CVS: A Compiler for the Analysis of Cryptographic Protocols

Antonio Durante[1] Riccardo Focardi[2] Roberto Gorrieri[1]

[1]Dipartimento di Scienze dell'Informazione,
Università di Bologna
Mura Anteo Zamboni 7, 40127 Bologna, Italy
e-mail: {durante,gorrieri}@cs.unibo.it

[2]Dipartimento di Informatica,
Università Ca' Foscari di Venezia
Via Torino 155, I - 30173 Venezia Mestre (Italy)
e-mail : focardi@dsi.unive.it

Abstract

The Security Process Algebra (SPA) is a CCS-like specification language where actions belong to two different levels of confidentiality. It has been used to define several non-interference-like security properties whose verification has been automatized by means of the tool CoSeC. In recent years, a method for analyzing security protocols using SPA and CoSeC has been developed. Even if it has been useful in analyzing small security protocols, this method has shown to be error-prone as it requires the description by hand of the protocol and of the environment in which it will execute. This problem has been solved by defining a protocol specification language more abstract than SPA, called VSP, and a compiler CVS that generates in an automatic way the SPA specification for a given protocol described in VSP. The VSP/CVS technology is very powerful and its usefulness is shown with the case-study of the Woo-Lam one-way authentication protocol, for which an attack undocumented in the literature is found.

1 Premise

Many security protocols, using cryptographic techniques, have been proposed in the literature to achieve several goals, e.g. authentication. They are supposed to succeed even in presence of malicious agents, sometimes called intruders or enemies, that can interfere with their correct execution. Unfortunately, many of them may fail when an intruder intercepts some messages and exploits the information they contain in order to introduce new messages into the network.

The approach based on the verification of Non-Interference properties (NI for short), used for analyzing computer security, has been recently applied in [4, 3, 5] to find attacks upon some cryptographic protocols, such as the Needham-Schroeder public key authentication protocol [13] and the ISO SC27 [7]. Analyzing a security protocol using NI consists of the following steps. We specify the whole system (protocol and enemy) in the Security Process Algebra (SPA [4]). We then define the observable actions, i.e. those actions that are relevant for security and that can reveal if the enemy has interfered with the correct execution of the protocol. The tool CoSeC [4] takes as input the SPA specification that represents the system that we want to analyze and checks a particular NI property, called SNNI, on that process. If the system verifies SNNI, then we are sure that the enemy cannot interfere with the correct execution of the protocol in the analyzed session; otherwise we observe some enemy interferences, i.e., execution traces that were not possible without enemy's intervention. These extra-traces may reveal some possible attacks to the protocol.

Hence, we do not specify any specific property that the protocol must satisfy; we only observe if the intruder is able to modify (in some way) the flow of information among the honest participants. The task of understanding what are the interferences that correspond to attacks is not hard; the real problem is to specify the system (protocol and enemy) in SPA. In this paper we show the solution to this problem and how to apply the method based on NI verification in an automatic way.

2 Making fully automatic the approach

We have briefly described how to analyze a protocol using NI according to [5]. It is important to note that if we want to analyze a protocol, we have to fix in advance the *context of analysis*, i.e. the session. Indeed, what we analyze is actually one particular session at a time. As a user can be potentially engaged in a infinite number of different sessions, with this method it is not possible to guarantee that the protocol is correct for all possible sessions.

In general if there is some flaw in the analyzed session we can conclude that the protocol is insecure; on the contrary, we cannot say that the the protocol is secure because we have no guarantee that the correctness result scales to more complex sessions. [1]

As mentioned above, the process of producing the SPA specification of the system (protocol and enemy) presents some drawbacks; as a matter of fact, it is:

- time-consuming: to produce a SPA specification for a small system (usually consisting of more than one thousand lines), we need some days of work. Moreover, when we change the session, the system is modified and we have usually to rewrite the SPA specification;
- only possible for a few expert people: although SPA could be used to describe any concurrent system, it is a really low level language; hence only well trained people can program easily in SPA;
- error-prone: when we specify the system by hand it is likely to forget some actions or to add unnecessary ones. In the first case the specification is incomplete and could miss to represent some important aspects of the real system; in the second case, the specification may be incoherent, because something that never happens in the real system may show up.

Another problem consists of the generation of an *enemy agent*; no standard method to build a powerful enemy which depends on the analyzed protocol was proposed in [4, 5].

These problems have been solved in the following way:

1. a notation more abstract than SPA was defined, called VSP notation;
2. a set of rules to build automatically an enemy was defined;
3. a compiler CVS to translate the VSP notation into SPA (also generating automatically the enemy) was implemented.

[1] For the particular case of secrecy properties, some preliminary results which permit to extend the proofs over small sessions to more complex ones have been proposed in [9]. We do not address this issue here.

In the next sections we give a brief overview on the VSP notation, we explain how the enemy can be made more or less powerful (defining suitably some parameters), we show how to use CVS and finally an application of VSP and CVS for the Woo-Lam one-way authentication protocol [19] is presented.

3 Overview on the VSP notation

VSP is an extension of SPA extended to accommodate value-passing in the usual way [12]. Moreover it requires to fully specify the parameters of the analysis. In the VSP notation, the specification of a protocol is composed of four parts:

1. *Declaration of types (e.g.: Agent, Nonce, Key) and their values, declaration of the structure of the protocol messages, declaration and definition of roles;* in this part we define abstractly the protocol, leaving unspecified the actual session and the other parameters for the analysis.
2. *Definition of the session;* this is done by considering a particular combination (sequential and/or parallel composition) of roles, each one instanced by assigning actual values (e.g. agent names, nonce values) to its parameters.
3. *Setting of the parameters of the enemy;* in this part we specify the features of the enemy, its initial knowledge and capability (e.g. ability to store intercepted values).
4. *Definition of visible actions;* the actions specified in this section are those used to observe and detect enemy interferences.

After the first and second parts of the VSP specification, CVS is able to build the SPA code of the whole protocol. If, in the session definition, the enemy instance appears, then CVS also builds the SPA code for the enemy according to the parameters specified in the third part. A more detailed description of these specification phases, is reported in Section 6 where we propose the VSP specification for the Woo-Lam one-way authentication protocol.

4 The Enemy

In order to analyze the protocol, it is necessary to study its behaviour when it is attacked by a malicious user, often called enemy. In particular, the enemy should be able to:

- intercept any sent message;
- store the information contained in intercepted messages for later uses;
- introduce new messages into the system using information he knows.

An enemy with the above features is able to use the information gained in the last intercepted message when he introduces new messages into the system or he can store stolen information and reuse it in future messages. It is important to note that an enemy with memory can store both *atom* information (like a single nonce) and structured information as *encrypted blocks* even if he cannot decrypt them. We assume that the encryption algorithm used is secure: the enemy is able to decrypt a message only if he knows the key that, hence, can be his own private key or one that he has learnt from an intercepted message.

It is possible to generate an enemy (for a given protocol) in an automatic way and, in principle, we can do it by generating the "most powerful" enemy. However the exponential nature of the enemy generation could prevent to complete the analysis, as we could not check SNNI in a reasonable amount of time or even run out memory very soon. For this reason we have given the possibility to restrict the enemy in some way. In VSP it is possible to set suitably the parameters of the enemy. These parameters are:

- *the initial knowledge*: the enemy can know a set of facts, such as the names of the participants or the agent public keys. The initial enemy knowledge should be the same as a real enemy on the net can know of. Nonetheless, in some cases, it is possible to reduce this set to speed-up the analysis or to see which pieces of information are strictly necessary for the attack.
- *memory*: the enemy in the session could learn new pieces of information and store them if the clause *memory* is true. When we define an enemy with memory we can choose the number of encrypted blocks that the enemy can store. An enemy without memory can reuse stolen information just after its interception, then forgetting it.

By setting these parameters, CVS is able to generate enemies of different power. These parameters can be set (by the user) depending on the kind of the analysis and of the features of the analyzed protocol.

5 Using CVS

The compiler CVS is written in standard C^{++} to facilitate the portability to different development environments and works under UNIX (as well as Windows). CVS compiles a VSP file to SPA code. The command sequence to compile a VSP file is:

```
cvs filename.vsp filename.spa
```

where the first *filename* is the source, and the second one the destination. With this command CVS first compiles the source file, checking if it contains syntax errors; in this case the compilation is interrupted and a suitable diagnostic message is generated. If there are no syntax errors, then CVS creates the output file denominated with the destination name, that can be directly loaded by CoSeC [4]. CVS generates also the log file, consisting of two kinds of information: the correspondence between the process names in VSP notation and those in the SPA code, as well as the knowledge that could be learnt by the enemy during a session.

Once CVS has generated the SPA code, CoSeC analyzes the protocol by checking SNNI on the SPA code. A protocol E is SNNI if and only if $E/H \approx_T E \setminus H$ where H represents the set of actions that correspond to the enemy activity; E/H is the process where we "hide" actions in H, this means that we leave the intruder act indiscriminately; $E \setminus H$ is the process where we "restrict" actions in H, i.e., we forbid the execution of the enemy. SNNI requires that these two processes are trace equivalent (written $\approx_T$). So if SNNI holds the protocol composed with the enemy (E/H) and the protocol without the enemy $(E \setminus H)$ must be equivalent. This has indeed the effect of checking if the enemy is able to interfere in some way with the protocol.

The usefulness of the tool has been tested with several case-studies of different protocols and for some of these protocols, we have found new (at least undocumented, to the best of our knowledge) attack sequences. In the next section we show how the methodology works, with a specific case-study, i.e., the Woo-Lam one-way authentication protocol [19].

6 The Woo and Lam one-way Authentication protocol

We consider the Woo and Lam one-way authentication protocol [19], that was already shown incorrect by Abadi (as reported in [20]). The aim of the Woo-Lam protocol is to establish one-way authentication between two users, say Alice (A) and Bob (B), using an authentication Server (S). Woo and Lam state that the protocol is correct if: "whenever a responder finishes execution of the protocol, the initiator of the protocol is in fact the principal claimed in the initial message".

Woo and Lam start with an initial version of the protocol and progressively simplify it. The version that we are going to analyze is the final version of the protocol. We will denote with K_{US} a key shared between a generic user *U* and the Server *S*. 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_U will represent a nonce generated by user *U*. The protocol is composed of five steps:

Message 1. $A \rightarrow B : A$
Message 2. $B \rightarrow A : N_B$
Message 3. $A \rightarrow B : K_{AS}(N_B)$
Message 4. $B \rightarrow S : K_{BS}(A, K_{AS}(N_B))$
Message 5. $S \rightarrow B : K_{BS}(N_B)$

Intuitively, *Message 1* represents the authentication request from *A* to *B*. In this message *A* sends her name in plain to *B*. In *Message 2*, *B* replies to *A* by sending the new nonce N_B in plain. Once *A* has received the nonce N_B, she encrypts it with the key K_{AS} and sends the result to *B*. In *Message 4*, *B* encrypts the name of the initiator *A* and $K_{AS}(N_B)$ with the key K_{BS} and sends the result to *S*. The Server receives a message of the form $K_{BS}(U, C)$, decrypts the cipher text C with the key K_{US} encrypts the result with K_{BS} and sends the result to B. When *B* receives the last message, he checks that it contains the nonce N_B and, in such a case, he concludes that was really talking to *A*, since only *S* could have decrypted the fourth message and checked the identity of the initiator.

Now we show how to specify the protocol using the VSP notation.

6.1 First phase of the VSP Specification

As we said above, a VSP protocol specification is divided into four parts. The first part consists of four sections and defines abstractly the protocol.

Name declarations. In the first section we define identifiers (e.g. agent names) that can belong either to built-in types or to user-definable ones. We have two kinds of built-in types: those where the identifiers have to be declared explicitly (i.e. **Agent**, **Key**), and those where the identifiers are implicitly generated by the compiler (i.e. **AsimKey**, **SimKey**). For the Woo-Lam protocol the identifiers declared explicitly are *A*, *B*, *S*, *E* of (built-in) type **Agent**, and *NB*, *NE* of (user-defined) type **Nonce**.

This protocol requires two shared keys between each of the honest users and the Server. These keys belong to a built-in type **SimKey**; their identifiers in SPA code are automatically generated by the CVS compiler hence we do not need to declare them. The built-in type **AsimKey** models the public and private keys used in asymmetric cryptographic algorithms. If we want to represent other key values, exchanged or shared between the agents, we can use the built-in type **Key** which permits to explicitly declare them.

```
#Names
Agent: A,B,E,S;
Nonce: NB,NE;
```

The identifiers *A*, *B*, are the names of the two honest agents; *S* is the name of the Server; *E* is enemy's name. *NB*, *NE* are the nonces generated by the agents *B* and *E*.

Action declarations. In the second section of the first part, the structure of the actions is declared. They can be divided into two sets. The first one contains the actions which are necessary to define the protocol, i.e., the actions representing the messages exchanged between the agents during a run of the protocol. The second one contains the actions used to observe the execution of the protocol and are necessary for the analysis.

Each action is composed by an action name with possibly one or more arguments, whose types are declared in the previous section. We have two kinds of argument: clear (or in plain text), crypted (or cipher text). A crypted block argument is enclosed between two brackets that are preceded by the word *crypt*. The first argument of a crypted block must be a key-type (e.g. **SimKey**, **AsimKey**, **Key**), the other arguments are the pieces of information to be crypted.

For every action it is possible to express some conditions on the values of the parameters. There are two kinds of condition: *equal to*, *different from*. For instance, "I_request(Agent,Agent) cond(0!1)" denotes the fact that the first argument of the action I_request must be different from the second one. These conditions are important to avoid senseless operations (e.g. self-request).

The VSP specification of the messages for the Woo-Lam protocol is:

```
#ActionDec
msg1(Agent)
msg2(Nonce)
msg3(crypt(SimKey,Nonce))
msg4(crypt(SimKey,Agent,crypt(SimKey,Nonce)))
msg5(crypt(SimKey,Nonce))
I_request(Agent,Agent)  cond(0!1)
R_running(Agent,Agent)
I_Connect(Agent,Agent,Number)
R_Connect(Agent,Agent,Number)
```

Note that it is possible to define actions parameters with many levels of nested encryption (e.g. action msg4). In this way it is possible to model also the mechanism of *digital signature*. Besides message declaration, we have specified five actions that we will use to observe the protocol executions. I_request(A1,A2) (R_running(A1,A2)) corresponds to the start of a session between agent A1 and agent A2, where A1 is the initiator (responder). The last two actions are used to observe the correct termination of the initiator and of the responder, respectively. In such actions, the special built-in type `Number`, is used whose aim will be explained in Section 7.

Roles declaration. In this section we declare the "roles" that the agents can play in a single session of a given protocol. The name of the enemy role is predefined as to be "Enemy". In the Woo-Lam protocol there are three roles: *Initiator*, *Responder*, *Server*. In a session the agents may play more than one role and more times the same role; in such a case we call it a "multiple session". In VSP the roles are simply declared as follows:

```
#RolesDec
Initiator
Responder
Server
Enemy
```

Processes definition. In this section we define the body of each role as a VSP process. Every VSP process consists of three parts (with the exception of the enemy that is generated automatically by CVS):

1. *a list of parameters*: this represents the declaration of the formal parameters of the process. For example, the VSP notation for the Responder in the Woo-Lam protocol is the following:

   ```
   #ProcessesDef
   Responder(dst: Agent, nonr: Nonce)
   ```

   Typically the first parameter (dst in this case) is used for the name of the agent that is playing the role. The other parameters strictly depend on the analyzed protocol. For the responder of the Woo-Lam protocol the second parameter is a nonce; indeed the responder has to generate a different nonce for every run.

2. *A set of variables*: these represent *the local environment* of the process. Every variable will appear in the list of parameters of the actions used in the process. It is possible to define some restrictions on the values that every variable can assume. For example the VSP notation for the Initiator of the Woo-Lam protocol is:

   ```
   Var
   Agent: dst1;     cond(Responder,Enemy)
   Nonce: nonr1;
   ```

   The side condition expresses that the variable dst1 can only assume the values in the set {Responder,Enemy}. For example if the name of the agent, playing the Responder role is *B* and the Enemy is *E*, then the variable dst1 can only hold *B* or *E*. The meaning of these variables will be clear when the body of the initiator process will be defined.

3. *A sequence of instructions*: it is the body of the VSP process. Instructions can be:

   - *Actions*: they can only be those previously declared in *#ActionDec*. They can be either input actions or output ones. Output actions are prefixed by a character `'` . Communication takes place by synchronous hand-shake.
   - *Conditional instructions*: VSP offers an if-then-else construct.

The bodies of the VSP processes are reported in Table 1.

As we explained, for some built-in types, CVS automatically generates their SPA values. In the Woo-Lam protocol this happens for the shared keys. We use the function of the form KA(A1), where A1 is a variable that belongs to type **Agent**. This function represents a key which is suitably generated for agent $A1$. This is useful when modelling situations where there is a trusted server that shares a key with every agent in the system. Indeed, we can refer to such keys directly through the KA function, as done in the Woo-Lam protocol specification. For example, the key used for encryption in msg3 of the Initiator is KA(src), which represents the key it shares with the server.

Another function used in the process body is `NUMB`; it is only used for the analysis and we will explain how in Section 7.

Besides, the specification for the enemy has to be given. This definition only fixes the strictly necessary parameters allowing the enemy to act as a honest user.

```
Enemy(e_id:Agent,n_id:Nonce)
Begin
End
```

As for Initiator and Responder, the needed pieces of information are the agent name (enemy) e_id and one nonce n_id.

6.2 Second phase of the VSP Specification

Session definition. Here we have to generate a mapping between the agent names and the roles; this mapping is not required to be injective (the same agent can be active in different roles), neither the inverse (several agents can be contemporaneously active in the same role). The actual agents may be composed either in parallel or in sequence. [2] Hence, complex sessions can be defined not only by generating more instances of each role, but also with a mixed composition (parallel/sequential) of processes. This specification phase is very important as it defines the actual structure of the (possibly multiple) session, hence of the whole

[2] In the latter case, the second agent can start its execution only after the first one has finished.

```
#ProcessesDef
Initiator(src : Agent)
Var
Agent: dst1;    cond(Responder,Enemy)
Nonce: nonr1;

Begin
        I_request(src,dst1).
        'msg1(src).
        msg2(nonr1).
        'msg3(enc(KA(src),nonr1)).
        'I_Connect(src,dst1,NUMB)
End

Responder(dst: Agent, nonr: Nonce)
Var
Agent: src1;    cond(Initiator,Enemy)
Agent: src4;    cond(Initiator,Enemy)
Nonce: nonr2,nonr1;

Begin
        msg1(src1).
        'R_running(dst,src1).
        'msg2(nonr).
        msg3(enc(KA(src4),nonr2)).
        'msg4(enc(KA(dst),src1,enc(KA(src4),nonr2))).
        msg5(dec(KA(dst),nonr1)).
        if (nonr1=nonr) then
         'R_Connect(dst,src1,NUMB)
endif
End

Server(ser: Agent)
Var
Agent: dst2;    cond(Responder,Enemy)
Agent: src2;    cond(Initiator,Enemy)
Agent: src3;    cond(Initiator,Enemy)
Nonce: nons;

Begin
        msg4(dec(KA(dst2),src2,dec(KA(src3),nons))).
        if (src2=src3) then
                'msg5(enc(KA(dst2),nons))
        endif
End
```

Table 1. The bodies of Initiator, Responder and Server

SPA specification. A VSP process can be instantiated using the names declared in the section `#Names`. For the Woo-Lam protocol, we have decided to study a simple single session with one initiator (namely A), one responder (B), one server (S) and the enemy (E), all composed in parallel. In the VSP notation, this is written as follows:

```
#Processes
Initiator(A)
Responder(B,NB)
Server(S)
Enemy(E,NE)
```

If we want to analyze a session where two or more processes are composed in sequence we use the `;` operator. For example, the VSP specification for the representation of two initiators in sequence, and a responder in parallel is the following:

```
Initiator(A);Initiator(B)
Responder(B,NB)
```

Channel definition. In this section we specify which are the actions that can be sent and received on the communication channel, i.e., that represent the protocol messages. They are a subset of the actions declared in `#ActionDec`. For our example:

```
#RestrictionOn
channel(msg1,msg2,msg3,msg4,msg5)
```

The actions which are not declared as protocol messages are called *visible actions*, as they are the actions observed in the analysis; we explicitly declare them in the last part of the VSP specification.

6.3 Third phase: the Enemy

In this section we specify the enemy parameters. CVS then builds in an automatic way a SPA specification of the enemy according to the chosen parameters. We have already explained in Section 4 what are the enemy parameters that can be set up; now we show how they are set for the Woo-Lam protocol.

```
#EnemyKnowledge
Agent: All;
Memory:False;
```

This specification means that the enemy initially knows all the agent names. Moreover, it has no memory, so when he introduces new messages in the system, besides the agent names he can additionally use only the information gained in the last intercepted message. It is possible to create a more powerful enemy with the clause: “Memory:True;”, but in this case-study an enemy without memory is sufficient.

We could want to give to the enemy the knowledge of only a subset of the identifiers of a particular type. This can be done by specifying the type followed by the identifiers list. For example, to represent that the enemy knows only the agent names A and B, we can write: `Agent:A,B;` .

Note that in the Woo-Lam protocol double encrypted messages are present; CVS is able to generate an enemy that also deals with this kind of messages in a correct way.

6.4 Fourth phase of the VSP: the Visible Actions

In the last part of the protocol specification, we explicitly specify the set of visible actions. If the enemy modifies their execution then property SNNI is not verified; as we will see this could correspond to the presence of one (or more) flaw(s) in the protocol.

When we analyze a protocol through SNNI, the choice of visible actions is very relevant. Typically, we observe the starts and the correct termination of the parties, since we have seen that these are sufficient to detect the attacks in all the protocols we have analyzed. Indeed, in most protocols we can simply observe the correct terminations but this does not hold in general, e.g., in the Woo-Lam protocol this is not sufficient. For the Woo-Lam protocol the chosen visible actions are:

```
#Observe
R_running(Responder,Initiator)
I_request(Initiator,Enemy)
R_Connect(Responder,Initiator,Number)
I_Connect(Initiator,Enemy,Number)
```

As said above the first two actions represent the start of the protocol for the responder and the initiator respectively; while the last two actions represent their correct termination.

The presence of an interference is not a guarantee that an attack is presented in the protocol session (as we will see in the next session); on the other hand the absence of interference ensures that no security flaw is present in the protocol session according to the observed actions. In this sense the approach offers a sufficient, but not necessary, condition to correctness.

7 Interpreting the Interference Traces

By analyzing the VSP specification of the Woo-Lam protocol, we find out that it is not SNNI. The result of this analysis consists of a list of all interference traces, which do not necessary correspond to an attack. Indeed some of them are permitted commits between the enemy and the honest agents; some other traces can be part of an enemy request that the enemy cannot conclude; the rest of the traces may correspond to attacks. We explain how to interpret the output of CoSeC showing the list of interferences obtained for the Woo-Lam protocol, when observing the actions defined above.

We can divide the set of interference traces found for this protocol into three different sets. The first set is:

```
(1) I_request_A_E 'I_Connect_A_E_3
(2) I_request_A_E 'I_Connect_A_E_4
```

These two traces correspond to two requests made by the initiator *A* to the enemy *E*, followed by two commits between them. These do not correspond to an attack: they appear simply because we consider the enemy as a normal user, so he can perform a normal run with some other user.

The second set is:

```
(1) I_request_A_E 'R_running_B_A
    'I_Connect_A_E_4
(2) 'R_running_B_A I_request_A_E
    'I_Connect_A_E_4
```

These two traces are similar to the previous ones; the only difference is that the enemy does a request to the Responder (pretending to be *A*), but *B* is not able to finish this second run.

The third set is:

```
(1) I_request_A_E 'R_running_B_A
    'I_Connect_A_E_3 'R_Connect_B_A_1
(2) I_request_A_E 'R_running_B_A
    'R_Connect_B_A_1 'I_Connect_A_E_3
```

These two last sequences correspond to an attack. We can understand this because the Responder *B* is able to do a commit with the Initiator *A* (action `'R_Connect_B_A_1`) without a request made by the Initiator *A*. These two interferences represent the same attack. Indeed they differ only in the order of the two `Connect` actions. As a matter of fact, when the initiator and the responder have finished the protocol, they can emit their respective `Connect` actions in any order.

The number that appear on the end of a `Connect` action is generated by CVS when the function NUMB is an argument of an action in the VSP body. It basically labels, with a different number, every possible execution "branch" that can be taken in the SPA-code. This is very helpful when we have to reconstruct an attack sequence performed by the enemy. Indeed, given a certain `Connect` labelled with a number we can see exactly in the SPA-code what the enemy has done. [3] In this way we can understand all the steps of the attack that can be written as follows:

1.a $A \rightarrow E : A$
1.b $E(A) \rightarrow B : A$
2.b $B \rightarrow E(A) : N_B$
2.a $E \rightarrow A : N_B$
3.a $A \rightarrow E : K_{AS}(N_B)$
3.b $E(A) \rightarrow B : K_{AS}(N_B)$
4.b $B \rightarrow S : K_{BS}(A, K_{AS}(N_B))$
5.b $S \rightarrow B : K_{BS}(N_B)$

The initiator *A* begins a first session with the enemy *E*; when the enemy receives the message 1.a, it sends a request to the responder *B* (message 1.b) starting another session where it pretend to be *A*. The responder answers and sends his nonce N_B to *A*. The message 2.b is intercepted by *E* that reuses the nonce N_B in the first session, sending it to *A*; *A* encrypts N_B and sends it to *E* which simply forwards it to *B*, (message 3.b). Now *B* can conclude the protocol with the server *S*. At the end of the protocol, the responder *B* thinks to be connected to the initiator *A*, while *A* has never begun a run with *B*!

The SPA specification represents a single session of the protocol, where every participant plays just one role at a time, but the attack found occurs in a multiple session. This happens because the enemy plays two different roles: the responder in session a and the initiator in session b.

This particular attack sequence is new, even if it is not the first one found for this protocol. All the other attacks reported in literature are avaible in [2]. However it is the only one which can be carried out in a single session; all the other attacks reported in [2] occurring in multiple sessions, were discovered through CVS by choosing an appropriate session. In particular we have: the attack by Abadi and Needham (as reported in [20]) where a partecipant plays two times the responder role; the attack by Woo and Lam in [20] where a partecipant plays the responder role and the initiator role.

8 Conclusion and future research

In this work we have presented a new notation, called VSP, which permits to define security protocols at a level more abstract than SPA. The VSP specifications are compiled to SPA code through the CVS compiler. The main advantages of VSP are (i) the possibility of giving very compact value passing specifications and (ii) the high level of parametrization in the analysis. In particular we have shown that it is possible to set up a number of parameters which reduce significantly the state space of the analyzed system. This permits to start the analysis with a simplified enemy, which is still generated automatically, and then proceed with more complex ones. In this way we can start with very fast checks which could already reveal some simple attacks on the protocol. Moreover this "gradual" approach is useful to understand how much power the enemy needs in order to be able to attack the protocol.

Once the SPA code is obtained, it can be analyzed using

[3] We do not show here the details of this analysis since we should report the whole SPA-code.

a Non-Interference based technique. This is what mainly differentiate our compiler from other existing ones. For example, the CASPER compiler [10] translates protocols defined in a language similar to VSP to CSP [6] specifications. It is very similar to our CVS if observed at the specification level. However, the way protocols are analyzed is completely different. The Non-Interference check can be seen as a sufficient condition to the absence of every kind of (not only authentication) attacks on the particular session we are analyzing. Indeed it checks that the presence of the enemy does not change in any way the behaviour of the protocol with respect to a set of "relevant" actions. These actions are the only things we have to fix in the analysis and represent what we want to observe of the protocol. We do not have to define a particular property on such actions. If the check is successful, we can conclude that the enemy is able to do nothing (good or evil) on such observable actions. For example, we can easily check the authentication properties verified by CASPER by simply considering as relevant the actions that are observed by such particular properties (e.g., the run and commit actions). If the Non-Interference check is unsuccessful we obtain a set of interferences that can be easily analyzed in order to understand which interferences are "dangerous" and which are not. The CVS tool has been developed with the aim of supporting this Non-Interference based approach for the analysis of cryptographic protocols which has been first studied and shown flexible enough to reveal different attacks in [4, 3].

Before the implementation of CVS, only few protocols were analyzed using such a technique. This tool has permitted us to successfully analyze many more protocols. This has in a sense confirmed that the Non-Interference based analysis of security protocols is profitable. We report here some of them containing well-known attacks: the Yahalom protocol presented [1] and shown insecure in [17]; the Andrew protocol [16] with the attack presented in [8]; the Otway-Rees [14] shown flawed in [15] and the TMN protocol [18] with the attacks reported in [11]. All the attacks on these protocols presented in the literature have been captured by CVS. Moreover, in the TMN we have found an attack which seems new. The Otway-Rees is particularly interesting since it is requires an enemy with memory in order to attack it. In all of the checks on these protocols we have keenly exploited the high parametrization level of our tool, that has permitted to obtain quite fast analysis and detection of attacks.

As a future work, we are considering other techniques based on the introduction of some auxiliary information which could allow a more efficient and "local" analysis of some properties. We are also trying to enhance the tool with some feature for cryptographic analysis in order to detect some attacks based on particular properties of the cryptographic algorithms adopted. Another possible improvement is a graphical interface that accepts as inputs the roles of the protocol, the handshake message between the participants (in a symbolic form) and the observable actions.

References

[1] M. Burrows, M. Abadi, and R. Needham. "A Logic of Authentication". *Proceedings of the Royal Society of London*, 426:233–271, 1989.

[2] J. Clark and J. Jacob. "A Survay of Authentication Protocols Literature: version 1.0". http://www.cs.york.ac.uk/ jac/papers/drareview.ps.gz, November 1997.

[3] R. Focardi, A. Ghelli, and R. Gorrieri. "Using Non Interference for the Analysis of Security Protocols ". In *Proceeding of the DIMACS Workshop on Design and Formal Verification of Security Protocols (H. Orman and C. Meadows Ed.)*, DIMACS Center, Rutgers University, September 1997.

[4] R. Focardi and R. Gorrieri. "The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties". *IEEE Transactions on Software Engineering*, 23(9), September 1997.

[5] R. Focardi and R. Gorrieri. "Analysis of Authentication Protocols using Non-Interference". Technical Report CS98-18, University of Venice, October 1998.

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

[7] "Entity Authentication Using Symmetric Techniques". Report ISO/IEC JTC1.27.02.2 (20.03.1.2), June 1990.

[8] G. Lowe. "Some New Attacks upon Security Protocols". *IEEE Computer Security Workshop 1996*, pages 164–165, 1996.

[9] G. Lowe. "CASPER: A Compiler for the Analysis of the security protocols". In *Proceedings Tenth IEEE Computer Security Foundation Workshop, (CSFW'97)*, pages 18–30, Rockport Massachusetts (USA), June 1997. IEEE Press.

[10] G. Lowe. "Towards a Completeness Result for Model Checking of Security Protocols". In *Proceedings Eleventh IEEE Computer Security Foundation Workshop, (CSFW'98)*, Rockport Massachusetts (USA), June 1998. IEEE Press.

[11] G. Lowe and B. Roscoe. "Using CSP to detect errors in the TMN protocol". *IEEE Transaction on Software Engineering*, 23 (10), October 1997.

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

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

[14] D. Otway and O. Rees. "Efficient and timely mutual authentication". *Operating system review*, 21, Jennuary 1987.

[15] L. C. Paulson. "The Inductive Approach to Verifying Cryptographic Protocols". In *Proceedings Tenth IEEE Computer Security Foundation Workshop, (CSFW'97)*, Rockport Massachusetts (USA), June 1997. IEEE Press.

[16] M. Satyanarayanan. "Integrating Security in large distribuited system". *ACM Transactions on Computer Systems*, 7(3):247–280, 1992.

[17] P. Syverson. “A Taxonomy of replay attacks”. In *Proceedings Seventh IEEE Computer Security Foundation Workshop, (CSFW'94), (Li Gong Ed.)*, Franconia (NH), June 1994. IEEE Press.

[18] M. Tatebayashi, N. Matsuzaki, and D. B. Newman Jr. “Key distribution protocol for digital mobile communication system”. *Advances in Cryptology: Proceedings of Crypto '89*, 435:324–333, 1990.

[19] T. Woo and S. Lam. “Authentication for distributed systems”. *Computer*, 25:10–10, March 1992.

[20] T. Woo and S. Lam. “A lesson on authentication protocol design”. *Operating system review*, pages 24–37, 1994.