$ performs $f(\mu)$; finally, Z does what E does, when $Z \stackrel{\text{def}}{=} E$.

Let $\mathcal{E}$ be the set of SPA agents, ranged over by E, F. Let $\mathcal{L}(E)$ denote the *sort* of E, i.e., the set of the (possibly executable) actions occurring syntactically in E. The sets of high level agents and low level ones are defined as $\mathcal{E}_H \stackrel{\text{def}}{=} \{E \in \mathcal{E} \mid \mathcal{L}(E) \subseteq Act_H \cup \{\tau\}\}$ and $\mathcal{E}_L \stackrel{\text{def}}{=} \{E \in \mathcal{E} \mid \mathcal{L}(E) \subseteq Act_L \cup \{\tau\}\}$, respectively. The operational semantics of SPA is given (as usual) associating to each agent a particular state of the labelled transition system $(\mathcal{E}, Act, \to)$ where $\to \subseteq \mathcal{E} \times Act \times \mathcal{E}$ and, intuitively, $E \stackrel{\mu}{\to} E'$ means that agent E can execute μ moving to E' (see [FG95, FG] for more details).

In the following the expression $E \stackrel{\mu}{\Longrightarrow} E'$ is a shorthand for $E(\stackrel{\tau}{\to})^* E_1 \stackrel{\mu}{\to} E_2 (\stackrel{\tau}{\to})^* E'$, where $(\stackrel{\tau}{\to})^*$ denotes a (possibly empty) sequence of τ labelled transitions. We also extend the '$\Longrightarrow$' notation to sequences of actions; $E \stackrel{\gamma}{\Longrightarrow} E'$ with $\gamma \in \mathcal{L}^+, \gamma = \gamma_1\gamma_2 \ldots \gamma_n$ means that $\exists E_1, E_2, \ldots, E_{n-1}$ such that $E \stackrel{\gamma_1}{\Longrightarrow} E_1 \stackrel{\gamma_2}{\Longrightarrow} \ldots \stackrel{\gamma_{n-1}}{\Longrightarrow} E_{n-1} \stackrel{\gamma_n}{\Longrightarrow} E'$. For the empty sequence ϵ we have that $E \stackrel{\epsilon}{\Longrightarrow} E'$ stands for $E(\stackrel{\tau}{\to})^* E'$.

We recall here the definition of trace equivalence

Definition 2.1 *A trace of a process E is a sequence of actions that E can execute. The set of traces is defined as follows: $T(E) \stackrel{\text{def}}{=} \{\gamma \in \mathcal{L}^* \mid \exists E' : E \stackrel{\gamma}{\Longrightarrow} E'\}$. E and F are* trace equivalent *(notation $E \approx_T F$) if and only if $T(E) = T(F)$.* ■

2.2 Value-Passing SPA

In this section we briefly present a value-passing extension of "pure" SPA (VSPA, for short). In all the examples reported in this paper will use such value passing calculus, because it leads to more readable specifications than those written in pure SPA.

As an example, consider the following buffer cell [Mil89]:

$$\begin{aligned} C &\stackrel{\text{def}}{=} in(x).C'(x) \\ C'(x) &\stackrel{\text{def}}{=} \overline{out}(x).C \end{aligned}$$

where x is a variable that can assume values in $\mathbb{N}$ (we usually write $x \in \mathbb{N}$). C reads a natural value n through action in and stores it in variable x. Then this value is passed to agent C' which can give n as output through action out moving again to C. So C represents a buffer which may hold a single data item.

Now we show how C can be translated into an SPA agent. The parametrized constant C' becomes a family of constants C'_v one for each value $v \in \mathbb{N}$. Similarly $\overline{out}(x)$ becomes a family $\overline{out}_v$

[1]For notational convenience, we use sometimes the $\sum$ operator to represent a general n-ary (or even infinitary) sum operator.

of prefixes. So the single definition for C' becomes the following family of definitions:

$$C'_v \stackrel{\rm def}{=} \overline{out}_v.C \quad (v \in \mathbb{N})$$

Now consider the prefix $in(x)$. To reflect the fact that it can accept any input value, binding variable x, we translate it into $\sum_{v\in\mathbb{N}} in_v$. So the definition becomes:

$$C \stackrel{\rm def}{=} \sum_{v\in\mathbb{N}} in_v.C'_v$$

VSPA is very similar to the value-passing CCS introduced in [Mil89]. The main difference is that in VSPA we can have more than one parameter for actions and parameters are multi-sorted. The syntax of VSPA agents is defined as follows:

$$\begin{array}{rl} E ::= & \underline{0} \;\big|\; a(x_1,\ldots,x_n).E \;\big|\; \bar{a}(e_1,\ldots,e_n).E \;\big|\; \tau.E \;\big|\; E+E \;\big|\; E|E \;\big|\; E\setminus L \;\big| \\ & \big|\; E/L \;\big|\; E[f] \;\big|\; A(e'_1,\ldots,e'_n) \;\big|\; \textbf{if } b \textbf{ then } E \;\big|\; \textbf{if } b \textbf{ then } E \textbf{ else } E \end{array}$$

where the variables $x_1,\ldots,x_n$, the value expressions $e_1,\ldots,e_n$ and $e'_1,\ldots,e'_n$ must be consistent with the arity of the action a and constant A, respectively (the arity specifies the sorts of the parameters), and b is a boolean expression.

It is also necessary to define constants as follows: $A(x_1,\ldots,x_m) \stackrel{\rm def}{=} E$ where E is a VSPA agent which may contain no free variables except $x_1,\ldots,x_m$, which must be distinct. As in [Mil89] the semantics of the value-passing calculus is given as a translation into the pure calculus. The translations rests upon the idea that a single label a of VSPA, with n parameters with sorts $S_1\ldots S_n$, becomes the set of labels $\{a_{v_1\ldots v_n} : v_i \in S_i, \forall i \in [1,n]\}$ in SPA. We consider only agents without free variables because if an agent has a free variable then it becomes a family of agents, one for each value of the variable. The translation can be given inductively on the structure of agents (see [Mil89] for more details about this kind of translation).

Note that we do not partition the set of actions into two levels; we directly refer to the partition in the pure calculus. In this way it is possible for a certain action in VSPA to correspond, in the translation, to actions at different levels in SPA. This can be useful if we want a parameter representing the level of a certain action.

2.3 NI generalization

In this section we present the *Non Deducibility on Compositions* (*NDC*, for short) property, proposed in [FG95] as a generalization of NI in a process algebraic setting. It is simply based on the idea of checking the system against all high level potential interactions. A system E is *NDC* if for every high level process Π a low level user cannot distinguish E from $(E|\Pi)\setminus Act_H$. In other words, a system E is *NDC* if what a low level user sees of the system is not modified by composing any high level process Π to E.

Definition 2.2 *E is NDC if and only if* $\forall \Pi \in \mathcal{E}_H$, $E/Act_H \approx_T (E \mid \Pi)\setminus Act_H$. ∎

This definition of *NDC* is difficult to use in practice, because of the universal quantification on high level processes. We need an alternative formulation of *BNDC* which avoids universal quantification, exploiting local information only. It is possible to prove [FG95] that *NDC* is equivalent to *Strong Non-deterministic Non Interference* (*SNNI*, for short) which is defined as follows:

Definition 2.3 $E \in SNNI \iff E \setminus Act_H \approx_T E/Act_H$. ∎

This property is a natural generalization of NI to non-deterministic systems. Intuitively, the high level does not interfere with the low level if the effects of high level actions are not visible by a low level user. In order to check this, we consider every trace γ of the system containing some high level actions. Then, we look if there exists another trace γ' with the same subsequence of low level actions and without any high actions. A low level user, which can only observe low level actions, is not able to distinguish γ and γ'. As both γ and γ' are legal traces, we can conclude that the possible execution of the high level actions in γ has no effect on the low level view of the system. As a matter of fact, in E/Act_H all the high level actions are hidden, hence giving the low level view of the system; $E \setminus Act_H$ instead removes all the traces containing high level actions. So, if the two terms are equivalent, then for every trace with high level actions we can find another trace without such actions but with the same subsequence of low level actions.

We have the following result [FG95]:

Proposition 2.4 $NDC = SNNI$ ■

It is interesting to observe that this characterization of NDC through the $SNNI$ property is possible since we are using an equivalence notion which is not able to detect deadlocks. In [FG95, FG] it is shown that under finer equivalence notions (testing, bisimulation) this result does not hold and a good (i.e., easy to check) characterization of NDC is still to be found.

3 The Needham-Schroeder public key protocol

The aim of the Needham-Schroeder protocol [NS78] is to establish mutual authentication between two users A and B. It uses public key cryptography.

We will denote with PK_U and SK_U the public and secret keys for a generic user U, respectively. The protocol uses *nonces* which are random numbers generated with the purpose of being used in a *single* run of the protocol. In general, N_A will represent a nonce generated by A.

The protocol starts with A consulting the authentication server in order to obtain B's public key:

$$A \rightarrow AS : A, B$$

where the notation $Usr_1 \rightarrow Usr_2 : Msg$ means that Usr_1 is sending the message Msg to Usr_2.

AS replies with:

$$AS \rightarrow A : SK_{AS}(PK_B, B)$$

where SK_{AS} is the authentication server secret key. So the AS is signing the public key of B and we assume that A knows the AS's public key PK_{AS}, which can be used to decrypt the message.

The next step is to initiate the communication with B:

$$A \rightarrow B : PK_B(N_A, A)$$

Since the message is encrypted with B's public key, then only B can decrypt it. The message means that someone who claims to be A wishes to establish communication; N_A is the nonce generated by A.

B decrypts the message and, similarly to A, asks for A's public key to AS:

$$\begin{array}{lcll} B & \rightarrow & AS & : B, A \\ AS & \rightarrow & B & : SK_{AS}(PK_A, A) \end{array}$$

At this point a double handshake is needed to authenticate A and B one another:

$$\begin{array}{lcll} B & \rightarrow & A & : & PK_A(N_A, N_B) \\ A & \rightarrow & B & : & PK_B(N_B) \end{array}$$

In the first message B replies to A, sending the new nonce N_B and the one received from A (N_A), both encrypted with A's public key. When A receives the nonce N_A back, (s)he can conclude that is really talking with B since only B could have decrypted the message sent by A containing N_A. In the second message A replies to B, sending back the N_B nonce. Following the same reasoning as above, B will conclude that is indeed talking with A.

The protocol is composed by seven steps, but four of them can be avoided if A and B have local caches of commonly used public key.

4 Protocol specification in SPA

In this section we present a formalization of the Needham-Schroeder protocol using VSPA. Here we consider the reduced version of the protocol, omitting the public key requests by A and B. The formalization we give is very similar to the one in [Low96]. The main difference is in the language used (the CSP in that case). We are not interested in giving a new formalization of the protocol and of the enemy. The aim of this paper is to show a new technique to analyze the protocol based on NI-like properties. So the main difference with respect to [Low96] work can be found in the protocol analysis section (i.e., Section 6).

The reduced protocol is composed by the following three steps:

$$\begin{array}{lcccl} Message\ 1 & A & \rightarrow & B: & PK_B(N_A, A) \\ Message\ 2 & B & \rightarrow & A: & PK_A(N_A, N_B) \\ Message\ 3 & A & \rightarrow & B: & PK_B(N_B) \end{array}$$

The protocol is modeled through two VSPA agents: the *Initiator* and the *Responder*. The *Initiator* accepts a user connection request, then starts the messages exchange with the *Responder*.

Protocol messages are all of the form:

$$msg\_i(mitt, dest, enc(key, p_1, p_2, \ldots, p_n))$$

where:

i is the message number (it ranges in $[1 \ldots 3]$);

$mitt$ is the identifier of the message sender;

$dest$ is the identifier of the message addressee;

$enc(\ldots)$ is a generic public key encryption algorithm[2];

key is an encryption key;

$p_1, p_2, \ldots, p_n$ represents the information tuple conveyed by the (encrypted) message.

[2] We are not interested in analyzing possible attacks to this algorithm, so we do not specify it in detail.

Besides protocol messages, there are some actions to allow interaction between users and the protocol.

A user *a_id* can make a connection request to *b_id* through action *request(a_id,b_id)*. To inform users that the connection is established, the *Initiator* and the *Responder* give as output *connection(a_id,b_id,ok)* and *r_connection(b_id,a_id,ok)*, respectively. In the case of a protocol failure the messages become: *connection(a_id,b_id,no)* and *r_connection(b_id,a_id,no)*, respectively. Another action is executed by *Responder* to inform users that someone has made a connection request: *r_running(a_id,b_id)*.

Initiator is defined using VSPA by:

$$
\begin{array}{lll}
Initiator(a\_id, a\_nonce) & \stackrel{\text{def}}{=} & request(a\_id, dest\_id). \\
& & \overline{msg\_1}(a\_id, dest\_id, enc(PK_{dest\_id}, a\_nonce, a\_id)). \\
& & msg\_2(dest\_id, a\_id, enc(PK_{a\_id}, a\_nonce', dest\_nonce)). \\
& & \ \textbf{if}\ \ a\_nonce' = a\_nonce\ \ \textbf{then} \\
& & \quad \overline{msg\_3}(a\_id, dest\_id, enc(PK_{dest\_id}, dest\_nonce)). \\
& & \quad \overline{connection}(a\_id, dest\_id, ok).0 \\
& & \textbf{else} \\
& & \quad \overline{connection}(a\_id, dest\_id, no).0
\end{array}
$$

Initiator receives, through action *request*$(a\_id, dest\_id)$, a request from user *a_id* for a connection with user *dest_id*; then it starts a run of the protocol. It sends message 1 containing the sender identifier and its nonce, both encrypted with receiver's public key. When it receives message 2 containing the two nonces $a\_nonce', dest\_nonce$ encrypted with the *Initiator* public key, it checks if a_nonce' is equal to the nonce sent in the first message (*a_nonce*). If this is the case, it sends the other nonce *dest_nonce* to user *dest_id* and informs the user *a_id* that the connection with *dest_id* is established (action *connection(a_id,dest_id,ok)*). If the nonce a_nonce' is different from *a_nonce*, the protocol fails and this is notified through action $\overline{connection}(a\_id, dest\_id, no)$.

Similarly to *Initiator*, the *Responder* is defined in VSPA as follows:

$$
\begin{array}{lll}
Responder(b\_id, b\_nonce) & \stackrel{\text{def}}{=} & msg\_1(send\_id, b\_id, enc(PK_{b\_id}, send\_nonce, send\_id')). \\
& & \textbf{if}\ \ send\_id = send\_id'\ \ \textbf{then} \\
& & \quad \overline{r\_running}(send\_id, b\_id). \\
& & \quad \overline{msg\_2}(b\_id, send\_id, enc(PK_{send\_id}, send\_nonce, b\_nonce)). \\
& & \quad msg\_3(send\_id, b\_id, enc(PK_{b\_id}, b\_nonce')). \\
& & \quad \textbf{if}\ \ b\_nonce = b\_nonce'\ \textbf{then} \\
& & \qquad \overline{r\_connection}(b\_id, send\_id, ok).0 \\
& & \quad \textbf{else} \\
& & \qquad \overline{r\_connection}(b\_id, send\_id, no).0 \\
& & \textbf{else} \\
& & \quad \overline{r\_connection}(b\_id, send\_id, no).0
\end{array}
$$

Responder receives message 1, containing sender's nonce and identity *(send_nonce,send_id')*. It checks if the identifier contained in the plaintext part of the message (*send_id*) is equal to that contained in the ciphertext part. If it is so, it sends message 2 containing the nonce received and its own nonce, encrypted with the sender public key. It receives message 3 and checks if the nonce received is equal to its own; if the check is successful it informs, through the action $\overline{r\_connection}(b\_id, send\_id, ok)$, the user that the connection is established, otherwise it gives as output $\overline{r\_connection}(b\_id, send\_id, no)$ which represent an unsuccessful termination of the protocol. The complete protocol specification is defined as parallel composition of the agents described above:

$$Protocol \stackrel{\text{def}}{=} (Initiator(a, na) \mid Responder(b, nb)) \setminus L$$

where $L = \{msg\_1, msg\_2, msg\_3\}$ is the set of messages sent during the protocol.

5 The Attacker

In order to analyze the protocol it is necessary to study its behaviour when it is attacked by a malicious user. This intruder can be a normal user of the computer network that has decided to assume an incorrect behaviour. In particular we want to model a typical "man-in-the-middle" attack. So the intruder is able to:

- intercept any sent message;
- hold or forward any intercepted messages, possibly changing plaintext parts, even if he does not understand the contents of the encrypted part;
- decrypt messages encrypted with his own public key; we assume that the encryption algorithm used is secure, so the enemy can decrypt only the messages which are encrypted with his own public key;
- introduce new messages into the system (using informations he knows).

In order to simplify the specification we do not consider the possibility of storing intercepted information for later uses. In particular the enemy is able to use only the information gained in the last intercepted message when (s)he introduces new messages in the system. This is different from what is done in [Low96] where the enemy has a local memory where (s)he can store stolen information and reuse it later. We could define this more complex enemy also in our model but we prefer to use the simpler version with no memory which is sufficient for this analysis.

First, we present the enemy activity for message 1 and then we will see the whole *Enemy* agent. We have inserted C-like comments in order to better explain what is going on.

$Enemy\_1(e\_id, e\_nonce) \stackrel{\text{def}}{=}$
  `/* the Enemy intercepts msg_1 */`
  $msg\_1(id1, id2, enc(PK_{id2}, nonce, id1)).$
  (
   `/* The Enemy can decide to do nothing with the intercepted message: */`
   $Enemy\_1(e\_id, e\_nonce)$
   $+$
   `/* or it can forward the message possibly changing some parts */`
   if $PK_{id2} = PK_{e\_id}$ **then**
    `/* The message is encrypted with the Enemy public key so it can reuse the ecrypted information */`

$$\sum_{\substack{ini\_id, id' \in I_{id} \\ res\_id \in R_{id} \\ nonce' \in \{nonce, e\_nonce\}}} \overline{msg\_1}(ini\_id, res\_id, enc(PK_{res\_id}, nonce', id')).Enemy\_1(e\_id, e\_nonce)$$

   else
    `/* The Enemy is not able to decrypt the message: it can only change the plaintext part */`

$$\sum_{\substack{ini\_id \in I_{id} \\ res\_id \in R_{id}}} \overline{msg\_1}(ini\_id, res\_id, enc(PK_{id2}, nonce, id1)).Enemy\_1(e\_id, e\_nonce)$$

)

where I_{id} and R_{id} are the sets of all the possible initiators and responder identifiers, respectively. We have an analogous situation for messages 2 and 3. So the VSPA specification for the *Enemy* is given by the following agent:

$$
\begin{array}{l}
Enemy(e\_id, e\_nonce) \stackrel{\text{def}}{=} \\
\quad fake.(\\
\quad\quad msg\_1(id1, id2, enc(PK_{id2}, nonce, id1)).(\\
\quad\quad\quad Enemy(e\_id, e\_nonce) \ + \\
\quad\quad\quad \textbf{if} \ \ PK_{id2} = PK_{e\_id} \ \ \textbf{then} \\
\quad\quad\quad\quad \sum_{\substack{ini\_id, id' \in I_{id} \\ res\_id \in R_{id} \\ nonce' \in \{nonce, e\_nonce\}}} \overline{msg\_1}(ini\_id, res\_id, enc(PK_{res\_id}, nonce', id')).Enemy(e\_id, e\_nonce) \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \sum_{\substack{ini\_id \in I_{id} \\ res\_id \in R_{id}}} \overline{msg\_1}(ini\_id, res\_id, enc(PK_{id2}, nonce, id1)).Enemy(e\_id, e\_nonce) \\
\quad\quad) \\
\quad\quad + \\
\quad\quad msg\_2(id1, id2, enc(PK_{id2}, nonce1, nonce2)).(\\
\quad\quad\quad Enemy(e\_id, e\_nonce) \ + \\
\quad\quad\quad \textbf{if} \ \ PK_{id2} = PK_{e\_id} \ \ \textbf{then} \\
\quad\quad\quad\quad \sum_{\substack{ini\_id \in I_{id} \\ res\_id \in R_{id} \\ any\_nonce1 \in \{nonce1, nonce2, e\_nonce\} \\ any\_nonce2 \in \{nonce1, nonce2, e\_nonce\}}} \overline{msg\_2}(res\_id, ini\_id, enc(PK_{ini\_id}, any\_nonce1, any\_nonce2)). \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Enemy(e\_id, e\_nonce) \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \sum_{\substack{ini\_id \in I_{id} \\ res\_id \in R_{id}}} \overline{msg\_2}(ini\_id, res\_id, enc(PK_{id2}, nonce1, nonce2)).Enemy(e\_id, e\_nonce) \\
\quad\quad) \\
\quad\quad + \\
\quad\quad msg\_3(id1, id2, enc(PK_{id2}, nonce)).(\\
\quad\quad\quad Enemy(e\_id, e\_nonce) \ + \\
\quad\quad\quad \textbf{if} \ \ PK_{id2} = PK_{e\_id} \ \ \textbf{then} \\
\quad\quad\quad\quad \sum_{\substack{ini\_id \in I_{id} \\ res\_id \in R_{id} \\ res\_nonce \in \{nonce, e\_nonce\}}} \overline{msg\_3}(ini\_id, res\_id, enc(PK_{res\_id}, res\_nonce)).Enemy(e\_id, e\_nonce) \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \sum_{\substack{ini\_id \in I_{id} \\ res\_id \in R_{id}}} \overline{msg\_3}(ini\_id, res\_id, enc(PK_{id2}, nonce)).Enemy(e\_id, e\_nonce) \\
\quad\quad) \\
\quad)
\end{array}
$$

The aim of the action $fake$ will be explained later. As for the other agents, the *Enemy* parameters are the intruder's identifier and nonce.

6 Protocol Analysis

In order to test the protocol correctness we consider the parallel composition of all the agents described so far. As we are considering the network as insecure, the right way to compose the *Enemy* with the partners of the protocol is exactly on the communication channel between the *Initiator* and the *Responder*. This means that the *Enemy* is composed in parallel with them inside the scope of the restriction operator. [3]

We want to reduce the problem of testing if the protocol correctly authenticates the two users to a problem of information flow control between two levels. For this reason the process actions have been divided into high and low level actions (sets Act_H and Act_L). Since we want to check the influence that an intruder can have on the protocol execution, Act_H is composed by the action *fake*, which is executed at every *Enemy* iteration and so can be seen as an abstraction of the *Enemy* activity. All the other actions belong to the low level.

$$Protocol \stackrel{\text{def}}{=} (Initiator(a\_id, na) \mid Responder(b\_id, nb) \mid Enemy(e, ne)) \setminus L$$

where:

- $L = \{msg\_1,\ msg\_2,\ msg\_3\}$
- $Act_H = \{fake\}$
- $Act_L = \{msg\_1,\ msg\_2,\ msg\_3,\ request,\ r\_running,\ connection,\ r\_connection\}$

Since we want to focus on the "correct" execution of the protocol, we consider, for the analysis, only the actions representing the start and the correct termination of a session between a_{id} and b_{id}:

$$request(a\_id, b\_id),\ connection(a\_id, b\_id, ok),\ r\_connection(b\_id, a\_id, ok).$$

All the other actions are hidden. So, we define process $Protocol'$ as follows:

$$\begin{aligned} Protocol' \stackrel{\text{def}}{=} Protocol/\{\ & r\_running, request(a\_id, e\_id), connection(a\_id, e\_id, ok), \\ & connection(any\_id, any\_id', no), r\_connection(b\_id, e\_id, ok), \\ & r\_connection(any\_id, any\_id', no)\ \} \end{aligned}$$

The security property we automatically check is *SNNI*, which is equal to *NDC*. The tool used for this analysis is the *Compositional Security Checker* (CoSeC), a semantic-based tool for the automatic verification of some compositional information flow properties [FG96, FG].

In particular, using CoSeC, we check if the following holds:

$$Protocol' \setminus fake \approx_T Protocol'/fake$$

As we said above, the action $fake$ represents the *Enemy* activity. So we can see $Protocol' \setminus fake$ as the protocol on a completely secure channel (i.e., with no enemies), and $Protocol'/fake$ as the protocol under the *Enemy* attack. If these two processes are not equivalent we can conclude that the *Enemy* is able to interfere with the correct execution of the protocol.

The automatic check gives the following answer:

[3] The semantics of the restriction operator on a channel name a ensures that such a channel is "perfectly" secure, i.e., no other process outside its scope can communicate through channel a with processes syntactically located inside its scope.

```
false
Agent Protocol'/fake
can perform action sequence 'r_connection_b_a_ok
which agent Protocol'\fake
cannot
```

where action $'r\_connection\_b\_a\_ok$ corresponds to $r\_connection(b\_id, a\_id, ok)$ in VSPA. This answer means that the high level action $fake$ (and so the enemy) interferes with the low level actions, i.e., with the correct execution of the protocol. In particular we see that when we hide the fake action, and so we permit the enemy to operate indiscriminately, we obtain an execution sequence where B is convinced to communicate to A (action $'r\_connection\_b\_a\_ok$) but A has never started the protocol with B. Since when we restrict action $fake$ (we do not allow any activity of the $Enemy$) we do not have this particular execution of the protocol, we can conclude that the enemy is able to force an anomalous behaviour of the protocol. If we study the labelled transition system generated by the initial process $Protocol$ looking for a trace with a $r\_connection(b\_id, a\_id, ok)$ and without a $request(a\_id, b\_id)$, we find out the following sequence corresponding to a specific attack:

$$request(a\_id, e\_id), r\_running(a\_id, b\_id), connection(a\_id, e\_id, ok), r\_connection(b\_id, a\_id, ok)$$

This means that user B is convinced to have established a connection with user A, while A has made a request to user E: hence, the enemy has cheated B.

To realize the cheat, two runs have to be active at the same time: one between the users A and E, and one between the users E (the intruder that claims to be A) and B. Using α and β to indicate the two runs, the sequence that allows to cheat is:

$$\begin{array}{lll}
Message\ \alpha 1. & A \rightarrow E & : \ PK_E(N_A, A) \\
Message\ \beta 1. & E(A) \rightarrow B & : \ PK_B(N_A, A) \\
Message\ \beta 2. & B \rightarrow E(A) & : \ PK_A(N_A, N_B) \\
Message\ \alpha 2. & E \rightarrow A & : \ PK_A(N_A, N_B) \\
Message\ \alpha 3. & A \rightarrow E & : \ PK_E(N_B) \\
Message\ \beta 3. & E(A) \rightarrow B & : \ PK_B(N_B)
\end{array}$$

In message $\alpha 1$, A sends his nonce and identifier to E, trying to establish a connection with him. When E receives this message, (s)he forwards it to B, pretending to be A ($\beta 1$). B accepts the message and sends back his nonce and the nonce just received ($\beta 2$). E intercepts this message and, since he cannot decrypt it, he uses A as an oracle forwarding to him the massage ($\alpha 2$). A accepts and decrypts it; then he sends to E the last message containing N_B ($\alpha 3$), considering established the session with E. In this way E has learn B's nonce: then he can send it back to B ($\beta 3$), that thinks to have just established a session with A.

The problem is in message 2 of the protocol that contains the two nonces. Since there is no identifier in it, anyone can intercepts and forwards it claiming to be anyone else.

The correct version of the protocol is:

$$\begin{array}{lll}
Message\ 1. & A \rightarrow B & : \ PK_B(N_A, A) \\
Message\ 2. & B \rightarrow A & : \ PK_A(N_A, N_B, B) \\
Message\ 3. & A \rightarrow B & : \ PK_B(N_B)
\end{array}$$

Where message 2 now contains also the sender identifier.

Using again SC to analyze the corrected formalization, we can see that now the agent $Protocol'$ satisfies the SNNI property, so we can conclude that $Enemy$ is not able to interfere with the protocol.

The formalization used so far presents some limitations. For example, only one session can be established. In [Ghe97] it has been studied an improved model which allows the execution of multiple sessions. Another possible extension of the model is related to the equivalence notion used in the analysis. Trace equivalence is not able to detect deadlocks. In [Ghe97] this protocol has been analyzed also with respect to a stronger equivalence notion (bisimulation equivalence) which is able to detect the presence of deadlocks.

7 Related work

The Needham-Schroeder public key protocol has been considered in many papers as an example to illustrate protocol analysis techniques and tools.

In [Low96] it is presented an analysis method based on the FDR tool and CSP [Hoa85] language. The FDR tool takes as input a CSP specification and a CSP implementation and verifies if the latter refines the former. In order to test if the protocol achieves mutual authentication, the author splits the problem in two: (i) test if the protocol authenticates the responder; (ii) test if the protocol authenticates the initiator.

In our model we can make an analogous check: it is sufficient to test if the security property SNNI is satisfied by the process *Protocol* where the visible actions are composed by the pair:

(i) *r_running(a_id,b_id), connection(a_id,b_id,ok)* for the responder authentication;

(ii) *request(a_id,b_id), r_connection(b_id, a_id,ok)* for t he initiator authentication.

As in [Low96] we obtain that the protocol fails in the initiator authentication, since we can have $r\_connection(b\_id, a\_id, ok)$ without $request(a\_id, b\_id)$. But unlike [Low96], our model allows to consider more than two actions at the same time. In this way it is possible to test if the protocol achieves mutual authentication with only one check.

Moreover we observe that the (i) check is not a sufficient condition to responder authentication. In fact, the execution of $r\_running(a\_id, b\_id)$ can be obtained by any enemy which pretends to be B in the first message (note that if this would not be possible we would have authentication of A before the execution of the protocol, which makes no sense). So, we could have an enemy which successfully pretends to be B (action $connection(a\_id, b\_id, ok)$) and also forces B executing $r\_running(a\_id, b\_id)$, satisfying in this way the (i) condition.

Another difference is that our method is based on a NI-like property. In this way we can reuse all the research in that field. Moreover the property we check is fixed in advance, so we do not need to define a specific property for every different protocol we want to verify which is, in general, a non trivial task as we have seen above for property (i). We can also see that the method is not tailored for authentication. In [FG], we show how to use exactly the same NI-based property and the same kind of analysis to check a simple session key distribution protocol based on public-key. In particular, in that paper we show how NI-based properties are able to detect the following two attacks:

- if the KDC (Key Distribution Center) does not sign the requested public keys, then an attacker can substitute those keys with its own key;
- if the KDC forgets to include in the signed message the identity of the owner of the key, then an attacker can fake the request from a user to KDC in such a way that the user receives the enemy key.

References

[FG] R. Focardi and R. Gorrieri. "The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties". To appear in *IEEE Transactions on Software Engineering*. A preliminary version can be found in http://www.cs.unibo.it/ focardi/www/ps/Tool.ps.gz.

[FG95] R. Focardi and R. Gorrieri. "A Classification of Security Properties for Process Algebras". *Journal of Computer Security*, 3(1):5–33, 1994/1995.

[FG96] R. Focardi and R. Gorrieri. "Automatic Compositional Verification of Some Security Properties". In *Proceedings of Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 167–186, Passau (Germany), March 1996. Springer-Verlag, LNCS 1055.

[Ghe97] A. Ghelli. "Analisi di un Protocollo di Autenticazione con il Security Checker". Master's thesis, Università degli Studi di Bologna, March 1997.

[GM82] J. A. Goguen and J. Meseguer. "Security Policy and Security Models". In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Low96] G. Lowe. "Breaking and Fixing the Needham-Schroeder Public-key Protocol using FDR". In *Proceedings of Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 146–166, Passau (Germany), March 1996. Springer-Verlag, LNCS 1055.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[NS78] R. Needham and M. Schroeder. "Using encryption for authentication in large networks of computers". *Communications of the ACM*, 21(12):993–999, 1978.