AVISPA

www.avispa-project.org

IST-2001-39252

Automated Validation of Internet Security Protocols and Applications

Deliverable D3.3: Session Instances

Abstract

In this deliverable, we describe a number of techniques that we have devised in order to specify parallel protocol sessions and thereby accelerate search and increase the performance and efficiency of our protocol validation tool.

Deliverable details

Deliverable version: *1.0*
Date of delivery: *12.01.2004*
Classification: *public*
Person-months required: *10*
Due on: *31.12.2003*
Total pages: *15*

Project details

Start date: *January 1st, 2003*
Duration: *30 months*
Project Coordinator: *Alessandro Armando*
Partners: *Università di Genova, INRIA Lorraine, ETH Zürich, Siemens AG*

Information Society Technologies

Project funded by the European Community under the *Information Society Technologies* Programme (1998-2002)

Contents

1 Introduction

In the context of security protocol analysis, we define a *problem* as the specification of a protocol paired with a security property that the protocol should ensure. To conduct an *experiment* (that is, an attempt to either prove that the protocol satisfies the desired security property or to find a counterexample showing that the property is violated), we also require a concrete instantiation of the protocol to be analysed.

Every honest agent is modelled as a process that can participate in an unbounded number of parallel *sessions* (or *session instances*), i.e. executions of the protocol, playing in any of the roles. To constrain search, we can bound this infinite set of possible protocol instantiations by specifying *scenarios*, which are finite sets of sessions, i.e. instantiations of roles with agent names, where session numbers (IDs) are used to distinguish parallel sessions between the same agents.

When solving a problem, it is often the case that multiple protocol sessions must be considered in parallel in order to discover an attack. A classic example is the man-in-the-middle attack on the Needham-Schroeder Public Key Protocol NSPK [4, 9], in which an honest agent starts a session with the intruder, who then starts a second session in parallel with another honest agent. Even such a simple protocol, then, reveals a flaw if two parallel sessions are allowed, while the very same flaw goes undetected if only one session is run. Hence, in order to validate protocols it is mandatory to consider as many parallel sessions as possible, potentially infinitely many. In this deliverable, we describe a number of techniques we have devised to specify parallel sessions in a security problem.[1] The problem is not trivial: as the above example shows, too few sessions might lead to incompleteness, i.e. there is an attack to the protocol but we cannot detect it; on the other side, in model-checking approaches such as ours, even a small number of parallel sessions usually leads to a blowup in the search space.

We proceeded in two ways. We began by devising and implementing an algorithm to automatically alter a HLPSL protocol specification to take into account several sessions in parallel, avoiding a number of useless or redundant ones; we describe this work in Section 3. We then devised a technique that exploits the lazy intruder in order to deal with parallel sessions *symbolically* at the IF level; this technique, which we implemented in the OFMC back-end of the AVISPA-tool, is described in Section 4.

In what follows, we will sometimes use terms coming from the grammars of our specification languages HLPSL and IF, which are given in detail in our previous deliverables [2, 3]. Moreover, we will refer to a single "run" of a protocol, that is, a session played by two or more actual principals, as to a *single session instance* (*SSI*); when more than one SSI is played in parallel, we will talk about a *parallel session instance* (*PSI*).

[1] An interesting, related research question is that of the identification of the smallest number of parallel sessions needed in order to detect an attack to a particular protocol. This question goes beyond the scope of this deliverable and is thus left to future investigations.

2 Background

Roles in HLPSL are parametrised by a set of variables. A *session* in HLPSL can thus be seen as an instantiation of the roles required for a single protocol run: that is, an assignment of values to all the parameters of the required roles.

We take as an example the SHARE protocol [1]. This is a two party protocol, so its formalisation in HLPSL will require two basic roles. The initiator role `SHARE_Init` is shown below (the responder role, which we call `SHARE_Resp`, is defined analogously):

```
role SHARE_Init ( A,B : agent,
                  Ka,Kb : public_key,
                  Snd, Rcv : channel(dy)) played_by A def=
  exists State: nat,
         Na: text(fresh),
         Nb   : text,
         K    : message
  owns   Snd
  init  State = 0
  accept State = 2
  knowledge(A) = {A,B,Ka,Kb,inv(Ka)}

  transition
   1. State=0 /\ Rcv(start) =|>
         Snd({Na'}Kb)
      /\ State'=1

   2. State=1 /\ Rcv({Nb'}Ka) =|>
      /\ State'=2
      /\ K'=Na.Nb'
end role
```

An execution of the SHARE protocol requires one agent playing the initiator role and one playing the responder role. A session or SSI, then, has the form[2]

```
     SHARE_Init(A,B,Ka,Kb,SA,RA)
  /\ SHARE_Resp(B,A,Kb,Ka,SB,RB)
```

for agents `A` and `B` with associated public keys `Ka` and `Kb`. These might be assigned concrete values, in which case we refer to the SSI as a *ground session* (see Section 3), or could be left as uninstantiated variables, in which case we call it a *symbolic session* (see Section 4).

A set of parallel sessions (a PSI) defines a *scenario* we wish to investigate. For instance, for two-party protocols, a typical scenario for detecting man in the middle attacks involves

[2]We assume the `SHARE_Resp` role takes the same arguments but with the agents and public keys in reverse order with respect to the initiator role.

three sessions: one in which two honest agents communicate, one in which an honest initiator talks to a dishonest responder, and one in which the intruder initiates a session with an honest responder. For concrete agents `a`, `b`, and the intruder `i` possessing public keys `ka`, `kb`, and `ki`, respectively, this scenario might look as follows for the SHARE example[3]:

```
   %% a talks with b
   SHARE_Init(a,b,ka,kb)
/\ SHARE_Resp(b,a,kb,ka)

   %% a talks with i
   SHARE_Init(a,i,ka,ki)
/\ SHARE_Resp(i,a,ki,ka)

   %% i talks with b
   SHARE_Init(i,b,ki,kb)
/\ SHARE_Resp(b,i,kb,ki)
```

SSIs can thus be seen as "permutations" of assignments of an agent to a role. For the set of 3 agents described above playing the 2 roles of the SHARE protocol, there are therefore $3^2 = 9$ possible SSIs. This, however, includes some possibilities which may be redundant, as well as the uninteresting case in which the intruder talks to himself. The question of how to specify scenarios and determine which SSIs are actually relevant is discussed in the following section.

3 Parallel sessions

In this section, we consider two alternatives of increasing sophistication and power: first, manual specification of PSIs, which is common in many tools, e.g. in previous versions of HLPSL and OFMC or CAPSL/CIL [6]; second, automatic generation of symbolic parallel sessions and PSIs, which has been developed in the context of the AVISPA-tool back-end Cl-atse.

3.1 Manual specification

The first alternative is that the user explicitly describes the PSI under which the protocol should be checked. This may be done by specifying a finite list of SSIs, i.e. by specifying a finite list of instantiations of the roles of the protocol with agent names, where `i` denotes the intruder and all other agents, denoted by `a`, `b`, ..., are honest. Note that the intruder can always send messages under any identity. However, an SSI between an honest agent

[3]For brevity, we ignore the channel arguments of the two roles here. We assume that each role instance is passed two distinct channels, and no channels are reused.

and the intruder explicitly models an honest agent who runs a protocol with a dishonest or corrupted agent. This also reflects that we do not distinguish several intruders and corrupted parties, but assume that they all work together and can thus be merged into one intruder.

We continue our example of the SHARE protocol from the previous session. To specify PSIs, one could specify a further role, `SHARE`, as shown here:

```
role SHARE (I: (agent,agent,public_key,public_key) set) def=
 composition
  /\_{in((A,B,Ka,Kb),I)}
     SHARE_Init(A,B,Ka,Kb) /\
     SHARE_Resp(B,A,Kb,Ka)
end role
```

This role takes as input a set of SSIs (in this case called `I`), each one characterised by two agents and two public keys, and "launches" them all in parallel using the composition operator — a logical conjunction, in line with the semantics of TLA upon which HLPSL is based (see [2, 8]). Conjunction is used here to gather together several SSIs into a PSI.

The top-level role `Environment` (cf. [2]), is then defined as:

```
role Environment() def=
 knowledge(i) = {a,b,ka,kb,ki,i}
 composition
  SHARE([(a,b,ka,kb),(a,i,ka,ki)])
end role
```

Here, `Environment` invokes a ground instance of `SHARE`, actually starting a PSI with two SSIs, in the first of which `a` executes a session with `b` (with public keys `ka` and `kb`), whereas in the second `a` executes a second session in parallel with the intruder `i` (with public keys `ka` and `ki`).

In this example, the AVISPA end-user manually specifies which SSIs to start, by defining an appropriate top-level role. The possibilities at the user's disposal are practically infinite — too many, actually. A mechanism by which the *appropriate* SSIs are started is needed here.

We can thus summarise by observing that manual specification of PSIs constitutes a basic mechanism to specify protocol analysis scenarios. However, it is unsatisfactory that the user must specify them manually. In practice, to avoid state-space explosion due to parallelism, it is desirable to iterate through many scenarios, with differing role instances. Support for searching among instances is called for here.

3.2 Automatic generation

The second alternative to tackle parallel sessions is to automatically generate PSIs. This may be accomplished, for a given number n of SSIs per PSI and a set of roles, by generating

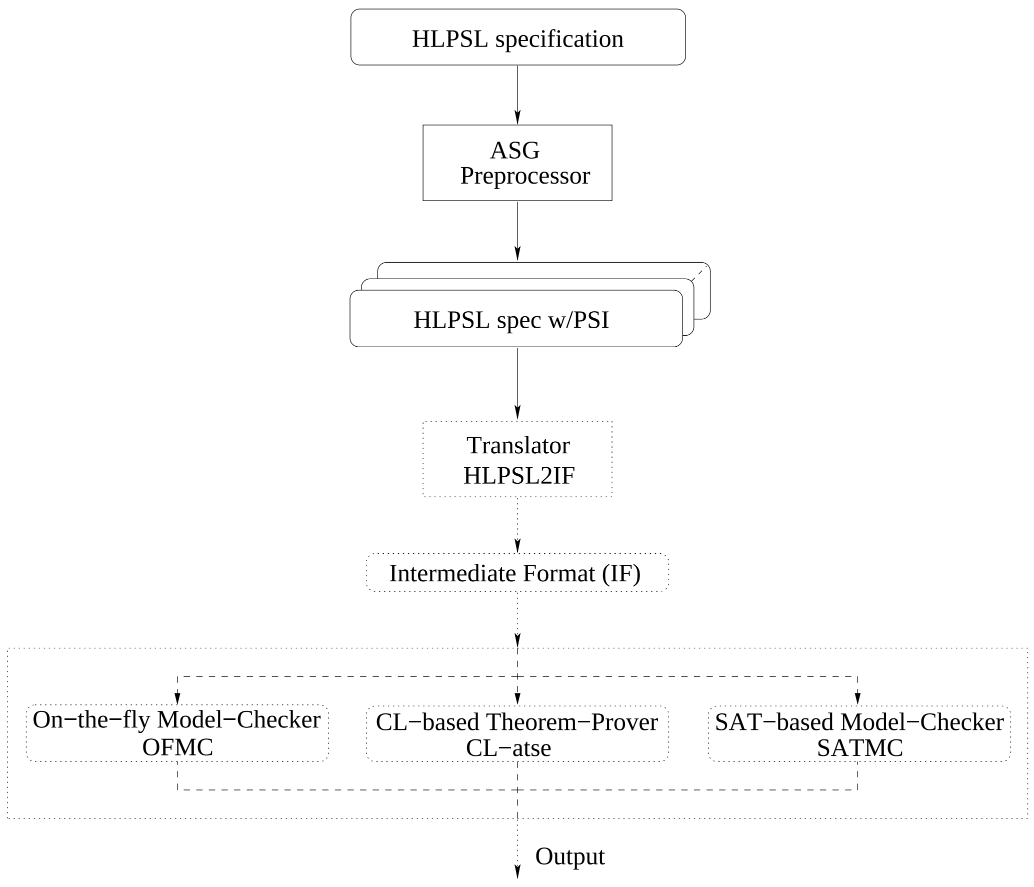

Figure 1: How the ASG preprocessor fits in the architecture of the AVISPA tool (the previous version of the architecture is depicted with dotted lines).

all PSIs with n SSIs. A model-checking approach can then analyse the protocol for each of the PSIs generated this way. However, even for a small number n, the number of PSIs to be considered is enormous.

In order to reduce this number, we have developed a HLPSL preprocessor called ASG (Automatic Session Generation) which takes as input a HLPSL specification and returns a set of new HLPSL specifications in which the top-level role is replaced by a new one; the new top-level role declares a PSI, determined according to two modalities:

1. if the back-end can handle variables in the initial state, i.e. if it can handle multiple initial states, then we use the *symbolic sessions* mode;

2. otherwise, we use the *ground sessions* mode.

See Section 3.2.3 for a detailed explanation of this distinction.

Figure 1 shows how the preprocessor fits into the global AVISPA picture. The HLPSL specification is now fed directly to the ASG preprocessor, which generates a number of new specifications (possibly just one in the symbolic sessions mode) to be fed to the "rest" of the AVISPA tool, here represented with dotted boxes. Note how ASG neatly decouples the problem of PSI generation from the compilation phase: what comes out of ASG is still a (set of) pure HLPSL file(s), which the compiler can compile exactly as before.

3.2.1 Making the notion of “session” in HLPSL precise

So far, there is no well-defined notion of a “session” in HLPSL. Actually, as we have said before, the end-user is given so much expressive power that a protocol session might be “scattered” among many role definitions. In order to overcome this, and also to make things clearer in the HLPSL specifications themselves, we first require that each specification adhere to the following two rules:

1. the top-level role must be called `Environment` (this was just a “soft” requirement in [2]);

2. there must be a role representing a *single protocol session*; this role is usually called `Session` and, once called, generates a SSI, and will therefore be called *SSI-role* in the following.

The scenario specification given above can then be equivalently formulated as follows:

```
role Session (A,B: agent, Ka, Kb: public_key) def=
 composition
  SHARE_Init(A,B,Ka,Kb) /\
  SHARE_Resp(B,A,Kb,Ka)
end role

role Environment() def=
 knowledge(i) = {a,b,i,ka,kb,ki}
 composition
  Session(a,b,ka,kb) /\
  Session(a,i,ka,ki)
end role
```

The first advantage of following this approach is that now the user can manually specify parallel sessions in a much cleaner way: `Session` is the SSI-role and takes care of one single session, whereas `Environment` contains precisely the composition of the required sessions and therefore establishes what PSI we are interested in for this problem. Once the HLPSL specification follows these guidelines, generating a PSI becomes a matter of building the right `Environment` role.

3.2.2 Preprocessing HLPSL

The HLPSL preprocessor, called ASG, takes a HLPSL specification as the one above as input, with three “switches”:

```
ASG [-s n|-g n] -p <role_inst>
```

The options `-s n` and `-g n` call for n symbolic or ground SSIs per PSI, while `-p <role_inst>` is the HLPSL `Role_instantiation` of the SSI-role[4]. `role_inst` gives the end-user the possibility of specifying what formal arguments of the SSI-role role will be permuted over when generating the new `Environment`. The rule of thumb is that whatever is a HLPSL variable (that is, begins with a capital letter like in Prolog) in the instantiation will be permuted over in the new `Environment`.

For example, specifying `-p Session(A,b,k(A),kb)` on the command-line tells ASG that `A` is the only object changing over all possible SSIs, whereas all other formal arguments stay the same. If, for example, there are three agents `a`, `b` and `c`, then `A` will be instantiated with one of them, generating SSIs in which the players are, in turn, `a` and `b`, `b` and `b`, and `c` and `b`.

3.2.3 Ground versus symbolic PSIs

In the *symbolic sessions* mode, a single new HLPSL file is generated, whose top-level role contains n calls to `role_inst`. In each call, logical variables specified on the command-line are different. The result is that a PSI with n sessions, each one with potentially different parameters, are launched. Of course, if we employ this mode, the IF will contain logical variables in the initial state, that is, the back-end must be able to symbolically handle multiple initial states.

In contrast, in the *ground sessions* mode, a number of distinct PSIs are generated, each one in a different HLPSL file. In each file the top-level role contains a PSI made of n SSIs. In this case, it is ASG that generates the SSIs for each PSI, since we are assuming that the back-end is not able to deal with symbolic initial states. This again poses the problem that, even for a small number of SSIs, the number of possible PSIs is huge.

3.2.4 Reducing the number of PSIs

Let n be the number of SSIs in each PSI, as above, and let r be the number of roles involved in a protocol. Given that, in a session, any agent *plus the intruder* can play any role, the set of all possible SSIs for k distinct agents, call it $SSI(k)$, has $(k+1)^r$ elements. With respect to the SHARE example above,

$$SSI(2) = \{\{a,a\},\{a,b\},\{a,i\},\{b,a\},\{b,b\},\{b,i\},\{i,a\},\{i,b\},\{i,i\}\}.$$

Given $SSI(k)$ then, the set of possible PSIs involving n parallel SSIs, call it $PSI(n,k)$, indeed consists of permutations of $SSI(k)$, that is

$$PSI(n,k) \subseteq \{S : |S| = n, S \in 2^{SSI(k)}\}.$$

In order to reduce $PSI(n,k)$ we notice that

[4]Note that the role called in `role_inst` must exist in the HLPSL specification as a role having the same number of formal arguments as specified on the command-line

1. the order in which SSIs are composed does not matter — recall that composition realises parallelism. So if two or more PSIs are multiset-equivalent, only one of them needs be considered; for example, $\{\{a,a\},\{a,i\}\}$ and $\{\{a,i\},\{a,a\}\}$ are equivalent;
2. it is pointless to consider PSIs in which the intruder does not come into play at all, or is the only agent playing;
3. finally, PSIs that are equal modulo renaming of the honest agents (without allowing an agent to be renamed to another agent already present in the PSI) are equivalent, and again, only one of them needs be considered. In fact, PSI $\{\{a,a\},\{i,b\}\}$ is obviously equivalent to $\{\{b,b\},\{i,a\}\}$ (but not to $\{\{a,a\},\{i,a\}\}$, for example).

Given n and k, it is quite easy to generate one by one all PSIs in $PSI(n,k)$ and discard those which obey one of the above criteria. This procedure has been implemented in ASG.

Again, with respect to the SHARE example above, and supposing we are looking for PSIs with 3 SSIs,

$$\begin{array}{rcl} PSI(3,2) & = & \{\ \{\{a,a\},\{a,a\},\{a,i\}\}, \\ & & \quad \{\{a,a\},\{a,a\},\{i,a\}\}, \\ & \vdots & \\ & & \quad \{\{i,i\},\{i,i\},\{i,b\}\}\ \} \end{array}$$

3.3 Examples

Consider the SHARE example again, and suppose that we want to consider PSIs with 3 SSIs. In this case, we call

```
ASG -s 3 -p "Session(A,B,k(A),k(B))"
```

(double-quotes are usually required to avoid UNIX shell expansion). The terms `k(A)` and `k(B)` capture the fact that public keys are functions of the agents (i.e. an agent uses the same public key in each role he plays). The result is an HLPSL specification in which `Environment` has been replaced by:

```
role Environment() def=
 knowledge(i) = {k}
 exists A1, B1, A2, B2, A3, B3: agent
 composition
  Session(A1,B1,k(A1),k(B1)) /\
  Session(A2,B2,k(A2),k(B2)) /\
  Session(A3,B3,k(A3),k(B3))
end role
```

(the preprocessor easily deduces the types of the logical variables, which must be declared as local in the newly-created role).

We specify the role call template on the command-line in order to have more flexibility in the generated SSIs/PSIs. Whatever is ground in the template remains ground, as is the case of the mapping `k` of agents to public keys; whatever contains logical variables undergoes an exhaustive replacement, as is the case of `A` and `B`.

We now consider an example with ground sessions. Suppose we have called

```
ASG -g 3 -p "Session(A,B,k(A),k(B))"
```

In the role instantiation, `A` and `B` denote the principals. The result is a set of HLPSL specifications, the first of which is

```
role Environment() def=
 knowledge(i) = {k,a,i}
 composition
  Session(a,a,k(a),k(a)) /\
  Session(a,a,k(a),k(a)) /\
  Session(a,i,k(a),k(i))
end role
```

and the last is

```
role Environment() def=
 knowledge(i) = {k,a,i}
 composition
  Session(i,i,k(i),k(a)) /\
  Session(i,i,k(i),k(a)) /\
  Session(i,b,k(i),k(a))
end role
```

4 Symbolic Sessions

In this section, we describe a technique that we have devised that exploits the lazy intruder in order to deal with parallel sessions symbolically at the IF level. We have implemented this technique in the OFMC back-end, but it can be similarly applied in tools that are based on a lazy intruder.

In the IF, the initial state determines the scenario to be explored: every state fact represents an honest agent that is willing to perform one run of the protocol. This is because every rule in the IF contains exactly one state fact both on the left-hand-side and on the right-hand side, so that the number of agent facts is the same in all reachable states.

A state fact of the initial state thus corresponds to the initial node of a strand in the strand space model or an agent process in a process calculus model; see, e.g., [7, 10, 11][5].

Given a protocol with r roles in HLPSL, for every specified session, the HLPSL2IF translator generates r state facts in the initial state of the resulting IF files, one for each role that contains the agent names of the respective session instance. For example, we could consider an initial state containing the facts

$$\begin{gathered} state(\mathcal{A}, 0, a, b, session_1).state(\mathcal{B}, 0, b, a, session_1). \\ state(\mathcal{A}, 0, a, b, session_2).state(\mathcal{B}, 0, b, a, session_2). \\ state(\mathcal{A}, 0, b, i, session_3).state(\mathcal{B}, 0, i, b, session_3) \end{gathered}$$

Every pair of *state* facts contains a unique identifier $session_j$. In OFMC's set rewriting approach, this allows to specify two or more parallel sessions between the same agents (in the example, the sessions 1 and 2).

The last state fact, $state(\mathcal{B}, 0, i, b, session_3)$, represents that the intruder can execute the protocol in the role $\mathcal{B}$ with b in the role $\mathcal{A}$. This and other state facts for the intruder are actually not necessary under the Dolev-Yao intruder model since an intruder can always execute a protocol in the intended way under his own, real name.[6]

One can interpret automatic session generation as a kind of parameter search: the protocol model is parametrised over a scenario and we can explore the values of this parameter (for a given upper bound). We now introduce a refinement of this idea that we use in OFMC and that can similarly be used by protocol analysis tools that employ the lazy intruder. Namely, we improve upon this approach by letting the lazy intruder take care of sessions. This is possible as there is no essential difference between choosing an agent from a limited set of possibilities when generating sessions and using the lazy intruder to choose a message from his knowledge during the normal search.

More in detail, we take advantage of the symbolic representation of the lazy intruder to avoid enumerating all possible session instances (for a given bound on the number of sessions). To do this, we instantiate the roles with variables rather than with constant agent names. The variables are then instantiated by unification during search either to constant agent names or to other variables. For instance, for $n = 3$ and a protocol with two roles $\mathcal{A}$ and $\mathcal{B}$, the lazy initial state contains

$$\begin{gathered} state(\mathcal{A}, 0, A_1, B_1, session_1).state(\mathcal{B}, 0, B_2, A_2, session_1). \\ state(\mathcal{A}, 0, A_3, B_3, session_2).state(\mathcal{B}, 0, B_4, A_4, session_2). \\ state(\mathcal{A}, 0, A_5, B_5, session_3).state(\mathcal{B}, 0, B_6, A_6, session_3) \\ A_1 \neq i \land B_2 \neq i \land A_3 \neq i \land B_4 \neq i \land A_5 \neq i \land B_6 \neq i \quad . \end{gathered}$$

[5]We assume here that the rules of an IF protocol specification ensure that every state fact can be involved only in a finite number of transitions. This assumption holds, for instance, for all protocols that can be specified in HLPSL, but it excludes streaming protocols, unless one bounds the length of the stream.

[6]Indeed, the HLPSL2IF translator checks that the HLPSL specification of the protocol is executable in the sense that, given the required initial knowledge, every agent can construct the messages he is supposed to, and this ability is of course subsumed by the abilities of the Dolev-Yao intruder.

Note that, following our previous discussion, we add inequalities that explicitly prevent unifying variables that are intended to represent honest agents with the name of the intruder.

The resulting specification of the initial state is slightly more general than the enumeration of ground sessions described above; there, the state facts of the same session ID contain the same agent names, i.e. it results from unifying $A_1 = A_2$, $B_1 = B_2$, etc:

$$\begin{array}{c} state(\mathcal{A}, 0, A_1, B_1, session_1).state(\mathcal{B}, 0, B_1, A_1, session_1). \\ state(\mathcal{A}, 0, A_3, B_3, session_2).state(\mathcal{B}, 0, B_3, A_3, session_2). \\ state(\mathcal{A}, 0, A_5, B_5, session_3).state(\mathcal{B}, 0, B_5, A_5, session_3) \\ A_1 \neq i \land B_1 \neq i \land A_3 \neq i \land B_3 \neq i \land A_5 \neq i \land B_5 \neq i \qquad . \end{array}$$

Let now $\mathsf{Agent} = \{i, a, b, \ldots\}$ be a set of agent names. If all the variables that we have introduced into the initial state range over the set Agent, then we have declared a symbolic initial state that represents the set of all initial states that would result from automatic session generation with this set of agents (for a given bound n of the sessions).

In order to integrate the approach just described with the lazy intruder there is, however, one subtlety that we must address: we have assumed that all constraint sets are well-formed, in particular, that each variable that appears in a constraint set is introduced on the left-hand side of some constraint. Intuitively speaking, combining session instantiation with the lazy intruder means that the intruder chooses the concrete names of the agents in the initial state, but leaves variables for these choices and lazily instantiates them during the search. The initial state must therefore contain a constraint set that requires that the lazy intruder "generates" all agent names from his initial intruder knowledge IK_0. That is, for symbolic agent names $A_1, \ldots, A_p$ in the initial state we have the constraint

$$\mathit{from}(\{A_1, \ldots, A_p\}, IK_0) \cup I_0 \quad ,$$

where I_0 is the initial set of inequalities and we assume that IK_0 contains the names of all agents (which is usually not a problem, at least for protocols not involving anonymity or pseudonymity). Note, however, that this constraint actually allows more than we want: each A_j can be instantiated with an arbitrary term that can be generated from IK_0 (e.g. the concatenation of two agent names). We can simply exclude this by enforcing the A_j's to be typed variables (that can only be instantiated with constants of type Agent). As previously remarked, it is not difficult to integrate a typed model into OFMC.

Regarding the set Agent as a type even allows us to use an infinite set of agent names, with the special rule that the intruder knows every constant of type Agent. Although for a finite number of sessions we only need a finite set of agents (everything else will be equivalent modulo renaming of constant agent names), it can be difficult to determine how many distinct agents are actually necessary (in particular, if there is some form of negation in the model). An infinite type Agent makes matters simpler in this regard. Also, the fact that the intruder knows all agent names implies that the initial constraint set is always satisfiable, and its ground solutions are exactly the possible instantiations of the sessions. In particular, we then have the following property:

Property 1. *Consider a symbolic initial state with variables $A_1, \ldots, A_p$ of type* Agent*, and the initial intruder knowledge IK_0 such that $a \in IK_0$ for $a \in \mathsf{Agent}$. Then the semantics of the initial constraint set is the set of all substitutions of the variables A_j with agent names, i.e.*

$$[\![from(\{A_1, \ldots, A_p\}, IK_0)]\!] = \{\, \sigma \mid domain(\sigma) = \{A_1, \ldots, A_p\} \ \wedge \ A_j\sigma \in \mathsf{Agent} \} \, .$$

Hence, the lazy intruder can be straightforwardly adapted to solve the problem of session instantiation.

As a concrete example, we show the states that form the trace for the man-in-the-middle attack on the Needham-Schroeder Public Key Protocol NSPK [4, 9] in the symbolic model where there is one session. The initial state in this case contains

$$\begin{array}{c} state(\mathcal{A}, 0, A_1, B_1, session_1).state(\mathcal{B}, 0, B_2, A_2, session_1) \\ A_1 \neq i \wedge B_2 \neq i \end{array} \quad .$$

The attack trace starts with agent A_1 sending a message for B_1, encrypted with B_1's public key. The intruder can analyse this message iff $B_1 = i$; hence, we have a case-split, i.e. one state where we perform the substitution $B_1 = i$ and one where we have the additional inequality $B_1 \neq i$. The attack corresponds to the first case, where the substitution is performed. In this case, the intruder finds out the nonce n_1 that A_1 has created for him. In the next step, the intruder sends a message to B_2, posing as A_2 (whoever that is). B_2 responds with a message encrypted with the public-key of A_2, and again the intruder can decrypt that message iff $A_2 = i$. Now consider the case $A_2 \neq i$. In this case, the intruder chooses to send (under his real name) an answer to A_1's first message. A_1 expects this message to be encrypted with his public key and to contain the nonce he sent earlier to i, i.e. we have the constraint set

$$\begin{array}{ll} \{ & from(\{A_1, A_2, B_2\}, IK_0)\, , \\ & from(N_A, IK_0 \cup \ n_1)\, , \\ & from(\{n_1, N_B\}_{k(A_1)}, IK_0 \cup \ n_1 \cup \ \{N_A, n_2\}_{k(A_2)}) \ \} \end{array}$$

and the inequalities

$$A_1 \neq i \wedge B_2 \neq i \, .$$

Here N_A is a variable representing the nonce the intruder sent earlier to B_2, n_1 is the nonce the agent A_1 has created for i, n_2 is the nonce the agent B_2 has created for A_2, and N_B is a variable for whatever the intruder sends to A_1 as his nonce.

The lazy intruder constraint reduction will identify two possible solutions for this constraint set: either the intruder generated the last message $\{n_1, N_B\}_{k(A_1)}$ from its components (which he can do, as he knows A_1, the key-table k, and the nonce n_1) or he replays the message he just received from A_2, since it is unifiable using the substitution

$$[\, A_2 \mapsto A_1 \, , \ N_A \mapsto n_1 \, , \ N_B \mapsto n_2 \,] \, .$$

Hence, we have found out that this trace (which leads to a man-in-the-middle attack) only works if the agent that B_2 thinks he is talking to (i.e. A_2) is indeed the one who receives the message (i.e. A_1), while A_1 thinks he is talking to i. As an answer, A_1 sends the final message of the protocol $\{n_2\}_{k(i)}$, so the intruder knows the nonce B_2 has generated for $A_2 \neq i$, which is a violation of secrecy. Furthermore, authentication can be violated if he sends this nonce to B_2.

This example demonstrates how, during the search, the space of possible instances is narrowed down in a demand-driven fashion. Note that in the manual session generation, even for only three agents $\mathsf{Agent} = \{a, b, i\}$, we would have 24 instances. However, under the demand-driven strategy of the lazy intruder, not all of these instances must be explored.

To conclude this section, observe that during the example derivation, variables were instantiated with other variables or with the constant i. In general, the A_j's can be instantiated only with those constant agent names that appear in the initial state or in some transition rule, and this is usually just the name of the intruder i. This reflects the result of [5] that "two agents are sufficient": in most cases, one can instantiate all the agent variables in an attack trace with one honest agent a. In other words, it is often sufficient to distinguish simply between honest and dishonest agents.[7]

[7]One actually needs more than two agents if the protocol allows arbitrary inequalities on agent names, but one can still bound the number of agents necessary.

References

[1] M. Abadi. Two facets of authentication. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW'98)*, pages 27–33. IEEE Computer Society Press, 1998.

[2] AVISPA. Deliverable 2.1: The High-Level Protocol Specification Language. Available at `http://www.avispa-project.org`, 2003.

[3] AVISPA. Deliverable 2.3: The Intermediate Format. Available at `http://www.avispa-project.org`, 2003.

[4] J. Clark and J. Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 17. Nov. 1997. URL: `www.cs.york.ac.uk/~jac/papers/drareview.ps.gz`.

[5] H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. In *Proceedings of ESOP'2003*, LNCS 2618, pages 99–113. Springer-Verlag, 2003.

[6] G. Denker, J. Millen, and H. Rueß. The CAPSL Integrated Protocol Environment. Technical Report SRI-CSL-2000-02, SRI International, Menlo Park, CA, October 2000. See `www.csl.sri.com/~millen/capsl/`.

[7] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

[8] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[9] G. Lowe. Breaking and Fixing the Needham-Shroeder Public-Key Protocol Using FDR. In T. Margaria and B. Steffen, editors, *Proceedings of TACAS'96*, LNCS 1055, pages 147–166. Springer-Verlag, 1996.

[10] G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998. See `http://web.comlab.ox.ac.uk/oucl/work/gavin.lowe/Security/Casper/`.

[11] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols.* Addison Wesley, 2000.