A Compiler for Analysing Cryptographic Protocols Using Non-Interference *

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 automated by 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 protocol description and its environment to be written by hand. This problem has been solved by defining a protocol specification language more abstract than SPA, called VSP, and a compiler CVS that automatically generates the SPA specification for a given protocol described in VSP. The VSP/CVS technology is very powerful and its usefulness is shown with some case-studies: the Woo-Lam one-way authentication protocol, for which a new attack to authentication is found, and the Wide Mouthed Frog protocol, where different kinds of attack are detected and analyzed.

1 Introduction

Many security protocols, using cryptographic techniques, have been proposed in the literature to achieve several goals, for example authentication and secrecy. They are supposed to succeed even in the 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 [16, 14, 17] for capturing (known) attacks upon some cryptographic protocols, such as the Needham-Schroeder public key authentication protocol [33] and the ISO SC27 [20]. Analyzing a security protocol using NI consists of the following

*Extended and revised version of "CVS: A Compiler for the Analysis of Cryptographic Protocols", in *Proceedings of 12th IEEE Computer Security Foundations Workshop* (CSFW'99), pages 203-212, IEEE CS Press.

steps. We specify the whole system (protocol and enemy) in Security Process Algebra (SPA [16]). 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 [16] takes as input the SPA specification representing the system that we want to analyze and checks a particular NI property, called *Strong Non deterministic Non-Interference* (SNNI, for short [16]), on that process. SNNI is a generalization of NI to the SPA model which basically verifies if the enemy is able to change in some way the possible sequences of observable actions executed by the protocol. As a consequence, if the system verifies SNNI successfully, 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 actual attacks upon the protocol.

Hence, we do not specify any specific property that the protocol must satisfy; we only observe whether the intruder is able to modify (in some way) the flow of information among the honest participants. The task of understanding which is the set of interferences corresponding 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. In particular, we present a protocol specification language more abstract than SPA, called VSP, and a compiler CVS that automatically generates the SPA specification for a given protocol described in VSP. It is important to remark that the main motivation for the introduction of the new language VSP (together with the CVS compiler) is to simplify the analysis of protocols through the NI-based techniques defined on SPA: The underlying analysis technique is indeed what mainly differentiate CVS from other existing tools. As our main issue was not to propose a "better" high level formalism for protocol specification, we have taken advantage of existing work in the area. In particular, our VSP notation has been influenced a lot from the one used in the Casper compiler [25]. Section 9 is devoted to a detailed comparison of VSP/CVS with other existing tools.

The usefulness of the CVS 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.

2 Automating our approach

We have briefly described how to analyze a protocol using NI according to [17, 18]. It is important to note that if we want to analyze a protocol, we have to fix the *context of analysis* (i.e. the session) in advance. In particular, we have to establish the number of instances of each protocol role (e.g., initiator, responder, server) and which agents are playing such roles. For example, we could have Alice playing both the initiator and the responder role, and Bob just acting as a responder. Indeed, we can actually analyze one particular session at a time. Since a protocol can be potentially instanced to an unbounded number of different sessions, our method cannot guarantee that a protocol behaves correctly in all cases, as this would require an infinite number of checks, one for each possible session.

In general, if there is some flaw in the analyzed session, then we can conclude that the protocol is insecure. On the contrary, we cannot say that 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 the following drawbacks:

[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 [24]. We do not address this issue here.

- 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, the likelihood of forgetting some actions or adding unnecessary ones is much greater. In the first case, the specification is incomplete and could fail to capture 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 [16, 17].

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 was defined to automatically build the enemy component;

3. a compiler, called CVS, was implemented to translate a VSP script into SPA code. CVS incorporates the set of rules in 2) thereby automatically generating the enemy component.

The paper is organized as follows. In the next section we give a brief overview on the NI-based analysis technique. In Section 4 we define the enemy and we explain how it can be made more or less powerful (suitably defining some parameters). Section 5 is devoted to a detailed explanation of the VSP notation. Section 6 reports the VSP specification of the automatically generated enemy component. In Section 7 we present the first case study: the Woo and Lam one-way authentication protocol, which is useful to understand the VSP notation and the NI-based analysis. The main advantage of our analysis technique is indeed exemplified in Section 8 where the Wide Mouthed Frog protocol is analyzed in detail. Here, it is shown that in a single check of SNNI it is possible to detect a number of attacks on different security properties. Section 9 is devoted to a comparison of VSP/CVS with other existing tools. Finally, in Section 10 we give some concluding remarks and Appendix A reports some details on SPA and trace-equivalence.

3 The analysis of protocols using CVS and CoSeC

In this section we describe how the CVS/CoSeC tool is used in the analysis of cryptographic protocols. Given a VSP specification of a protocol, we can compile it using CVS producing the corresponding SPA code. This translation is carried out by following standard techniques (see Appendix A), hence guaranteeing that each translated SPA protocol *correctly* corresponds to its original VSP version. The output of CVS can directly be used as input to the CoSeC tool which is able to determine all the execution traces induced by the enemy over the protocol. Such execution traces, called *interferences*, are then analyzed in order to see if they correspond to attacks and, in such a case, to determine which is the specific attack sequence (see Figure 1). This last phase, called interference analysis, is not automated yet.

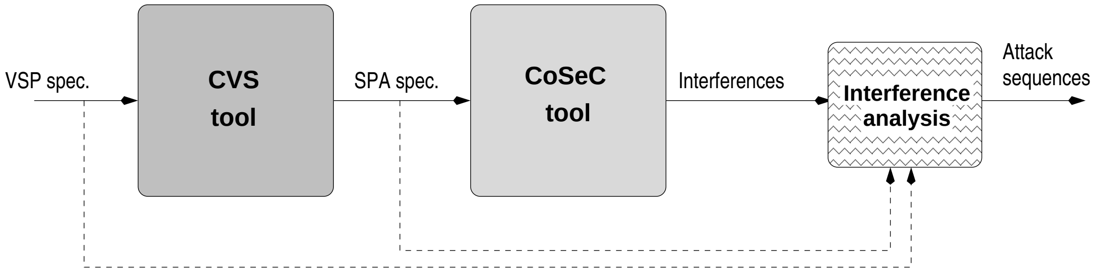

Figure 1: Analysis scheme.

The compiler CVS is written in standard C^{++} to facilitate the portability to different development environments and works under UNIX (as well as Windows). 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, that can be directly loaded by CoSeC [16]. CVS also generates the log file, consisting of two kinds of information: the correspondence between the process names in VSP notation and those in the SPA code, and a detailed description of the automatically generated enemy component containing information that can be useful in the analysis, for example, the knowledge of the enemy in a particular state.

Once CVS has generated the SPA code, CoSeC analyzes the protocol by checking the security property SNNI [16] on the SPA code. Intuitively, a protocol P composed with the enemy E is SNNI if it is behaviourally equivalent to P. This has indeed the effect of checking whether the enemy is able to interfere in some way with the protocol. This property is formally defined using the operators of the SPA language as follows: Let P_E be a SPA specification of protocol P composed with the enemy E, and let H represent the set of actions that correspond to the enemy activity. P_E is SNNI if and only if $P_E/H \approx_T P_E \setminus H$. In particular, P_E/H is the process where all the actions in H are "hidden", i.e., transformed into "internal" actions that are always executable. This makes the intruder act indiscriminately on P. On the other hand, $P_E \setminus H$ is the process where we restrict actions in H, i.e., we forbid their execution and, consequently, the execution of the enemy. SNNI requires that these two processes are trace equivalent ($\approx_T$). So if SNNI holds the protocol with the enemy (P_E/H) and the protocol without the enemy ($P_E \setminus H$) must exhibit the same execution traces. An interesting feature of SNNI is that it verifies different properties depending on which are the events we observe in the protocol execution. These events are represented by observable actions suitably inserted in the protocol specification. Once these actions are specified, SNNI verifies if the enemy is able to interfere on the execution of these particular actions. Thus, if we can specify all the observable actions that characterize a set of security properties then SNNI will be able to verify all these properties in just one check. This can be easily seen by observing that SNNI implies properties of the form "P under the attack of E never shows up *bad execution traces*". As an example, (a version of) the *agreement* property defined in [22] requires that "whenever Alice concludes correctly the protocol convinced she is communicating with Bob, then Bob has indeed started the protocol with Alice". This property is checked over a protocol P by adding in the protocol specification two actions $commit(B, A)$ and $run(A, B)$, representing the fact that Alice has concluded correctly the protocol with Bob and the fact that Bob has started the protocol with Alice, respectively. Then, it is required that $commit(B, A)$ is always executed after $run(A, B)$. This property can be rephrased as follows: "P under the attack of E never shows up execution traces where $commit(B, A)$ is not preceded by $run(A, B)$".

Now, if we consider a process $\tilde{E}$ which executes all the possible sequences of the actions executable by P except the *bad execution traces*, then the corresponding property, called $Sec_{\tilde{E}}$, can be formalized as follows: P_E satisfies $Sec_{\tilde{E}}$ iff $P_E/H \leq_T \tilde{E}$, where, $\leq_T$ represents the *trace preorder*, i.e., we write $E \leq_T F$ if and only if all the traces of E are also traces for F. $Sec_{\tilde{E}}$ basically requires that the protocol with the enemy P_E/H only exhibits traces that are also executable by $\tilde{E}$ and that, consequently, are not bad traces.

It is now easy to see that, under a very reasonable assumption, SNNI implies $Sec_{\tilde{E}}$. First, we note that $P_E \setminus H \leq_T P_E/H$, i.e., interference traces are exactly those traces in P_E/H that do not occur in $P_E \setminus H$. This gives a necessary condition for a protocol to satisfy $Sec_{\tilde{E}}$. Indeed, if $P_E/H \leq_T \tilde{E}$ then, by $P_E \setminus H \leq_T P_E/H$, it must be $P_E \setminus H \leq_T \tilde{E}$. So, for a protocol to satisfy $Sec_{\tilde{E}}$, it is necessary that at least the protocol without the enemy ($P_E \setminus H$) does not execute bad actions, which is what we expect from a correct protocol specification. If we assume this, we can show that SNNI implies $Sec_{\tilde{E}}$. As a matter of fact, if P_E is SNNI then $P_E/H \approx_T P_E \setminus H$. Now, since $P_E \setminus H \leq_T \tilde{E}$ we find out that also $P_E/H \leq_T \tilde{E}$, i.e., P_E satisfies $Sec_{\tilde{E}}$.

It is important to observe that we need to define in P the actions that are relevant for the property $Sec_{\tilde{E}}$. As an example if we do not define the $commit(B, A)$ and $run(A, B)$ actions in P the agreement property is always satisfied, since such actions are never executed. In such a case the SNNI check will reveal nothing about the agreement property. If we instead define in P a set of actions which is relevant for n different properties $Sec_{\tilde{E}_1}, \ldots Sec_{\tilde{E}_n}$, then SNNI will verify all of them in just one check. More details about the rephrasing of security properties in a NI-based schema can be found in [18].

4 The Enemy

In order to analyze the protocol, it is necessary to study its behaviour in the presence of a malicious user, often known as the *enemy*, which can perform the following actions:

- 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 previously intercepted messages when he introduces new messages into the system. In Section 6, we will give a detailed specification of this enemy in the VSP notation. 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.

In general, a protocol should be checked against every possible enemy. Indeed, it is possible to formally prove that for the trace-based properties $Sec_{\tilde{E}}$ and SNNI defined in Section 3 it is sufficient to consider the so called *most powerful* enemy described above, i.e., the enemy which is able to intercept and store every information exchanged during the protocol execution. If we call *Top* such an enemy, it is indeed possible to prove that $P_E/H \leq_T P_{Top}/H$ for every enemy E, thus if P_{Top} satisfies $Sec_{\tilde{E}}$ or SNNI, then also every P_E satisfies such properties. More details can be found in [18], where SNNI corresponds to the static (i.e., with only the most powerful enemy) characterization of the more general property NDC [15], that checks a protocol against every possible enemy. For a given protocol, it is possible, in principle, to automatically generate the *most powerful* enemy. However the exponential nature of the enemy generation could prevent to complete the analysis, as SNNI verification could be not feasible in a reasonable amount of time or even could run out of memory. For this reason, in VSP we have given the possibility to limit the enemy size through suitable parameters:

- *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 The VSP notation

VSP extends SPA to accommodate value-passing in a *standard* way [31] (see Appendix A for more details). Moreover, VSP allows the user to specify in a compact way a wide range of cryptographic protocols. In particular, referring to cryptographic primitives, in VSP it is possible to describe every classical symmetric cryptosystem where the same key is used both for encryption and decryption. Some facilities are provided in order to handle shared keys between two or more agents and for declaring non-reversible hash-functions (potentially nested). These can also be used as encryption/decryption keys (more details below). VSP also allows the users to describe asymmetric cryptosystems (secret and public keys). The two complementary usages of asymmetric cryptography are both implemented in VSP: secrecy using the public/secret keys and signature using the secret/public keys. The symmetric and asymmetric cryptosystems can be clearly used also together in a protocol specification, and for both of them the CVS system supports nested encryption.

Besides the protocol specification, VSP also requires the user 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 an abstract description of the protocol, leaving the actual components of the session and the other parameters of the analysis unspecified.

2. *Definition of the session.* In this part, we define the actual system by specifying the combination (sequential and/or parallel composition) of roles, where each one is instantiated with actual values, for example, agent names and nonces.

3. *Enemy setting.* In this part, we define the enemy component by specifying its initial knowledge and its memory capacity.
4. *Definition of visible actions and secret values.* 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 and the fourth one.

In the following we describe in detail the four phases that compose a VSP specification.

5.1 First phase

The first phase consists of five sections that define the protocol abstractly.

Name declarations. In the first section we define identifiers (e.g. agent names) that can belong either to built-in types or to user-defined 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. **AsymKey**, **SymKey**). A number of protocols assumes that there exists a trusted server which shares a key with every user. This can easily be handled through the built-in type **SymKey** which makes the key identifiers in the SPA code generated by the CVS compiler automatically. We can then refer to such keys through the function $\mathrm{KA}(A)$ that returns the key for agent A. Analogously, the built-in type **AsymKey** 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 the user to explicitly declare them.

An example of name declaration follows:

```
#Names
Agent:  A,B,E;
Nonce:  NA;
Key:    Kab;
```

Function declarations. Here the user can declare non-reversible hash-functions (potentially nested) and the predefined **SharedKey** function, used to represent shared key among an arbitrary set of agents. These can be used as encryption/decryption keys. An example of functions declaration follows:

```
#Functions
F : Agent, Nonce;
H : Agent, F, Key;
SharedKey : Agent, Agent, Agent;
```

The first simple function `F` takes an agent `A` and a nonce `N` as inputs, written `F(A,N)`, and generates a SPA value of the form `F_A_N`. Function `H` is a nested function, as it takes values generated by `F` as inputs. For instance, `H(A,F(B,N),K)` generates the SPA value `H_A_F_B_N_K`. The last function is a built-in predefined function, used to represent a shared key among three agents. E.g., `SharedKey(A,B,C)` represents a key which is shared among `A`, `B` and `C`.

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 among the parties 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 plain text), crypted (or cipher text). A crypted block argument is enclosed between two brackets that are preceded by the word `crypt`, where the first argument represents the cryptographic key. In particular, the first argument of a crypted block must be a key-type (e.g. **SymKey**, **AsymKey**, **Key**), the other arguments are the pieces of information to be crypted. For example:

```
msg1(Agent)
```

defines the message `msg1` which is composed by a unique piece of information of type **Agent**, while

```
msg2(Agent,crypt(Key,Nonce))
```

represents the message `msg2` with two arguments: an agent name and an encrypted nonce. An example where the functions declared in the previous section are used is the following:

```
msg3(Agent,H,crypt(SharedKey,Nonce,F))
```

A value generated by function F is encrypted using a key shared between three agents while a value generated by function H is not 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,

```
msg4(Agent,Agent) cond(\$0 != \$1)
```

denotes the fact that the first argument must be different from the second one (`$i` corresponds to the $(\mathtt{i}+1)$-th parameter). These conditions are important to avoid senseless operations (e.g. self-request).

It is also possible to define action parameters with many levels of nested encryption. In this way it is possible, for example, to model *digital signature* mechanisms. We give an example of action declaration, where an action representing a message of the protocol and another action representing the observable event of the running of a certain agent are defined (more details about observable actions will be given below):

```
#ActionDec
msg2(Agent,crypt(Key,Nonce))
running(Agent)
```

Roles declaration. In this section we declare the roles that the agents can play in a specific session of the given protocol. The name of the enemy role is predefined as `Enemy`. In VSP the roles are simply declared as a sequence of (role) names. An example 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*, representing the declaration of the formal parameters of the process.

   Typically the first parameter is used for the name of the agent that is playing the role. The other parameters strictly depend on the protocol being analyzed/modelled and represent specific initial knowledge provided to that agent. For example in

   ```
   #ProcessesDef
   Responder(dst: Agent, n1: Nonce)
   ```

   we have a process `Responder` where the first parameter is an agent name and the second one is a nonce.

2. *A set of variables*, representing 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 can be assigned to the variables. An example of variable declaration follows:

   ```
   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.

3. *A sequence of instructions*, representing 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 the character `'` . Communication takes place by synchronous hand-shake: in one single atomic step, the two processes exchange a value across the channel represented by the pair of input/output actions with the same name, resulting in an internal invisible step.
   - *Conditional instructions*: VSP offers an if-then-else construct.

   The syntax is the following:

   $$\begin{array}{rcl} \mathit{INSTSEQ} & :== & \texttt{Begin}\ \ \mathit{INST}\ \ \texttt{End} \\ \mathit{INST} & :== & \mu \mid \mu\ .\ \mathit{INST} \mid \texttt{if (v1 = v2)}\ \mathit{INST}\ \texttt{endif} \mid \\ & & \mid \texttt{if (v1 = v2)}\ \mathit{INST}\ \texttt{else}\ \mathit{INST}\ \texttt{endif} \end{array}$$

   where μ ranges over the set of actions and $v1$ and $v2$ are bound variables or parameters. As expected, the variables that appear in input actions are bound, after the input, to the actual values received during the communication. Of course, when an output action occurs, all its parameters have to be already bound to some actual value. In order to handle crypted messages, VSP provides two built-in functions `enc` and `dec`, representing encryption and decryption respectively.

5.2 Second phase

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 (usually called *multiple*) 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 SPA specification. A VSP process can be instantiated using the names declared in the section `#Names`. If we want to analyze a session where two or more processes are composed in sequence we use the `;` operator. For example, a session where A plays the initiator role two times in parallel, B plays the responder role two times (with two different nonces) in sequence and E is the enemy appears as follows:

```
#Processes
Initiator(A)
Initiator(A)
Responder(B, NB1); Responder(B, NB2)
Enemy(E)
```

Channel definition. In this section we specify 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 example, if we have three messages `msg1, msg2, msg3`, then we specify it as follows:

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

In particular, when we declare these protocol messages, the parties of the protocol are forced to communicate using the corresponding actions `msg1, msg2` and `msg3`. The communication becomes internal and no external users can observe what is happening on the communication channel. All the actions which are not declared as protocol channels are called *visible actions*, as they are the actions that can be observed in the analysis; in the last part of the VSP specification it is possible to specify the subset of them that will be effectively observed during the analysis.

5.3 Third phase

In this section we specify the enemy parameters. CVS then automatically builds 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 in VSP.

We may wish to define the enemy's knowledge as a subset of the identifiers of a particular type only. This can be done by specifying the type followed by the identifiers list. For example

```
#EnemyKnowledge
Agent: A,B;
```

means that the enemy initially knows the agent names `A` and `B` only. We can use the special reserved word `All` in order to specify that the enemy knows every identifier for that particular type.

We can also specify if the enemy has memory or not. For example:

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

```
Memory: False;
```

means that it has no memory; more precisely, the enemy can only use the information gained in the last intercepted message. It is possible to create a more powerful enemy with the clause: `Memory:True;` making it able to memorize all the possible plain values plus a (finite) set of encrypted messages. The dimension of the memory is set to a natural number n through the clause: `Block: n`.

5.4 Fourth phase

In the last part of the protocol specification, we explicitly specify the set of visible actions that will be observed during the analysis. In particular, if the enemy is able to modify the behaviour of the protocol observed only through the execution of this particular set of actions, then property SNNI will not be 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. As already observed in Section 3, if we want to analyze a security property that requires a particular behaviour of the protocol with respect to some actions, it is sufficient to insert them in the set of observable actions. Moreover, if we consider a set that is big enough to detect different security properties, we can check all of them in a unique analysis. Our experience is that by simply observing the starts and the terminations we can reveal not only simple authentication problems (as also remarked in, e.g., [43, 27, 24]) but also a wide range of security properties, as we will show in Section 8 through a simple example.

For example, if we want to observe the running of the responder (which is represented by action `r_running`) and the correct termination for the initiator (action `i_commit`), we specify it as follows:

```
#Observe
r_running(Responder,Initiator)
i_commit(Initiator,Responder)
```

The presence of an interference is not a guarantee that an attack sequence is possible in the protocol session; on the other hand the absence of interferences 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.

Note that if we observe events where the enemy is involved (for example the running of the initiator with the enemy) this will certainly cause the presence of interferences that do not necessarily represent attack sequences. As an example consider a connection between the initiator and the enemy. This can only happen if the enemy is present and is always revealed if we observe the running of the initiator with the enemy. However it may not represent an attack. In order to reduce the number of these useless interferences, it is good practice to choose events that does not involve the presence of the enemy. For example, we could just observe the running of the responder with the initiator and the commit of the initiator with the responder, as specified in the `#Observe` section above. However, this choice completely hides the activity of the enemy during the attack and makes more difficult to understand the attack. Moreover, by hiding the enemy activities new kinds of attack where the observability of the enemy activity is fundamental may not be revealed (see the last but one attack in Section 8). For this reason, in the case studies of Sections 7 and 8 we have decided to observe some events involving the enemy. In our opinion, this will help in better understanding the NI approach which is at the base of our analysis technique.

We can also check if there are some attacks on secrecy by simply specifying which are the values that should remain secret, i.e., that the enemy should not discover. For example, if K_{AB} represents a session key that we would like to keep secret, we can specify this through the following command:

```
#Secret
KAB;
```

This automatically adds to the SPA specification an observable action `E_knows` which is executed when one of the secret values is learnt by the enemy. Thus, if the enemy learns `KAB`, then it will execute action `E_knows_KAB`. Note that this does not change the real behaviour of the enemy. It simply makes observable the fact that the enemy is learning some value which has been previously declared as a secret. Indeed, this modification has the effect of making secrecy attacks directly observable, thus detectable through our NI-based analysis. In fact, if the enemy is able to learn a secret value, the consequent execution of an `E_knows` action will make the system non SNNI, since such an action can never be executed without the presence of the enemy. We will show a detailed example of this in Section 8.

6 A VSP Specification of the Enemy

In this section we show the VSP specification for the *most powerful* enemy, which is automatically generated by CVS and then translated into SPA. Let K be a set of data item representing the actual knowledge of the enemy. The general scheme of the enemy is described as a family of VSP processes:

```
Enemy(K)
Begin
   'activate_enemy(K). (
      ∑_{c∈Chan} c(data_list).Enemy(add_to_knowledge(K,data_list))
      +
      ∑_{c∈Chan,d∈∩D(K)} 'c(d).Enemy(K)
   )
End
```

where the function `add_to_knowledge` simply adds to the first parameter `K` the list of data items represented by the second parameter `data_list`. The actions `'activate_enemy(K)` are the high level actions that represent the abstraction of the enemy activity. Moreover, the deduction function $\mathcal{D}(K)$ returns the set of all the data items that can be constructed by manipulating the data items in K. The set of rules includes the following:

$$\frac{m \qquad (m_1,\ldots,m_n)}{(m,m_1,\ldots,m_n)}\ (\vdash_{list}) \qquad \frac{(m,m_1,\ldots,m_n)}{m}\ (\vdash_{head}) \qquad \frac{(m,m_1,\ldots,m_n)}{(m_1,\ldots,m_n)}\ (\vdash_{tail})$$

$$\frac{(m_1,\ldots,m_n) \qquad k}{\{m_1,\ldots,m_n\}_k}\ (\vdash_{enc}) \qquad \frac{\{m_1,\ldots,m_n\}_k \qquad k^{-1}}{(m_1,\ldots,m_n)}\ (\vdash_{dec})$$

It shows all the operations a process can do over messages besides communicating them. In particular it can combine messages obtaining a list (rule $\vdash_{list}$); it can extract messages from a list (rules

$\vdash_{head}$ and $\vdash_{tail}$); it can encrypt a list of messages $m_1, \ldots, m_n$ with a key k obtaining $\{m_1, \ldots, m_n\}_k$ and finally decrypt a list of messages of the form $\{m_1, \ldots, m_n\}_k$ only if it has the corresponding (inverse) key k^{-1} (rules $\vdash_{enc}$ and $\vdash_{dec}$). Of course, this deduction system can be made more sophisticated if it is necessary to model, for example, some particular encryption scheme. And indeed, the version implemented in the tool is richer (e.g., with rules for handling hash functions).

The initial enemy is thus represented by the process with the knowledge declared in sections `#Processes` and `#EnemyKnowledge`. For example, if we have declared `Enemy(E,NE)` in section `#Processes` and `Agent:A,B;` in section `#EnemyKnowledge` then we obtain an enemy with initial knowledge `K` = {`E, NE, A, B`}. The limitation on the knowledge of the enemy given in section `#EnemyKnowledge` basically modifies the function `add_to_knowledge`. For example, if we have specified `Memory:False;` in order to reduce the enemy size, we must generate an enemy which can only use the information obtained in the last intercepted message. This can be obtained by specifying the function `add_to_knowledge` in a way that when a new intercepted message is inserted into K, the previous intercepted message is indeed removed from K. The specification `Block:n` can be handled in a similar way.

Finally, if some value s has been specified as secret in section `#Secret`, the Enemy specification is modified as follows: every time a new message is intercepted through an action `c(data_list)`, if the secret value s is such that $s \notin \mathcal{D}(K)$ (i.e., the secret was not deducible from the actual knowledge K) and $s \in \mathcal{D}($`add_to_knowledge(K,data_list)`$)$ (i.e., the secret is now deducible), then an action `E_knows_s` is generated. Such an action indeed represents the fact that the enemy has learnt the secret value s.

7 The Woo and Lam one-way Authentication protocol

We consider the Woo and Lam one-way authentication protocol [42]. 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 an 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, that was already shown incorrect by Abadi (as reported in [44]). 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 (see also Figure 2):

$$
\begin{array}{llcll}
\text{Message 1} & A & \rightarrow & B & : A \\
\text{Message 2} & B & \rightarrow & A & : N_B \\
\text{Message 3} & A & \rightarrow & B & : K_{AS}(N_B) \\
\text{Message 4} & B & \rightarrow & S & : K_{BS}(A, K_{AS}(N_B)) \\
\text{Message 5} & S & \rightarrow & B & : K_{BS}(N_B)
\end{array}
$$

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

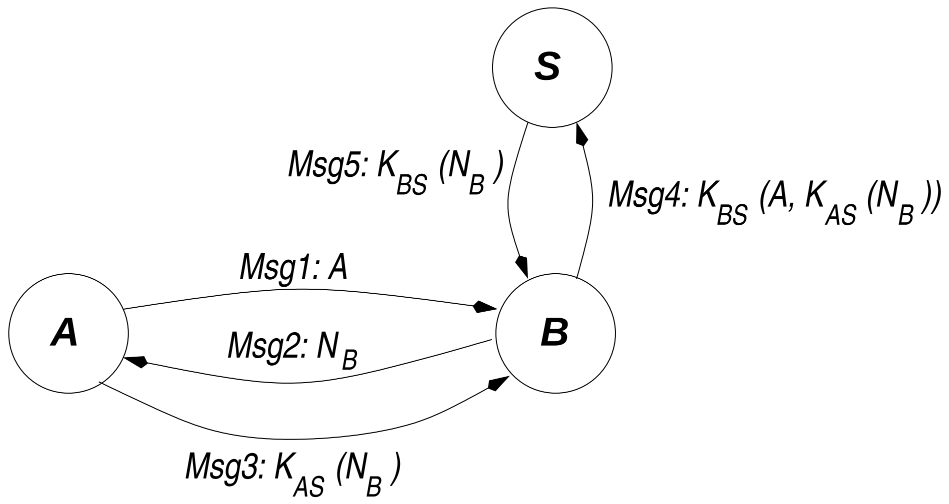

Figure 2: The Woo-Lam one-way authentication protocol.

checks that it contains the nonce N_B and, in such a case, he concludes that he 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.

7.1 First phase

Name declarations. 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 **SymKey**; as we have stated above, their identifiers in SPA code are automatically generated by the CVS compiler hence we do not need to 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.

```
msg1(Agent,Agent)
msg2(Nonce)
msg3(crypt(SymKey,Nonce))
msg4(crypt(SymKey,Agent,crypt(SymKey,Nonce)))
msg5(crypt(SymKey,Nonce))
i_running(Agent,Agent)  cond(\$0!=\$1)
r_running(Agent,Agent)
i_commit(Agent,Agent,Number)
r_commit(Agent,Agent,Number)
```

Besides message declaration, we have specified four actions that we will use to observe the protocol executions. i_running(A1,A2) (r_running(A1,A2)) corresponds to the starting of the protocol between agent A1 and agent A2, where A1 is the initiator (responder). [3] 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 7.5.

Roles declaration.

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

Processes definition. The bodies of the Initiator, Responder and Server 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 7.5.

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 `e_id` and one nonce `n_id`.

7.2 Second phase

Session definition. 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)
```

[3] Note that `msg1` has two parameters, intended to be the sender and the addressee identifiers. The second one, that is absent in the description of the protocol, has been added to bind the intended responder declared by the action `i_running` with the actual addressee of `msg1`. If not included, the event `i_running(A1,A2)` could be observed even if `A1` starts the protocol with another party.

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

Begin
        i_running(src,dst1).
        'msg1(src,dst1).
        msg2(nonr1).
        'msg3(enc(KA(src),nonr1)).
        'i_commit(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,dst).
        '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_commit(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 Initiator, Responder and Server for the Woo-Lam protocol.

Channel definition.

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

7.3 Third phase

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

It is possible to create a more powerful enemy, but in this case-study an enemy without memory is sufficient.

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.

7.4 Fourth phase

As we already stated, the Woo-Lam protocol has been proposed for obtaining the following one-way authentication property: "whenever a responder finishes an execution of the protocol, the initiator is in fact the principal claimed in the initial message". This corresponds, in our specification, to the following: "Every time we observe a `'r_commit(B,A)` action, then a `i_running(A,B)` action has been previously executed", since `'r_commit(B,A)` represents the correct termination of the responder B with the initiator A and `i_running(A,B)` represents the running of the initiator A with B. A natural choice of visible actions is thus:

```
#Observe
i_running(Initiator,Responder)
r_commit(Responder,Initiator,Number)
```

In the Woo-Lam protocol there are no values that should remain secret. Hence, we leave the section `#Secret` empty.

7.5 Interpreting the Interference Traces

By analyzing the VSP specification of the Woo-Lam protocol we find out that it is not SNNI. This means that the enemy is able to induce some new (possibly bad) behaviour of the protocol. All the execution traces induced by the enemy are called *interference traces* or simply *interferences*. When a protocol does not satisfy SNNI, the list of all the interferences is returned by the CoSeC tool. In this case we obtain only one interference trace:

```
'r_commit_B_A_1
```

This is clearly an attack to the required authentication property, since it represents the situation where B is correctly terminating the protocol convinced to communicate with A but A has never started the protocol with B. Note that in the SPA code, where no value passing is possible, the action parameters are embedded in the action name, e.g., `i_running(A,B)` becomes the SPA action `i_running_A_B`. This, of course, causes an increase in the specification dimension, when translated in SPA. As an example, for the Woo-Lam protocol the specification is translated to about 100 lines of SPA code (without considering the enemy which is about 700 lines of SPA code).

This interference just signals the presence of an attack but is not very informative. We can perform a more detailed analysis, where we observe more visible actions in order to have more information about this attack. We consider the following set of observable actions:

```
#Observe
r_running(Responder,Agent)
i_running(Initiator,Agent)
r_commit(Responder,Agent,Number)
i_commit(Initiator,Agent,Number)
```

Of course, the result of this analysis gives many interference traces. Some of these traces 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.

Since we already know which is the attack we are interested in, we can just consider the interferences that include `'r_commit_B_A_1` as a subtrace, without a preceding `i_run_A_B`. What we find out is basically a set of permutations of

```
i_running_A_E,'r_running_B_A,'i_commit_A_E_3,'r_commit_B_A_1
```

which clearly represents a *Man-in-the-middle attack*, i.e., A starts the protocol with E, and E exploits this session to impersonate A in a new session with B.

The number that appears on the end of a `commit` 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 `commit` labelled with a number we can see exactly in the SPA-code what the enemy has done. [4] In this way we can understand all the steps of the *Man-in-the-middle attack* that can be written as follows:

$$
\begin{array}{lrcll}
\text{Message } 1.a & A & \rightarrow & E & : A \\
\text{Message } 1.b & E(A) & \rightarrow & B & : A \\
\text{Message } 2.b & B & \rightarrow & E(A) & : N_B \\
\text{Message } 2.a & E & \rightarrow & A & : N_B \\
\text{Message } 3.a & A & \rightarrow & E & : K_{AS}(N_B) \\
\text{Message } 3.b & E(A) & \rightarrow & B & : K_{AS}(N_B) \\
\text{Message } 4.b & B & \rightarrow & S & : K_{BS}((A, K_{AS}(N_B)) \\
\text{Message } 5.b & S & \rightarrow & B & : K_{BS}(N_B)
\end{array}
$$

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 pretends 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!

Note that the specification we have given represents a single session of the protocol, i.e., every participant plays just one role at a time, but the attack found occurs in a multiple session. At a first glance this could seem strange, however it is simply due to the fact that the enemy is always

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

able to play different parallel roles. In this attack it plays the responder role in session a and the initiator one in session b.

This particular attack sequence is undocumented, even if it is not the first one found in the literature (see, e.g., [8] for a good survey on attacks to cryptographic protocols). However, it is the only one which can be carried out in a *single* session, where only one initiator and one responder are considered; all the other attacks reported in the literature, occurring in multiple sessions, were discovered through CVS by choosing an appropriate session definition and a suitable set of observable actions. This attack is not present in the first (not simplified) version of the Woo-Lam protocol [42]. However, it occurs also in other simplified versions of this protocol, e.g., the one studied in [3].

8 A detailed analysis of the Wide Mouthed Frog protocol

In the previous section we have seen that NI-based properties can be used to detect authentication attacks. Indeed, as we have already stated, NI-based properties simply reveal if the enemy is able to modify in some way the execution of the protocol, with respect to the chosen set of observable events. In principle, if we suitably choose this set of observable actions, we then should be able to detect whatever kind of attack. In this section we want to show through a simple example that with a unique analysis we are indeed able to detect quite different attacks, and also some interesting "mixture" of them (e.g., entity authentication with secrecy attacks).

Indeed, our intent is to show that the use of NI-based properties presents two interesting advantages with respect to classical analyses where a single particular property is studied: (i) it permits, in a single pass, to detect more attacks that can thus be repaired all at once; (ii) because of this "wide" range of action, the probability of detecting some "new" attacks on a protocol highly increases.

The example is intentionally very simple. We start with a simplified version of the Wide Mouthed Frog Protocol [6], which already presents a couple of different attacks. We then modify it in order to show how other attacks come out, and how they can all be observed as interference traces obtained by just one Non-Interference check.

8.1 First analysis

We consider two processes A and B sharing keys K_{AS} and K_{BS} with a trusted server S. In order to establish a secure channel with B, A sends a fresh key K_{AB} encrypted with K_{AS} to the server S. Then, the server decrypts the key and forwards it to B, this time encrypted with K_{BS}. Now B has the key K_{AB} and A can send a message M encrypted with K_{AB} to B. The protocol should guarantee that when B receives M, such a message has been indeed originated by A.

The protocol is composed of the following three messages (see also Figure 3):

$$\begin{array}{lllll}
\text{Message 1} & A & \rightarrow & S & : A, K_{AS}(B, K_{AB}) \\
\text{Message 2} & S & \rightarrow & B & : K_{BS}(A, K_{AB}) \\
\text{Message 3} & A & \rightarrow & B & : K_{AB}(M)
\end{array}$$

The main difference with respect to the original protocol is that here messages 1 and 2 do not contain timestamps. This makes the protocol sensitive to a replay attack (as already remarked in [2]).

The complete specification of this protocol in VSP is reported in Table 8.1. Note that we only observe the run of the initiator (`i_running`) and the commit of the responder (`r_commit`). This

```
#Names
Agent:   A,B,S,E;
Key:     KAB;
Message: M;

#ActionDec
message1(Agent,crypt(SymKey,Agent,Key))
message2(crypt(SymKey,Agent,Key))
message3(crypt(Key,Message))
i_running(Agent,Agent)
cond($0!=$1)
r_commit(Agent,Agent,Message)

#RolesDec
Initiator
Responder
Server
Enemy

#ProcessesDef
Initiator(src:Agent,k:Key,msg:Message)
Var
   Agent:  dst; cond(Responder,Enemy)
Begin
   i_running(src,dst).
   'message1(src,enc(KA(src),dst,k)).
   'message3(enc(k,msg)).
End

Responder(rsp:Agent)
Var
   Agent: ini; cond(Initiator,Enemy,Responder)
   Key:  ksess;
   Message:  msg2;
Begin
   message2(dec(KA(rsp),ini,ksess)).
   message3(dec(ksess,msg2)).
   'r_commit(rsp,ini,msg2)
End
```

```
Server(srv:Agent)
Var
   Agent:  src2; cond(Initiator,Enemy)
   Agent:  src3; cond(Initiator,Enemy)
   Agent:  res2; cond(Responder, Enemy)
   Key:  k2;
Begin
   message1(src3,dec(KA(src2),res2,k2)).
   if (src2=src3)
      'message2(enc(KA(res2),src2,k2))
   endif.
End

Enemy(enm:Agent)
Begin
End

#Processes
Initiator(A,KAB,M)
Responder(B)
Server(S)
Enemy(E)

#RestrictionOn
channel(message1,message2,message3)

#EnemyKnowledge
Agent:   All;
Memory:  False;

#Observe
i_running(Initiator,Agent)
r_commit(Responder,Agent,Message)
```

Table 2: The VSP specification for the Wide Mouthed Frog protocol.

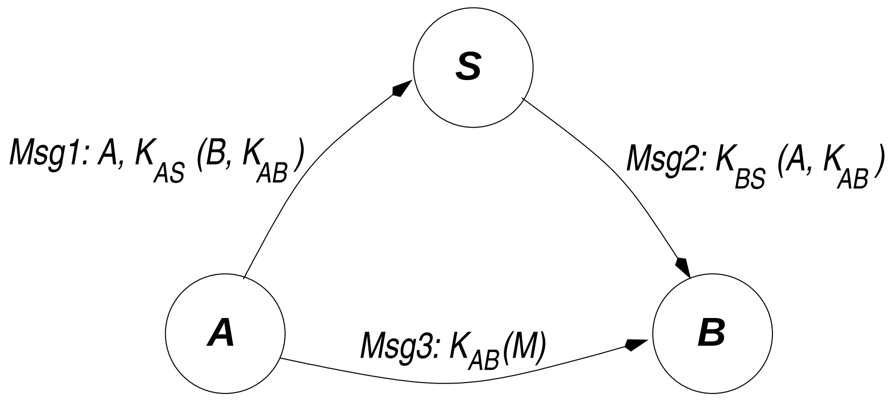

Figure 3: The simplified Wide Mouthed Frog protocol.

reflects the directionality of the protocol. Indeed, only the responder receives some data encrypted with the exchanged key and can thus expect that some security properties are guaranteed by the protocol itself, e.g., that M and K_{AB} indeed come from A (are authentic) and that no one has learnt M (secrecy). In order not to repeat the analysis twice, as done in the previous section, we decide to observe also the enemy activity from the beginning. This is specified by the parameter `Agent` inside action `i_running` and `r_commit`. Interestingly enough, later on in this Section, when analysing the third variant of this protocol, we will see an attack that is revealed only because the activity of the enemy is observable.

Here we have only specified a single session. Indeed in the `#Processes` section we have only one Initiator (A) and one Responder (B). If we analyze this protocol we find out only one interference trace:

```
i_running_A_E
```

which simply represents the fact that, in the presence of the enemy, A can also start a session with it. This cannot represent an attack on B since we have no commits by B at all.

We can now refine the analysis by specifying a more complex session. A good idea is to add more instances of the responder in order to observe if the enemy is able to exploit one protocol session, started with one particular responder, in order to cheat someone else.

We consider the following session:

```
#Processes
Initiator(A,KAB,M)
Responder(B)
Responder(B)
Responder(A)
Server(S)
Enemy(E)
```

We have B playing the Responder role twice and A that is both Initiator and Responder. The analysis of this session gives some interesting results. Below we report all the interference traces we have obtained:

```
(1) i_running_A_E,'r_commit_A_E_M
(2) i_running_A_B,'r_commit_B_A_M,'r_commit_B_A_M
```

```
(3) i_running_A_B,'r_commit_A_B_M
(4) i_running_A_B,'r_commit_A_B_M,'r_commit_B_A_M
(5) i_running_A_B,'r_commit_B_A_M,'r_commit_A_B_M
(6) i_running_A_B,'r_commit_A_B_M,'r_commit_B_A_M,'r_commit_B_A_M
(7) i_running_A_B,'r_commit_B_A_M,'r_commit_A_B_M,'r_commit_B_A_M
(8) i_running_A_B,'r_commit_B_A_M,'r_commit_B_A_M,'r_commit_A_B_M
```

We can interpret these traces as follows:

(1) does not represent an attack: A runs the protocol with E, then E starts again the protocol with A and A commits on M as a responder. E is doing nothing bad, indeed it can use M as it wants after it has completed the first session with A.

(2) is a *replay attack*. Indeed we see that A starts the protocol with B once, while B commits as a responder with A on message M twice. This means that the enemy is able to *replay* the first session getting B to commit a second time on the same message. We can see with a detailed analysis (using the NUMB built-in and examining the SPA-code in detail) that the replay sequence is the following:

Message 1	A	$\rightarrow$	S	:	$A, K_{AS}(B, K_{AB})$	
Message 2	S	$\rightarrow$	B	:	$K_{BS}(A, K_{AB})$	E eavesdrops and replays this
Message 2'	$E(S)$	$\rightarrow$	B	:	$K_{BS}(A, K_{AB})$	
Message 3	A	$\rightarrow$	B	:	$K_{AB}(M)$	E eavesdrops and replays this
Message 3'	$E(A)$	$\rightarrow$	B	:	$K_{AB}(M)$	

Note that B successfully commits on message M on both sessions. Note also that the parallel session starts with message $2'$, i.e., there is no message $1'$. Indeed, this is an attack just because A never starts a second session with B (message $1'$) who is instead committing twice with A. In order to do this the enemy does not even need that A starts a new session with someone else (e.g., with E). Of course, B is not able to discover whether message $1'$ has been executed or not, since it is sent by A to the server. In the next attack also a symmetric situation happens: message 2 is not executed and A is not able to discover this.

(3) is a *parallel session* attack. As a matter of fact, the trace shows that A starts the protocol as an initiator with B and after that commits as a responder with B. This is an attack for sure since B is not acting as an initiator at all. The detailed sequence is the following:

Message 1	A	$\rightarrow$	$E(S)$	:	$A, K_{AS}(B, K_{AB})$
Message 2'	$E(S)$	$\rightarrow$	A	:	$K_{AS}(B, K_{AB})$
Message 3	A	$\rightarrow$	$E(B)$	:	$K_{AB}(M)$
Message 3'	$E(B)$	$\rightarrow$	A	:	$K_{AB}(M)$

This attack exploits a symmetry in the message structure (as typically done in parallel session attacks). Indeed, in order to generate message $2'$, the enemy uses the encrypted part of message 1. As a matter of fact, note that encrypted parts of messages 1 and 2 of the protocol have exactly the same structure.

In particular, A starts a session with B but the enemy intercepts the first message. Then E impersonates the server by sending message $2'$ to A. This message represents a request of communication from B to A. When A sends the last message of the first session to B, the enemy uses it to conclude the parallel session where it (successfully) pretends to be B.

Note that this attack can also be carried out in two sequential sessions as follows:

$$\begin{array}{llcll} \text{Message 1} & A & \rightarrow & E(S) & : A, K_{AS}(B, K_{AB}) \\ \text{Message 3} & A & \rightarrow & E(B) & : K_{AB}(M) \\ \text{Message 2}' & E(S) & \rightarrow & A & : K_{AS}(B, K_{AB}) \\ \text{Message 3}' & E(B) & \rightarrow & A & : K_{AB}(M) \end{array}$$

However in order to perform this attack, the enemy needs to store both messages of the first session. As a matter of fact, this attack sequence is detected by CVS if we specify `Memory: True` in the `#Enemyknowledge` section.

(4) and (5) are basically the *parallel session* attack (3) with a correct termination of B. The enemy can obtain this behaviour by simply eavesdropping messages 1 and 3 instead of intercepting them. In this way, the (correct) run between A and B terminates successfully.

(6), (7) and (8) are the parallel session attack of (3) together with the replay attack of (2). This is interesting, since it is in some sense the "merging" of two different attacks. We see that the enemy is able to get A to commit as a responder and then get B to commit twice. This can be obtained through a further extension of (4).

We can check if there are also some attacks on secrecy by simply specifying which are the values that should remain secret, i.e., that the enemy should not discover. In our example, K_{AB} and M are of course sensible values that we would like to keep secret. Note that it is sufficient to observe if M is learnt by the enemy. As a matter of fact, if the enemy is able to learn K_{AB} then it will certainly be able to learn M by intercepting and decrypting the last message of the protocol. We specify it as follows:

```
#Secret
M;
```

With this additional specification, our analysis gives out the following additional trace:

```
i_running_A_E,'Eknows_M
```

So the enemy is able to learn M. However in this trace we note that A is indeed starting the protocol with E which becomes, in a sense, authorized to learn M (and K_{AB}). This does not represent an attack to secrecy since A is intentionally starting the protocol with E. We will see in the following how a secrecy attack can be observed.

8.2 Second analysis

Now, we are interested in analyzing more in detail the structure of the protocol. In particular, we will try to understand why the encryption of the identifiers (in messages 1 and 2) is important. Indeed, it is possible to see that it is necessary in order to guarantee some security properties.

For example, in message 1 the identifier of B is encrypted together with the key. If we want to understand why this is necessary, we can simply modify such a message by sending B as plaintext as follows:

$$\text{Message 1} \quad A \;\rightarrow\; S \;:\; A, B, K_{AS}(K_{AB})$$

Now, we check if this modified protocol suffers from some new attack. We find out the following traces (we are still checking the secrecy of M):

```
(1) i_running_A_B,'Eknows_M
(2) i_running_A_E,'Eknows_M
(3) i_running_A_B,'r_commit_A_A_M
(4) i_running_A_B,'r_commit_B_A_M,'r_commit_B_A_M
(5) i_running_A_E,'r_commit_A_A_M
(6) i_running_A_E,'r_commit_B_A_M
(7) i_running_A_E,'r_commit_B_A_M,'r_commit_B_A_M
```

We interpret the traces as follows:

(1) This is a *secrecy attack*: A starts the protocol with B but the enemy learns M. In order to obtain this, the enemy simply substitutes its identifier in the first message as follows:

$$\begin{array}{lrcll} \text{Message 1} & A & \rightarrow & E(S) & : A, B, K_{AS}(K_{AB}) \\ \text{Message 1}' & E(A) & \rightarrow & S & : A, E, K_{AS}(K_{AB}) \\ \text{Message 2}' & S & \rightarrow & E & : K_{ES}(A, K_{AB}) \\ \text{Message 3} & A & \rightarrow & E(B) & : K_{AB}(M) \end{array}$$

By message $2'$ the enemy learns K_{AB} and, by message 3 which is addressed to B, it finally learns M.

(2) This does not represent an attack since A is starting the protocol with E.

(3) This is a strange *parallel session attack* similar to the one we have already analyzed in the previous analysis. Here the enemy is able to exploit a session initiated by A in order to get A to commit as a responder with herself. This can be done as in (1) by substituting B with A. It is clear that a commit of an entity with itself should be avoided in a real implementation of the protocol, however if this is not done it could obviously lead to unexpected behaviours of the protocol. Note that if we are not interested in observing these attacks we can simply impose the condition `cond($0!=$1)` to the `r_commit` action.

(4) This is the replay attack we have already analyzed in the previous analysis.

(5) This is exactly like (3) but A is starting the protocol with E.

(6) This is an *entity authentication attack*: B successfully terminates the protocol convinced it is communicating with A but A has never started the protocol with B. This can be done as follows:

$$\begin{array}{lrcll} \text{Message 1} & A & \rightarrow & E(S) & : A, E, K_{AS}(K_{sess}) \\ \text{Message 1}' & E(A) & \rightarrow & S & : A, B, K_{AS}(K_{sess}) \\ \text{Message 2}' & S & \rightarrow & B & : K_{BS}(A, K_{sess}) \\ \text{Message 3} & A & \rightarrow & E & : K_{sess}(M) \\ \text{Message 3}' & E(A) & \rightarrow & B & : K_{sess}(M) \end{array}$$

Basically, A starts the protocol with E which is able to start a new session with B by simply substituting its identifier with B in the first message. Then, when A sends the last message to E, it forwards the message to B which is then convinced it is communicating with A.

(7) This is a combination of (4) and (6).

We can observe that the sending of a plaintext identifier in the first message basically induces three important attacks: the first one on secrecy of M and the other two on authentication. The replay attack we have found in the previous analysis is still possible, while the parallel session one is not possible anymore. This is due to the fact that by modifying the first message we have eliminated the symmetry between messages 1 and 2.

8.3 Third analysis

Now, we analyze what happens if we eliminate the encryption of the identifier in message 2 as follows:

$$\text{Message 2} \quad S \ \rightarrow \ B \ : A, K_{BS}(K_{AB})$$

while keeping B encrypted in Message 1, as in the original version. We obtain the following traces:

```
(1) i_running_A_E,'Eknows_M
(2) i_running_A_B,'r_commit_B_A_M,'r_commit_B_A_M
(3) i_running_A_B,'r_commit_B_E_M
(4) i_running_A_B,'r_commit_B_A_M,'r_commit_B_E_M
(5) i_running_A_B,'r_commit_B_E_M,'r_commit_B_A_M
(6) i_running_A_B,'r_commit_B_E_M,'r_commit_B_E_M
```

with the following interpretation:

(1) is not an attack, as above.

(2) is the usual replay attack.

(3) is a particular attack on authentication. Indeed B is convinced it is communicating with E but A is starting the protocol with B. As remarked in [1], this can be exploited by the enemy in order to get credit of something in place of A. Assume that M represents a request like "give me one million dollars". In this case, it is clearly important to avoid that B incorrectly believes that who is doing such a request is E. The attack can be simply obtained as follows:

$$\begin{array}{lrcll} \text{Message 1} & A & \rightarrow & S & : A, K_{AS}(B, K_{AB}) \\ \text{Message 2} & S & \rightarrow & E(B) & : A, K_{BS}(K_{AB}) \\ \text{Message 2}' & E(S) & \rightarrow & B & : E, K_{BS}(K_{AB}) \\ \text{Message 3} & A & \rightarrow & B & : K_{AB}(M) \end{array}$$

Note that this attack can be detected only if we observe the termination of the responder with the enemy, i.e., an event that involves the presence of the enemy. Indeed, this attack is not revealed through typical authentication analysis like, e.g., [43, 27, 24], where the correspondence of starts and terminations between the Initiator and the Responder only is observed.

(4) and (5) are combinations of (3) with a normal execution of the protocol.

(6) is a combination of (3) with (2). Better, it is attack (3) replayed twice.

Another interesting attack, suggested by an anonymous referee, can be revealed if the enemy has its own session key K_{EB} and message M_E. This can be specified in VSP by replacing the enemy specification with the following one:

```
Enemy(enm:Agent, k_enm:Key, m_enm:Message)
Begin
End
```

Moreover in the section `#Processes` we write `Enemy(E,KEB,ME)` instead of `Enemy(E)`. This is sufficient to add K_{EB} and M_E to the initial knowledge of the enemy. In general, it is a good idea to provide the enemy with an initial knowledge which is similar to the one given to the protocol agents.

If we repeat all the verifications with this new enemy, we find out a new attack only in the last version of the protocol, i.e. the one where we eliminated the encryption of the identifier in the second message. In particular, the interference trace representing the attack is the following:

```
'r_commit_B_A_ME
```

This represents an authentication attack where B is convinced that message M_E has been sent by A but A does not know M_E and, moreover, has never started a session with B. The corresponding attack sequence is the following:

$$\begin{array}{lrcll} \text{Message 1} & E & \rightarrow & S & : E, K_{ES}(B, K_{EB}) \\ \text{Message 2} & S & \rightarrow & E(B) & : E, K_{BS}(K_{EB}) \\ \text{Message 2}' & E(S) & \rightarrow & B & : A, K_{BS}(K_{EB}) \\ \text{Message 3}' & E(A) & \rightarrow & B & : K_{EB}(M_E) \end{array}$$

Other interesting attacks can be detected if we consider more than one instance of the server. This can be easily done in CVS by just specifying the server instances in the `#Processes` section. We do not give the details of these further analyses here.

We stop here the analysis of this protocol. Indeed, our aim was to show that the methodology and the analysis technique supported by CVS/VSP are quite flexible and permit to perform a detailed analysis of cryptographic protocols, detecting different attacks in a unique run of the tool.

9 Comparison with related work

As shown in Figure 1, our approach can be split into two fundamental phases: (i) specification of the protocol (using VSP and the associated compiler CVS) and (ii) verification (using the model-checker CoSeC which performs the NI-based analysis). In the following we relate our approach with those in the literature [24, 27, 28, 30, 32, 38] that, like ours, are essentially model-checkers. In doing this, we consider separately the two phases mentioned above. We omit instead a comparison with the approaches based on theorem proving such as, e.g., [35, 5].

As explained below, what mainly makes our approach appealing is the original correctness condition based on a Non-Interference property and the simplicity in the specification of the protocol (the latter shared with other similar approaches). On the other hand, our tool is to be improved in some respects in order to cope with the state space explosion problem and to cope with the problem of general correctness (i.e. for an unlimited number of different sessions) of the protocol. These problems will be discussed also in Section 10.

9.1 Protocol specification and compiler

Let us start with the approaches more similar to ours. The approach of [24] is very close; actually, we took inspiration from it for our work. Lowe defines an abstract notation, similar to VSP, and translates a specification into another process algebra, namely CSP [19]. The differences in the two compilers are inessential. Currently Casper includes a few features that CVS does not support yet (e.g., role specification in textbook-style, timed authentication and type confusion) that can be implemented also in CVS. On the other hand, VSP/CVS has a bit greater flexibility in the definition of the enemy and slightly better support for predefined types. Another high level notation that shares some similarities with VSP is CAPSL [30], the *Common Specification Protocol Specification Language*. Currently CAPSL is going to be used as a sort of universal language for many tools by means of a translator to an intermediate notation, called CIL, which can be directly used as input to them (see [9] for more details). Also the way processes are defined in [27] is rather similar to ours. The protocol is represented as a system consisting of a set of participants, each one playing a protocol role, each role specified as a sequence of actions in a CSP-like style.

Rather different approaches to specify protocols, at least in their original formulation, are the following. In the NRL Protocol Analyzer (NPA for short) [28], the specification language is PROLOG-based; it is expressive enough to specify complex protocol, even if it is low-level enough to make difficult to write and understand a protocol that involves many choices and/or long sequences of state transitions. It is necessary to provide some assistance to the specification writer in determining whether or not the specified protocol was the intended protocol. To solve this problem, recent work aims at defining a translator from CAPSL to the specification language of NPA.

Murϕ [11] is a general-purpose finite-state machine verification tool. The input language is a simple high-level notation with features familiar from conventional programming languages. Nothing specific to the application domain has been introduced to help the protocol specification. In [32] a cryptographic protocol is specified defining a set of global structures. Each role is described using one of the above structures, consisting essentially of a set of transition rules, enabled by boolean conditions on the values of the global variables in the current state. The concept of process is not primitive and to compose processes in parallel one has to make the union of the transition rules of the two roles. Communication takes place via shared variables.

9.2 Definition of the security properties and their verification

This is the part where our approach is largely different from other approaches. The closest ones are [24] and [27]; both introduce additional observable events in the protocol specification, as we also do, but with the limited aim to implement authentication checking using the correspondence idea introduced by Woo and Lam [43]. We have seen, for instance in the last variation on the Wide-Mouthed Frog protocol, that some authentication problems are not captured by their definition. Moreover, if other security properties are of interest, new specific definitions have to be provided and checked, while we can capture all of them at once by suitably choosing the observable events. On the other hand, the FDR tool underlying Casper is more efficient than CoSeC in managing the enemy as it implements the optimization technique of [36]. An advantage of [27] is that the intruder is not encoded into the model, hence avoiding one source of state space explosion.

In Murϕ [32] the security properties are specified as invariants, i.e., boolean conditions on global variables that have to be true in every reachable state. Like in our approach and in Lowe's one, Murϕ can only guarantee correctness of the scaled-down version of the protocol, where one specific (possibly multiple) session is actually verified; it cannot check the correctness of the protocol in general. In the Murϕ tool are implemented various states reduction rules, that make the analysis of large systems more effective than with CoSeC.

In the NRL Protocol Analyzer [28, 29], the user has to define an insecure state and then the

Analyzer explores the state space, by Prolog-based backward reasoning, in order to find out an initial state (hence an attack is found) or an unreachable state. The Analyzer makes no assumptions about the limits on the number of principals and about the number of nested applications of cryptographic functions. However, without suitable reductions of the potentially infinite state space, the model checker may not terminate. The state space can be reduced by proving inductive lemmas about the unreachability of classes of states, hence, relying heavily on the user during the verification much in the same way a theorem prover relies on the user to guide it during the search of a proof.

Finally, the recent tool Athena [38] introduces a logic on the Strand Space model [41] that is used to define security properties; then, the (backward search) model checking algorithm is particularly efficient as it takes advantage of the implicit partial order structure of the model (hence ignoring interleavings) and of the symbolic representations for data to obtain concise models. Nonetheless, the state space can be infinite and in such a case the tool allows the user to incorporate unreachability results from theorem provers.

10 Conclusion and future research

In this work we have presented a new notation, called VSP, which permits defining security protocols at a level more abstract than SPA. The VSP specifications are compiled to SPA code through the CVS compiler. The CVS/VSP technology has been developed with the main aim of supporting the Non-Interference based approach for the analysis of cryptographic protocols which has been first studied and shown flexible enough to reveal different (known) attacks in [16, 14].

The Non-Interference check can be seen as a sufficient condition to the absence of every kind of (not only authentication) attacks carried out by the defined enemy 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. For a specific session and a particular enemy, 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. 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 main advantages of VSP with respect to SPA are (i) the possibility of giving very compact value passing specifications and (ii) the high level of parameterization 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 starting the analysis with a simplified enemy, which is still generated automatically, and then proceeding 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 step by step approach is useful to understand how much power the enemy needs in order to be able to attack the protocol. Also the session can be made complex gradually, for example by starting with a single session it is very likely that the analysis will be feasible. Then, we can try with multiple sessions, in order to see if the protocol presents new attacks, e.g, replay and parallel session attacks. When the session is complex it is also useful to include in the specification of the protocol suitable details like, e.g., the identifiers of the addressees in the messages, which could reduce the number of useless actions where an entity receives a message not meant for him.

10.1 Validation of the Tool

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 in [6] and shown insecure in [39]; the Andrew protocol [37] with the attack presented in [23]; the Otway-Rees [34] shown flawed in [35] and the TMN protocol [40] with the attacks reported in [26]. All the attacks on these protocols presented in the literature have been captured by CVS. Moreover, the attack to the Woo-Lam protocol we have presented in this paper is undocumented in the literature, although the Woo-Lam protocol was already known to be flawed. Other new kinds of failure on flawed protocols have been found for EKE [4] and Station to Station [10]. In the TMN protocol, we have found a variant of a known attack.

In all of the checks on these protocols we have keenly exploited the high parameterization level of our tool, that has permitted obtaining reasonably fast analysis and detection of attacks. We are preparing a survey on all the protocols that have been analysed with CVS/CoSeC.

10.2 Human Effort and Performance

A typical VSP specification of a protocol is less that one hundred lines of code in length and it is very similar to a program written in a high level programming notation. This characteristic allows anyone who knows a bit of process algebra and has a bit of knowledge of cryptographic protocols, to get used to the VSP notation in a few hours.

The generation of the SPA specification from the VSP one usually takes few seconds. The critical part of the analysis in indeed the equivalence check performed by the CoSeC tool. As other approaches based on exhaustive state space searching, our approach suffers from the state space explosion problem due to parallel composition, where all the possible interleaving of execution sequences must be considered. For example, while for the Woo-Lam protocol analyzed in this paper, the analysis takes a few minutes, for protocols with 6-7 complex messages (with nested encryptions) we are not able to check multiple sessions. Typically, with more than four agents (usually two initiators and two responders) together with the enemy, the analysis slows down significantly, and can possibly run out of memory. We are working on an optimization of the Non-interference check which could hopefully reduce this problem as much as possible.

Actually, our check is only performed on protocols which have a finite-state labelled transition system, i.e., a finite number of sessions. This means that we still cannot reason about protocol correctness on every possible protocol session.

10.3 Extensions

As a future work, it would be useful to extend the CVS/CoSeC technology in three different aspects: specification, verification and attack reconstruction.

At the specification level it could be possible to extend the VSP language in order to deal with timed authentication checks and type confusion attacks. These extensions are indeed not problematic. We are also implementing a graphical interface that will reduce significantly the specification time (by allowing textbook-like specifications).

At the verification level we are presently optimizing the underlying NI check. This should allow us to check more complex sessions. Moreover, the technique for reducing the state-space of the enemy proposed in [36] could be included in the next release of the tool. We are also considering other techniques based on the introduction of some auxiliary information which could allow a more efficient and "local" analysis of some properties [13, 7]. We are also trying to enhance the tool with some features for cryptographic analysis in order to detect some attacks based on particular properties of the adopted cryptographic algorithms.

For the reconstruction of the attacks, it would be very useful to have a more detailed trace as output of the CoSeC tool. This will be included in the optimization of the NI-check we are developing. Moreover, it would be possible to implement an interference filter which basically extracts from the set of interferences given by CoSeC the ones that correspond to attacks to known properties.

Acknowledgements

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

References

[1] M. Abadi. "Two Facets of Authentication". In *Proceedings of 11th IEEE Computer Security Foundations Workshop*, pages 25–32. IEEE press, June 1998.

[2] M. Abadi and A. D. Gordon. "A Calculus for Cryptographic Protocols: The Spi Calculus". *Information and Computation 148*, 1:1–70, January 1999.

[3] M. Abadi and R. Needham. "Prudent Engineering Practice for Cryptographic Protocols". *IEEE Transactions on Software Engineering*, 22(1), January 1996.

[4] S. M. Bellovin and M. Merritt. "Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks". In *Proceedings of the 1992 IEEE Computer Society Conference on Research in Security and Privacy*, pages 72–84, 1992.

[5] D. Bolignano. "Formal Verification of Cryptographic Protocols". In *Procs ACM Conf. on Computer and Communication Security*, 1996.

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

[7] C. Bodei, P. Degano, R. Focardi, and C. Priami. Authentication via localized names. In *Proceedings of CSFW'99*, pages 98–110. IEEE press, 1999.

[8] 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.

[9] G. Denker and J. Millen. "CAPSL Intermediate Language". In *Proceedings of the FLoC Workshop on Formal Methods and Security Protocols*, Trento (Italy), July 1999.

[10] W. Diffie, P. C. van Oorschot, and M. J. Wiener. "Authentication and Authenticated Key Exchanges". *Design, Codes and Cryptography*, 2:107–125, 1992.

[11] D. L. Dill. "The Murϕ Verification System". In *Procs Eight Int.l Conf. on Computer Aided Verification (CAV'96)*, pages 390–393. Springer LNCS, 1996.

[12] R. Focardi. *"Analysis and Automatic Detection of Information Flows in Systems and Networks"*. PhD thesis, University of Bologna, November 1998.

[13] R. Focardi. "Using Entity Locations for the Analysis of Authentication Protocols". In *Proceedings of Sixth Italian Conference on Theoretical Computer Science (ICTCS'98)*, November 1998.

[14] 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.

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

[16] 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.

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

[18] R. Focardi and F. Martinelli. "A Uniform Approach for the Definition of Security Properties". In *Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99), LNCS 1708*, pages 794–813, Toulouse (France), September 1999. Springer-Verlag LNCS.

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

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

[21] R. Keller. "Formal Verification of Parallel Programs". *Communications of the ACM*, 19 (7):561–572, 1976.

[22] 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.

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

[24] G. Lowe. "CASPER: A Compiler for the Analysis of the security protocols". *Journal of Computer Security*, 6:53–84, 1998.

[25] 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.

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

[27] W. Marrero, E. Clarke, and S. Jha. "A Model Checker for Authentication 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.

[28] C. Meadows. "The NRL Protocol Analyzer: An Overview". *Journal of Logic Programming*, 26(2):113–131, 1996.

[29] C. Meadows. "Analysis od the Internet Key Exchange protocol using the NRL protocol analyzer". In *Procs IEEE Symposium on Security and Privacy*, pages 216–231. IEEE Press, 1999.

[30] J. Millen. "CAPSL: Common Authentication Protocol Specification Language". Technical Report MP 97B48, The Mitre Corporation, 1997.

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

[32] J. C. Mitchell, M. Mitchell, and U. Stern. "Automated Analysis of Cryptographic Protocols Using Murϕ". In *Proceedings of the 1997 IEEE Symposium on Research in Security and Privacy*, pages 141–153. IEEE Computer Society Press, 1997.

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

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

[35] Lawrence C. Paulson. "The Inductive Approach to Verifying Cryptographic Protocols". *Journal of Computer Security*, 6:85–128, 1998.

[36] A. W. Roscoe and M. H. Goldsmith. "The perfect spy for model-checking crytpo-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.

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

[38] Dwn X. Song. "Athena: a new efficient automatic checker for security protocol analysis". In *Proceedings 12th IEEE Computer Security Foundation Workshop, (CSFW'99), (P.Syverson Ed.)*, pages 192–202, Mordano (Italy), 1999. IEEE Press.

[39] 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.

[40] 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.

[41] F. J. Thayer, J. C. Herzog, and J. D. Guttman. "Strand Spaces: Why is a Security Protocol Correct?". In *Proceedings of the 1998 Symposium on Security and Privacy.* IEEE Computer Society Press, May 1998.

[42] T.Y.C. Woo and S.S. Lam. "Authentication for distributed systems". *Computer*, 25:10–10, March 1992.

[43] T.Y.C. Woo and S.S. Lam. "A Semantic Model for Authentication Protocols". In *Proceedings of 1993 IEEE Symposium on Security and Privacy*, pages 178–194, 1993.

[44] T.Y.C. Woo and S.S. Lam. "A lesson on authentication protocol design". *Operating system review*, pages 24–37, 1994.

A SPA and VSPA

In this appendix we report from [16, 12] the syntax and semantics of SPA and VSPA languages, the latter being the core language actually used in the definition of the roles.

A.1 The Model

The model used to represent systems is the labelled transition system model [21]. A labelled transition system is essentially an automaton, although it may have infinitely many states.

Definition A.1 *A* labelled transition system (LTS) *is a triple* $(S, T, \rightarrow)$ *such that:*

- S *is a set of states,*
- T *is a set of labels (actions),*
- $\rightarrow \subseteq S \times T \times S$ *is a set of labelled transitions.* ■

$(S_1, \alpha, S_2) \in \rightarrow$ (or equivalently $S_1 \stackrel{\alpha}{\rightarrow} S_2$) means that the system can move from the state S_1 to the state S_2 through the action α. A LTS is *finite* if it has a finite number of states and transitions.

A.2 The SPA Language

The *Security Process Algebra* (SPA for short) is a slight extension of Milner's CCS [31], where the set of visible actions is partitioned into high level actions and low level ones 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; $\mathcal{L} = I \cup O$ is called the set of *visible* actions, ranged over by α.
- A function $\bar{\cdot} : (I \rightarrow O) \cup (O \rightarrow I)$, extended to sets as follows: $\bar{L} \stackrel{\text{def}}{=} \{\bar{a} : a \in L\}$, with $L \subseteq \mathcal{L}$.
- Two sets Act_H and Act_L of high and low level actions which are closed with respect to function $\bar{\cdot}$ (i.e., $\overline{Act_H} = Act_H$, $\overline{Act_L} = Act_L$); moreover they form a covering of $\mathcal{L}$ and they are disjoint (i.e., $Act_H \cup Act_L = \mathcal{L}$, $Act_H \cap Act_L = \emptyset$).
- 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* (or *processes*) is defined (as for CCS) as follows:

$$E ::= \underline{0} \;\big|\; \mu.E \;\big|\; E + E \;\big|\; E|E \;\big|\; E \setminus L \;\big|\; E/L \;\big|\; Z$$

where $L \subseteq \mathcal{L}$ and $f : Act \rightarrow Act$ is such that $f(\bar{\alpha}) = \overline{f(\alpha)}$, $f(\tau) = \tau$. For every constant Z there must be the corresponding definition: $Z \stackrel{\text{def}}{=} E$.

Intuitively, we have that $\underline{0}$ is the empty process, which cannot do any action; $\mu.E$ can do an action μ and then behaves like E; $E_1 + E_2$ is the alternative composition choose [5] of 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.

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

Prefix	$$\frac{-}{\mu.E \xrightarrow{\mu} E}$$
Sum	$$\frac{E_1 \xrightarrow{\mu} E_1'}{E_1 + E_2 \xrightarrow{\mu} E_1'} \qquad \frac{E_2 \xrightarrow{\mu} E_2'}{E_1 + E_2 \xrightarrow{\mu} E_2'}$$
Parallel	$$\frac{E_1 \xrightarrow{\mu} E_1'}{E_1 \vert E_2 \xrightarrow{\mu} E_1' \vert E_2} \qquad \frac{E_2 \xrightarrow{\mu} E_2'}{E_1 \vert E_2 \xrightarrow{\mu} E_1 \vert E_2'} \qquad \frac{E_1 \xrightarrow{\alpha} E_1' \quad E_2 \xrightarrow{\bar{\alpha}} E_2'}{E_1 \vert E_2 \xrightarrow{\tau} E_1' \vert E_2'}$$
Restriction	$$\frac{E \xrightarrow{\mu} E'}{E \setminus L \xrightarrow{\mu} E' \setminus L} \ \text{if } \mu \notin L \cup \bar{L}$$
Hiding	$$\frac{E \xrightarrow{\mu} E'}{E/L \xrightarrow{\mu} E'/L} \ \text{if } \mu \notin L \qquad \frac{E \xrightarrow{\mu} E'}{E/L \xrightarrow{\tau} E'/L} \ \text{if } \mu \in L$$
Constant	$$\frac{E \xrightarrow{\mu} E'}{A \xrightarrow{\mu} E'} \ \text{if } A \stackrel{\text{def}}{=} E$$

Table 3: The operational rules for SPA.

A.3 Operational Semantics of SPA

Let $\mathcal{E}$ be the set of SPA agents, ranged over by E and 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. Note that $\mathcal{E}_H \cup \mathcal{E}_L \subset \mathcal{E}$, i.e., there exist systems that execute both high and low level actions allowing communications between the two levels.

The operational semantics of SPA is the LTS $(\mathcal{E}, Act, \rightarrow)$, where the states are the terms of the algebra and the transition relation $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is defined by structural induction as the least relation generated by the axioms and inference rules reported in Table 3. The operational semantics for an agent E is the subpart of the SPA LTS reachable from the initial state E. We denote with $\mathcal{E}_{FS}$ the set of all the SPA agents with a finite LTS as operational semantics.

We recall now trace equivalence, according to which two agents are equivalent if they have the same execution traces.

Definition A.2 *The expression $E \stackrel{\alpha}{\Longrightarrow} E'$ is a shorthand for $E(\xrightarrow{\tau})^* E_1 \xrightarrow{\alpha} E_2 (\xrightarrow{\tau})^* E'$, where $(\xrightarrow{\tau})^*$ denotes a (possibly empty) sequence of τ labelled transitions. Let $\gamma = \alpha_1 \ldots \alpha_n \in \mathcal{L}^*$ be a sequence of actions; then $E \stackrel{\gamma}{\Longrightarrow} E'$ if and only if there exist $E_1, E_2, \ldots, E_{n-1} \in \mathcal{E}$ such that $E \stackrel{\alpha_1}{\Longrightarrow} E_1 \stackrel{\alpha_2}{\Longrightarrow} \cdots \stackrel{\alpha_{n-1}}{\Longrightarrow} E_{n-1} \stackrel{\alpha_n}{\Longrightarrow} E'$. We say that E' is* reachable *from E when $\exists \gamma : E \stackrel{\gamma}{\Longrightarrow} E'$ and we write $E \Longrightarrow E'$.* ■

Definition A.3 *For any $E \in \mathcal{E}$ the set $T(E)$ of* traces *associated with E is defined as follows: $T(E) = \{\gamma \in \mathcal{L}^* \mid \exists E' : E \stackrel{\gamma}{\Longrightarrow} E'\}$. E shows less traces that F (notation $E \leq_T F$) if and only if $T(E) \subseteq T(F)$. Moreover, E and F are* trace equivalent *(notation $E \approx_T F$) if and only if $E \leq_T F$ and $F \leq_T E$, i.e., $T(E) = T(F)$.* ■

A.4 Value-Passing SPA

In this section we briefly present a value-passing extension of "pure" SPA (VSPA, for short). VSPA is very similar to the value-passing CCS introduced in [31]. 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:

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

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. The arity of actions and constants is given by function ari. This function returns a tuple of sets (called $Sorts$) that represent the ranges of the parameters for the specific action or constant considered. For example, $ari(a) = (S_1,\ldots,S_n)$ means that action a has n parameters with ranges $S_1,\ldots,S_n$, respectively.

It is also necessary to define constants as follows: $A(x_1,\ldots,x_m) \stackrel{\text{def}}{=} E$ where E is a VSPA agent which may contain no free variables except $x_1,\ldots,x_m$, which must be distinct. As in [31] 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 recursively on the structure of agents. Note that, since we have no free variable all the value and boolean expressions can be calculated; we will write $\widehat{b}$ and $\widehat{e}$ for the value obtained by evaluating a boolean expression b and a value expression e respectively.

We will use the notation $E\{a/b\}$ to represent agent E with all the occurrences of b replaced by a. We will also use $\mathcal{E}^+$ to denote the set of VSPA agents. For each agent $E \in \mathcal{E}^+$ without free variables, its translation $[\![E]\!]$ is given in Table 4 where $ari(a) = S_1 \ldots S_n$; $\widehat{L} = \{l_{v_1,\ldots,v_n} : l \in L, ari(l) = S_1 \ldots S_n, v_i \in S_i, \forall i \in [1,n]\}$ is the set of the translations of actions in L. Furthermore, the single definition $A(x_1,\ldots,x_m) \stackrel{\text{def}}{=} E$ with $ari(A) = S_1 \ldots S_m$, is translated to the set of equations:

$$A_{v_1,\ldots,v_m} = [\![E\{v_1/x_1,\ldots,v_m/x_m\}]\!]; v_i \in S_i \forall i \in [1,m]\}$$

$E \in \mathcal{E}^+$	$[\![E]\!] \in \mathcal{E}$
$\underline{0}$	$\underline{0}$
$a(x_1, \ldots, x_n).E$	$\sum_{i \in [1,n], v_i \in S_i} a_{v_1,\ldots,v_n}.[\![E\{v_1/x_1, \ldots v_n/x_n\}]\!]$
$\bar{a}(e_1, \ldots, e_n).E$	$\bar{a}_{\widehat{e_1},\ldots,\widehat{e_n}}.[\![E]\!]$
$\tau.E$	$\tau.[\![E]\!]$
$E_1 + E_2$	$[\![E_1]\!] + [\![E_2]\!]$
$E_1 \vert E_2$	$[\![E_1]\!] \vert [\![E_2]\!]$
$E \setminus L$	$[\![E]\!] \setminus \widehat{L}$
E/L	$[\![E]\!]/\widehat{L}$
$A(e_1, \ldots, e_n)$	$A_{\widehat{e_1},\ldots,\widehat{e_n}}$
if b **then** E	$\begin{cases} [\![E]\!] & \text{if } \widehat{b} = True \\ \underline{0} & \text{otherwise} \end{cases}$

Table 4: Translation of VSPA to SPA.