Using Animation to Improve Formal Specifications of Security Protocols

Yohan Boichut[1], Thomas Genet[1], Yann Glouche[1] and Olivier Heen[2]

[1]*IRISA, Rennes, France,* yohan.boichut@irisa.fr, thomas.genet@irisa.fr, yann.glouche@irisa.fr

[2]*THOMSON R&D France, Security Lab,* olivier.heen@thomson.net

The verification of cryptographic protocols has greatly improved these last years. Automated tools such as AVISPA provide real help in finding and characterizing attacks. The counterpart is the formal specification of the protocol, using an appropriate language such as HLPSL. Since HLPSL is a very expressive language, this stage is complicated and error-prone before a correct specification is eventually obtained. The verification tools of AVISPA are not designed to detect such specification errors. Unfortunately, as long as it contains typo-like errors, the verification of a HLPSL specification is pointless. In this paper, we propose an animation tool called SPAN [†]. It turns a formal protocol specification into an execution diagram, according to user choices. We show how the visualization eases the formal specification stage in many ways: drawing of typical execution diagrams, visualization of protocol termination, understanding of interleaved sessions, detection of unwanted side effects, etc. We also show how visualization and simulation of an intruder helps in finding attacks that are not automatically detected by tools.

Mots-clés: Cryptographic protocols, Verification, Animation, AVISPA, HLPSL, SPAN

1 Introduction

As they expand, digital transmissions require more and more security solutions. Sometimes, a straightforward adaptation of widely known protocols (SSL, IPSEC, PGP, ...) is adequate. Sometimes, a new protocol must be designed. In this case, there are many error factors.

- The protocol is an answer to a recent security problem: all the aspects of the problem may not be known already.
- The protocol is an answer to a critical situation: the design time may be short.
- The protocol is designed for commercial use: some details may not be published too early.

In such situations, there is an urge need for proofs, even before any public review can take place. Verification tools such as proposed by AVISPA [ABB+05] are a good answer: they let the designers carry out the automated formal verification on their own. Of course the designers have to learn the associated concepts and languages, i.e. the HLPSL [CCC+04] formal specification language in the particular case of AVISPA.

Is it possible to ease the learning task and the day-to-day use of such tools? In this paper we describe SPAN a companion tool for AVISPA. It is dedicated to the animation of HLPSL specifications of cryptographic protocols. Given such specifications it draws visual transition diagrams, according to user choices. In particular it achieves constraint solving on message patterns to determine which transitions (message exchanges in our setting) are likely to be fired. As a result, this kind of animation requires specific software and cannot be solved using a generic library for animating MSC like [WBL06]. Doing so, SPAN facilitates designers work in several ways: at first during the specification stage in order to insure that the written specification leads to an acceptable transition diagram; then during the correction checking, for quickly building,

[†] SPAN is freely available at http://www.irisa.fr/lande/genet/span/

recording and replaying execution cases. Also when AVISPA finds attacks, SPAN makes it easy to replay it and to find variants. At last, it allows interactive search for specific attacks which are not automatically detected.

The remainder of the paper is organized as follows. We first recall the necessary basis about cryptographic protocols and their flaws in Section 2, then we sum up the basics of AVISPA and HLPSL in Section 3. In Section 4, we describe SPAN and exemplify its many ways of helping designers. Finally, Section 5 provides some further experiments. Note that, all along the paper, we use a straightforward variant of Diffie-Hellman protocol as a running example.

2 Cryptographic protocols and their verification

The Diffie-Hellman protocol is a key establishment protocol between two agents A and B. A simple variant of this protocol is composed of three steps and is presented below using "Alice & Bob" notation. The established key is denoted by K and, in the final step, a secret sent by A to B is encoded using K.

1 - $A \hookrightarrow B : G^{N_a}$
2 - $B \hookrightarrow A : G^{N_b}$, A and B compute key $K = (G^{N_a})^{N_b} = (G^{N_b})^{N_a}$
3 - $A \hookrightarrow B : \{Nsecret\}_K$

At step 1, A generates the *nonce* (a random number) N_a and computes G^{N_a} where G is a number known by every agent. Thus A sends the message G^{N_a} to the agent B. At step 2, the agent B also generates a number N_b and computes G^{N_b} and $K = (G^{N_a})^{N_b}$. The former is sent to A and the latter stands for the symmetric key shared between A and B. As soon as A receives the message G^{N_b} from B, it then computes $(G^{N_b})^{N_a}$ and thus considers it as the symmetric key shared with B. Indeed, according to the algebraic properties of the exponentiation, $K = (G^{N_a})^{N_b} = (G^{N_b})^{N_a}$. Finally, the message $\{Nsecret\}_K$ is sent by A to B in which $Nsecret$ is a datum standing for a secret between A and B, and $\{\ \}_K$ denotes the use of a symmetric encryption algorithm with a key K.

This protocol is well-known to suffer from a *man in the middle* attack. This attack is detailed below. In the Dolev and Yao model [DY83], the intruder can read every message over the network. A classical hypothesis is to consider the *intruder as the network*. Thus, every message is sent, first of all, to the intruder before being transmitted to the expected agent. The notation $I(A)$ means that the intruder pretends to be A.

1 - $A \hookrightarrow I : G^{N_a}$
2 - $I(A) \hookrightarrow B : G^{N_i}$
3 - $B \hookrightarrow I : G^{N_b}$, B and I compute the key $K_{IB} = (G^{N_i})^{N_b} = (G^{N_b})^{N_i}$
4 - $I(B) \hookrightarrow A : G^{N_i}$, A and I compute the key $K_{IA} = (G^{N_i})^{N_a} = (G^{N_a})^{N_i}$
5 - $A \hookrightarrow I : \{Nsecret\}_{K_{IA}}$
6 - $I(A) \hookrightarrow B : \{Nsecret\}_{K_{IB}}$

Roughly, the intruder establishes two keys: $K_{IA} = (G^{N_a})^{N_i}$ with A at Steps 1 and 4, and $K_{IB} = (G^{N_b})^{N_i}$ with B at Steps 2 and 3. At Step 5, the agent A sends the secret data to B using the key K_{IA} shared with the intruder. The intruder then extracts the secret data and forward it to B with the other key. Finally, the intruder knows a secret shared between both A and B.

In general, safety of cryptographic protocols is not decidable. However, bounding the number of sessions allows automatic attack detection. On the other hand, using over-approximations gives a criterion for safety properties on protocols with an unbounded number of sessions. The tools of the AVISPA system use these two techniques for protocol verification.

3 The AVISPA system

In the AVISPA tool [ABB+05], cryptographic protocols are specified using the High Level Protocol Specification Language (HLPSL) [CCC+04]. There exist four tools devoted to the automatic verification of security properties over these specifications: OFMC [BMV05], CL-Atse [Tur06a], SATMC [AC05] and TA4SP [BHK05].

3.1 The specification language HLPSL

The language HLPSL, developed in the framework of the European Union project AVISPA[‡], is modular and allows the specification of control-flow patterns, data structures and various intruder models. Its semantics are based on Lamport's TLA (Temporal Logic of Actions [Lam94]). In this part, we give a flavor of HLPSL using the specification of the Diffie Hellman protocol described in Section 2.

HLPSL specifications are based on role descriptions, i.e. finite state automata, where transitions are fired when a message is sent or received. Contrary to "Alice & Bob" notation, HLPSL imposes explicit definition of roles, nonce generation, message sending and reception, etc. Here is the role declaration for A in the HLPSL specification of Diffie-Hellman, where `=|>` stands for the transition relation and `/\` stands for usual conjunction symbol.

```
role alice(A,B:agent, G:text, Snd,Rcv:channel(dy)) played_by A def=
        local State:nat, Na,Nsecret:text, X,K:message
        init State:=1
transition
1.      State=1 /\ Rcv(start) =|>
        State':=2 /\ Na':=new() /\ Snd(exp(G,Na'))
2.      State=2 /\ Rcv(X') =|>
        State':=3 /\ K':=exp(X',Na) /\ Nsecret':= new() /\ Snd({Nsecret'}_K')
end role
```

As mentioned previously, the HLPSL is based on a notation à la TLA where the meaning of a primed variable X' depends on the location of this variable. Indeed, if X' occurs in a message pattern of the left hand side of a transition then a new value is obtained for X by matching the message pattern on received messages. Then this value is accessible by X' in the same transition (see the variable `X` in transition `2.` of the role `alice`). If X' occurs only in the right hand side of a transition then this variable specifies a nonce. A random value is assigned to the variable X using the instruction `new()` and this value is accessible by X' in the current transition (see transition 1 of the role `alice` concerning the variable `Na` for instance). Find below the role for B:

```
role bob(B,A:agent, G:text, Snd,Rcv:channel(dy)) played_by B def=
        local State:nat, Y,K:message, Nb,Nsecret:text
        init State:=1
transition
1.      State=1 /\ Rcv(Y') =|>
        State':=2 /\ Nb':=new() /\ K':=exp(Y',Nb') /\ Snd(exp(G,Nb'))
2.      State=2 /\ Rcv({Nsecret'}_K) =|>
        State':=3
end role
```

Roles can be composed together in sessions where the knowledge shared between the roles, the datum `G` for instance, are made explicit:

```
role session (A,B:agent, G:text) def=
        local SND_A,RCV_A,SND_B,RCV_B:channel(dy) def=
composition
        alice(A,B,G,SND_A,RCV_A) /\ bob(B,A,G,SND_B,RCV_B)
end role
```

The environment used for protocol execution and verification is defined, where `i` denotes the intruder. The environment covers the initial knowledge of the intruder and the initial setting for the sessions, i.e. how many sessions are run and who run them.

```
role environment() def=
        const a,b:agent, g:text
composition
        session(a,b,g,Snd,Rcv)
end role
```

[‡] http://www.avispa-project.org/

3.2 Verification tools in AVISPA

AVISPA is a push-button tool for the automated validation of Internet security-sensitive protocols and applications. Its input is an HLPSL specification of a protocol and the verification is performed by different back-ends that implement a variety of state-of-the-art automatic analysis techniques.

The current version of the AVISPA tool integrates four back-ends: OFMC, CL-ATSE, SATMC and TA4SP. The On-the-fly Model-Checker (OFMC) [BMV05] performs protocol falsification and bounded verification by exploring the transition system. The Constraint-Logic-based Attack Searcher (CL-AtSe) applies constraint solving as in [Tur06a], with some powerful simplification heuristics and redundancy elimination techniques. The SAT-based Model-Checker (SATMC) [AC05] builds a propositional formula encoding a bounded unrolling of the transition relation, the initial state and the set of states representing a violation of the security properties. The propositional formula is then fed to a state-of-the-art SAT solver and any model found is translated back into an attack. The TA4SP (Tree Automata based on Automatic Approximations for the Analysis of Security Protocols) back-end [BHK05] approximates the intruder knowledge by using regular tree languages and rewriting. For secrecy properties, TA4SP can show whether a protocol is flawed (by under-approximation) or whether it is safe for any number of sessions (by over-approximation).

Among the four tools mentioned above, only OFMC and CL-ATSE can perform analysis on the Diffie-Hellman protocol because it uses the exponentiation operator. Against this protocol, both tools return the following attack:

```
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND
  TYPED_MODEL
PROTOCOL
  Diffie-Hellman.if
GOAL
  Secrecy attack on (n2(Nsecret))
BACKEND
  CL-AtSe
STATISTICS
  Analysed   : 0 states
  Reachable  : 0 states
  Translation: 0.00 seconds
  Computation: 0.00 seconds
ATTACK TRACE
 i -> (a,3): start
 (a,3) -> i: exp(g,n1(Na))
 i -> (a,3): g
 (a,3) -> i: {n2(Nsecret)}_(exp(g,n1(Na)))
             & Secret(n2(Nsecret),set_53);
             Add a to set_53; Add b to set_53;
```

```
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND

PROTOCOL
 Diffie-Hellman.if
GOAL
  secrecy_of_secretna
BACKEND
  OFMC
STATISTICS
  parseTime: 0.00s
  searchTime: 0.01s
  visitedNodes: 1 nodes
  depth: 1 plies
ATTACK TRACE
i -> (a,3): start
(a,3) -> i: exp(g,Na(1))
i -> (a,3): g
(a,3) -> i: {Nsecret(2)}_(exp(g,Na(1)))
```

Roughly, the attack raised by both verification tools is not the man in the middle attack described in Section 2. In fact, this attack is even not of the *man in the middle* form since it only brings into play two agents. In the constructed attack, let Alice be `(a,3)` and the intruder be `i`. Alice starts the protocol by computing her half key `exp(g,n1(Na))` in CL-ATSE (resp. `exp(g,Na(1))` in OFMC) where `n1(Na)` (resp. `Na(1)`) are freshly generated constants standing for the new value of nonce `Na`. Alice sends this message on the network, i.e. to the intruder since the intruder *is* the network. Alice expects Bob to receive this message from the network. However, the intruder pretends to be Bob by sending the half key, being `g`, to Alice. Alice waits for a message of the form `X` where `X` is supposed to be the half key sent by Bob. However, since she has no way of checking any structural property about `X` she accepts `g` sent by the intruder in `X`. Then, Alice computes the key `exp(X,n1(Na))` which is nothing else than `exp(g,n1(Na))`. Finally, she composes the message containing the secret to share with Bob, encode it with `exp(g,n1(Na))`, and sends it on the network (the intruder). The intruder has no difficulty to extract the secret since it already knows `exp(g,n1(Na))`. In order to build the man in the middle attack given in Section 2, the intruder can proceed in a similar way with Bob than it does with Alice. However, this particular attack has to be constructed by hand and cannot be obtained from the verification tools.

Because of the quality of HLPSL and of the verification tools, the project has obtained very strong results and the tools have a large community of users. However, since HLPSL specifications are far more precise

than usual "Alice & Bob" notation, they are also much more difficult to conceive and to read. In particular, HLPSL specifications are defined role by role rather than message by message. As a result, it is sometimes difficult for the protocol designers to figure out if the HLPSL specification they write really corresponds to the "Alice & Bob" protocol they have in mind. This is one of the reason why AVISPA is still not broadly used in industry. The SPAN tool was designed in order to bridge this gap and ease the formal specification of industrial cryptographic protocols in HLPSL.

4 SPAN: an Animation tool for AVISPA

We have designed a protocol animator, named SPAN [GG06], whose role is to help to design such formal specifications. SPAN can animate HLPSL specifications, i.e. interactively produce Message Sequence Charts [HT03] (MSC for short) which can be seen as an "Alice & Bob" trace of an HLPSL specification. SPAN is written in Ocaml and Tcl/Tk, is distributed under LGPL license and is freely available at `http://www.irisa.fr/lande/genet/span/`. This tool uses some AVISPA libraries: one for HLPSL translation, designed by Laurent Vigneron [CCC+04], and one for matching modulo xor and exponential theories, designed by Mathieu Turuani [Tur06b]. As far as we know, SPAN [GG06] is the first tool designed to help writing, animating, and thus understanding, HLPSL specifications. In [GGHC06], we have shown SPAN helped for the HLPSL specification of the User Supervised Device Pairing protocol (USDP for short) designed by THOMSON [CHKD06].

In Section 4.1, we show some of the basic features of SPAN. In Section 4.2, we show how SPAN can be used to debug HLPSL formal specifications of protocols. In Section 4.3, once the specification corresponds to the protocol to model, we show how SPAN can be used to experiment with the specification and try different environment assumptions. Finally, in Section 4.4, we show how SPAN intruder mode can be used to interactively find and build attacks over protocols defined in HLPSL. In particular, we will see that this tool is of great interest to construct and replay a particular attack when the verification tools of AVISPA stick to a another one.

4.1 SPAN basics

Starting from an HLPSL specification, SPAN helps in building one possible execution (a MSC drawing) according to interactive user choices.

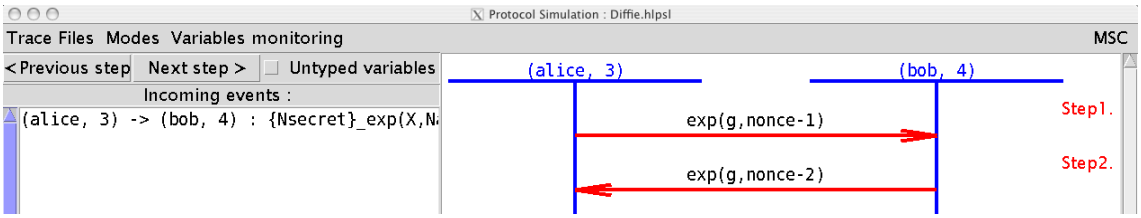


Figure 1: Basic animation of Diffie-Hellman HLPSL specification

For instance, on the HLPSL specification of Diffie-Hellman protocol of Section 3.1, we can run SPAN and interactively construct the following MSC (see Figure 1). In the right hand-side frame, SPAN displays the messages already sent. Note that, SPAN displays terms that correspond to instantiated executions. For instance, the generic role `alice` from the HLPSL specification of Section 3.1 is instantiated by agent `(alice,3)`. Similarly, message patterns are instantiated by constant values: `exp(G, Na)` is instantiated by `exp(g, nonce-1)` where `nonce-1` is a fresh constant value standing for the nonce computed by agent `(alice, 3)` for this step of the protocol. In the left-hand side frame, SPAN displays the possible transitions (here message sending) to trigger by a double-click. In this particular case, there is only one transition to trigger which corresponds to the third step of the protocol. There may be also zero transitions to trigger if the protocol is finished or if there was a mistake in the specification so that the protocol cannot be executed to its end.

During normal simulation, only *useful transitions* are presented to the user. A useful transition corresponds to a message that can be sent *and* received. As a result, the user is not overflowed with too many choices between the transitions to trigger. On the opposite, during intrusion simulation, even non useful transitions are presented to the user since the intruder is able to receive any message. The user may purposely choose a transition, not useful w.r.t. protocol evolution, just to improve the intruder knowledge. For instance, on the following simple HLPSL specification:

```
role alice(A:agent, Snd,Rcv:channel(dy)) played_by A def=
        local Hello:message
transition
1.      Rcv(start) =|>
        Snd(Hello)
end role

role environment() def=
        local Snd,Rcv:channel(dy)
        const a:agent
composition
        alice(a,Snd,Rcv) /\ alice(a,Snd,Rcv)
end role
environment()
```

Figure 2 and Figure 3 show respectively the execution of this protocol under normal simulation (there is no useful transition since there is no reception in any role) and under intruder simulation (the intruder can receive any message).

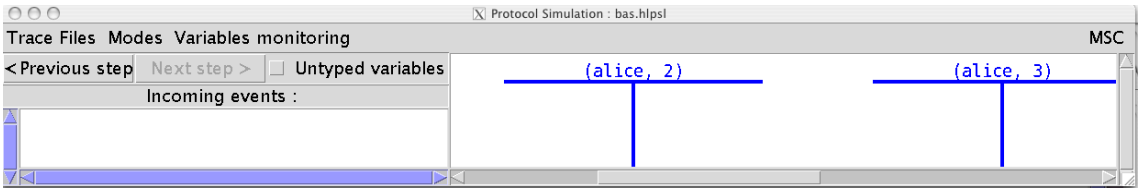


Figure 2: Basic animation without intruder, only useful transitions are displayed

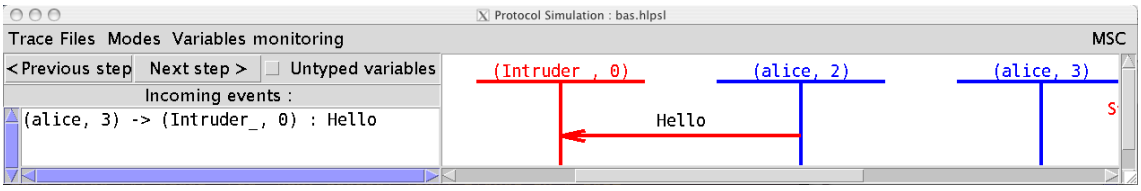


Figure 3: Basic animation with the intruder, the intruder may take advantage of any message

4.2 Debugging HLPSL specifications using SPAN

Although much effort was made on the expressiveness, precision and formal semantics of the HLPSL language, writing a specification in this language is still hard. This is mainly due to the fact that the only way to experiment with a specification is to apply one of the verification tools (OFMC, CL-AtSe, SATMC or TA4SP) whose role is to find attacks on a specific verification model. However, before searching for attacks, the user wants to figure out if the formal HLPSL specification corresponds to what is expected of the protocol. In particular, the user wonders if the HLPSL specification can be run from the beginning to the end, at least for most typical execution cases.

Animating an HLPSL specification makes it possible to find errors in a wrong specification of a protocol. Many specification errors fall out of the scope of the verification tools. This is due to the fact that these tools

do not focus on the *execution* of the specification but rather on their *robustness* against a specific intruder model, i.e. Dolev-Yao model with some arithmetic properties for *exp* and *xor* symbols. With regards to the specification text itself, the verification essentially consists of type checking. We give here some examples of common errors made in the specification. All these errors cannot be found by the verification tools but they can be detected using the SPAN animation. These bugs are very dangerous since they may result into trivially *safe* protocols w.r.t. the verification tools. This is due to the fact that, since the protocol can only be run partially, the intruder does not have enough knowledge to build an attack, and the protocol may be trivially safe. The errors are categorized by the section they belong to in the HLPSL specification.

Environment

The environment definition may be incomplete. For instance, a session may be missing in the environment definition. This is a simple bug that may occur especially when the number of sessions concerned by the basic protocol execution is huge. This can easily be detected in SPAN because of a missing role in the chart.

Sessions

The session definition may be incorrect or incomplete w.r.t. the protocol to model. The completeness problem of sessions is similar to the previous point and can similarly be detected. However, a more tricky bug is, for instance, an error made while declaring the shared (or known) keys in the session declaration. Here is a part of a possible session declaration for the Needham-Schroeder public key protocol [NS78] where roles `alice` and `bob` are supposed to know `Ka` and `Kb` the public keys of each principals. A common error is to make the following typo in the HLPSL specification and write `ii)` instead of `i)`:

```
i)  alice(A,B,Ka,Kb,SA,RA) /\ bob(A,B,Ka,Kb,SB,RB)
ii) alice(A,B,Ka,Kb,SA,RA) /\ bob(A,B,Kb,Kb,SB,RB)
```

In `ii)`, `Ka` has been accidentally replaced by a `Kb`. Such bugs, are easily shown by animating the protocol with SPAN: while animating the protocol, some transitions cannot be triggered. On this example, since role `bob` does no longer know `Ka` it is no longer able to send any message ciphered with this key, and thus the protocol animation is stopped before sending such messages. Note that, using SPAN's variable monitoring, it is possible to display the value of the variables and figure out that the value of variable `Ka`, which is set to constant `kb`, is not correct. On Figure 4, one can see that after the first step of the protocol there are no transition to trigger although this protocol has three steps and that the value of variable `Ka` for role `bob` is not correct.

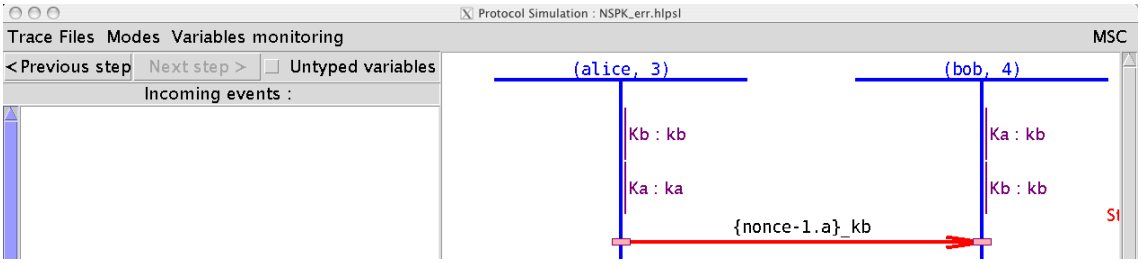


Figure 4: Finding and fixing a bug in an HLPSL specification using variable monitoring

This bug here result into a specification of the Needham-Schroeder protocol which becomes trivially safe. Indeed, since the first message is the only one that can be exchanged in this erroneous version, the usual man in the middle attack on this protocol cannot be built by the intruder.

Roles

The transitions or the message structure may be incorrectly defined. Defining HLPSL transitions is an error-prone task and SPAN can help in detecting many bugs. For instance, assume that we wrote `Snd(exp(Na',G))` instead of `Snd(exp(G,Na'))` in the first transition of role `alice` in specification of Section 3.1. This

error is not detected by verification tools but can be shown while animating the protocol: after the second transition of the protocol, there is no transition to trigger although the protocol is not finished. Furthermore, it is possible to see on the messages labeling the MSC that they do not have a similar structure: `exp(nonce-1,g)` for the first and `exp(g,nonce-2)` for the second one (see Figure 5). The fact that no third transition can be fired means that the third message may be sent but nobody can receive it. This is due to the fact that A and B disagree on the key K. Thus, the message likely to be sent by A is of the form `{Nsecret}_exp(exp(G, Nb), Na)` whereas B waits for a message of the form `{Nsecret}_exp(exp(Na,G),Nb)`, and of course $(G^{N_b})^{N_a} \neq (N_a^G)^{N_b}$.

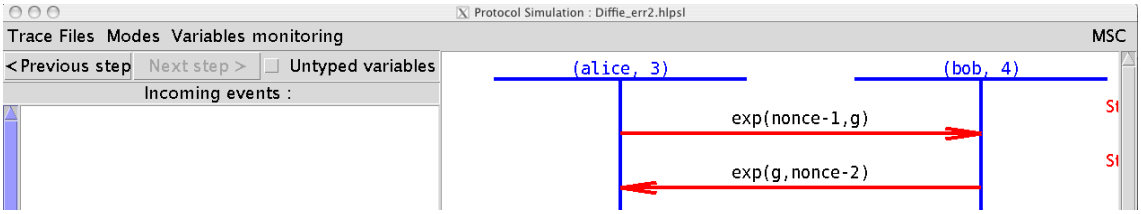


Figure 5: Detecting the bug on message structure (`exp`)

The last example we propose deals with typos made on primed variables which is a problem really hard to detect in HLPSL specification without animation. For instance, assume that we made the following common mistake when writing the HLPSL specification of the role A: we forget the prime for the variable `X` in the second transition:

```
2.      State=2 /\ Rcv(X) =|>
        State':=3 /\ K':= exp(X,Na) /\ Nsecret':=new() /\ Snd({Nsecret'}_K')
```

Without the prime notation, the semantics of this transition is different: the agent A waits for a message that should correspond exactly to `X` where `X` is supposed to have been defined before. In this erroneous specification, since `X` has not been initialized then A cannot receive such a message. However, this specification is *correct* w.r.t. HLPSL semantics and thus the verification tools are not able to detect this problem. Similarly, this may result in a protocol specification which is declared as *safe* although it does not model the protocol we want to specify. Running SPAN on this example brings to light the problem: after the first step of the protocol no transitions can be triggered like it is shown on Figure 6.

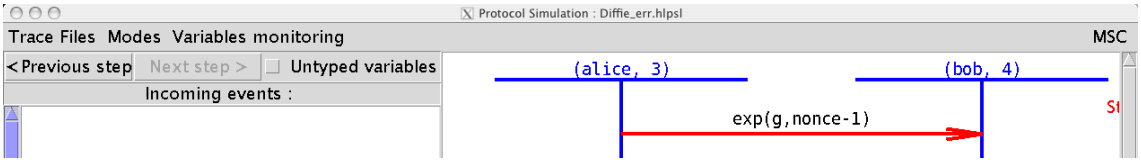


Figure 6: Detecting the bug on primed variables

4.3 Explaining, tuning and experimenting with HLPSL specifications

While using SPAN to develop a formal specification for an industrial protocol [GGHC06], it appears that animation of HLPSL specifications can also be of great interest during the design of the protocol itself. Instead of short-lived, laborious and intricate drawings on a black board, designing protocol specification using HLPSL and SPAN reveals to be very efficient. Using HLPSL and animation at early stages of development is a convenient way of explaining and justifying the choices made during the protocol development.

Furthermore, it also permits to rapidly consider and play with different environment assumptions or different execution of the protocol. This is the case for instance, with the animation of non-deterministic protocols or the animation of deterministic protocols for multiple sessions. The Diffie-Hellman protocol of

Section 3.1 is deterministic and there is only one session between agents A and B. Now, assume that we start from a different environment where we run the same protocol between A and B and between C and D, then the protocol execution becomes non deterministic. This new running environment can be specified by replacing the environment section of Section 3.1 by the following one:

```
role environment() def=
        const a,b,c,d,i:agent, g,ni:text
composition
        session(a,b,g,Snd,Rcv) /\ session(c,d,g,Snd,Rcv)
end role
```

Running SPAN on this new initial environment leads to a different animation. On Figure 7 we can see that there are two instances, namely `(alice,3)` and `(alice,6)` of role `alice` and two instances, namely `(bob,4)` and `(bob,7)` of role `bob`. Note that, real identities, `(alice,3)` is the agent `a` of the HLPSL specification. Similarly, the real identities of the agents `(bob,4)=b`, `(alice,6)=c` and `(bob,7)=d` can be obtained using variable monitoring. In the left-hand side window, one can notice that transitions to be triggered are more numerous than expected.

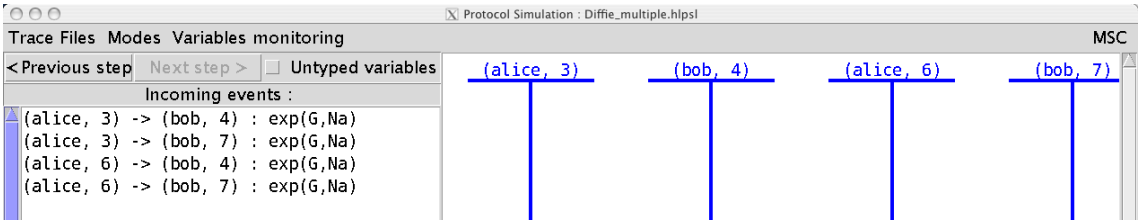


Figure 7: Animation on several sessions of Diffie-Hellman HLPSL specification

Indeed, there is no guarantee that the initial protocol message sent by `(alice,3)=a` is received by `(bob,4)=b`. In particular, it can be received by `(bob,7)=d` and thus we get such a additional transition to trigger in SPAN. Hence, by animating this specification, we can obtain all the interleavings between messages sent by `a`, `b`, `c` and `d`. Since the execution is non deterministic, the user has to choose between all the transitions to trigger so as to construct a possible MSC. In Figure 8 and Figure 9, we show two different interleaved executions of the Diffie-Hellman protocol. Figure 8 shows that it is possible to establish sessions between `(alice,3)=a` and `(bob,4)=b` and between `(alice,6)=c` and `(bob,7)=d`. Figure 9 shows that it is also possible to establish sessions between `(alice,3)=a` and `(bob,7)=d` and between `(alice,6)=c` and `(bob,4)=b`.

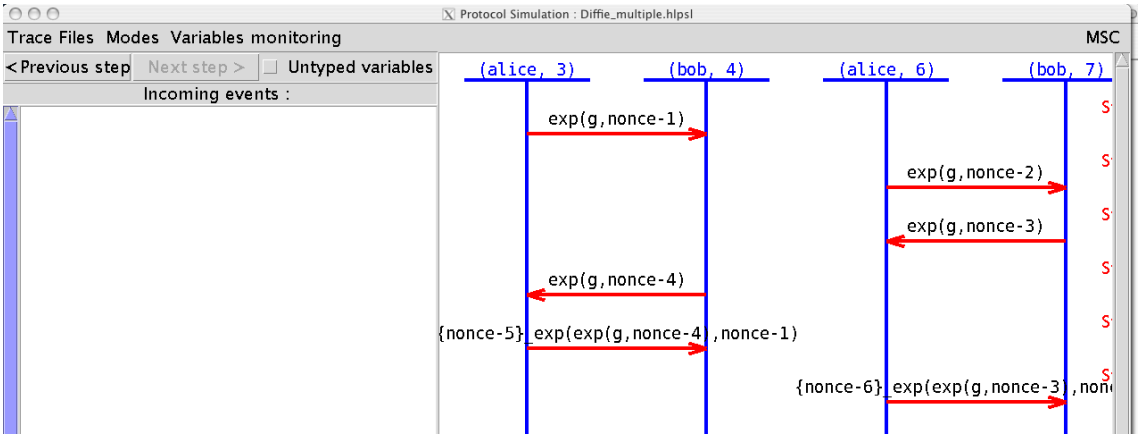


Figure 8: Interleaving of sessions $a-b$ and $c-d$

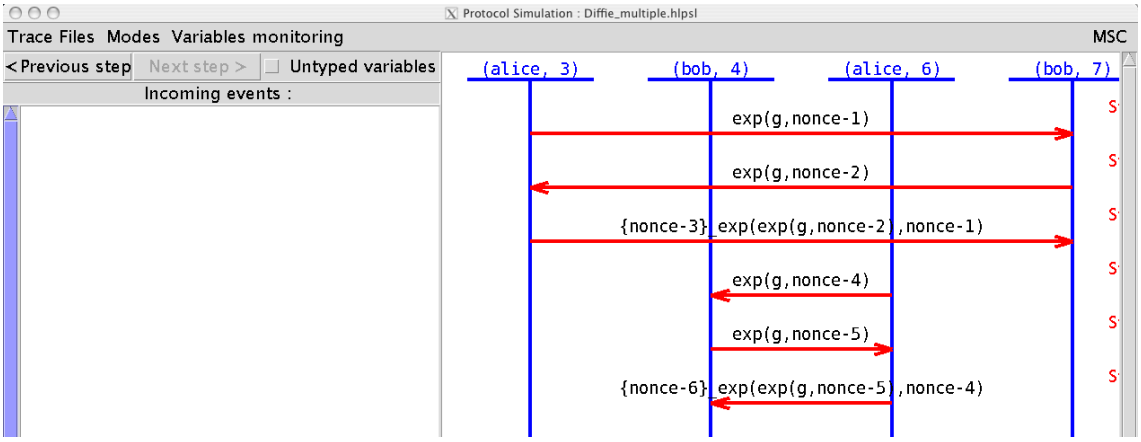


Figure 9: Interleaving of sessions $a-d$ and $c-b$

Note that at every moment SPAN lets the user undo the choices he made or replay them. For a given protocol, it is also possible to save a particular scenario, as a so-called trace file, in order to replay it later or to send it by e-mail.

4.4 Using SPAN to construct attacks over protocols

Once the HLPSL specification is debugged it can be checked automatically for attack detection using the AVISPA verification tools. However, the output trace may not be the expected one as shown in Section 3.2. Animating an HLPSL specification on SPAN allows the user to construct the attack s/he has in mind or to find new ones. SPAN offers specific mechanisms making the attack construction easier. After each step, in the intruder part, SPAN shows the current intruder knowledge and proposes to construct and send malicious messages from this knowledge. Message patterns are proposed to the user conjointly with intruder data, relevant with regards to pattern structure and type. After having selected a message pattern, the user can select between all the proposed data in order to fill the pattern and construct a valid message. Another feature proposed by SPAN is to let the user choose if a message is eavesdropped: the user decides if a message is received by the intruder or not. This makes explicit the way the intruder gain knowledge. This is of great interest for the analysis of protocols like [CHKD06] where the usual assumption – the intruder is the network – is not true.

We now illustrate all these features on the Diffie-Hellman protocol specification of Section 3.1 and show how to construct the usual man in middle attack using SPAN. Starting from the specification, using the intruder mode of SPAN we can reach the situation shown on Figure 10. This figure shows that the user chooses to send the first message of the protocol to the intruder. In the left-hand window, one may note that many messages are likely to be sent by the intruder to `alice`. This is due to the fact that in the Diffie-Hellman HLPSL specification, `alice` waits for a message with no particular structure, i.e. the pattern `X`. Hence, the intruder is able to send any data it already knows (`a, b, exp(g, nonce-1)`, ...) such that `alice` accepts it as a valid instance for `X`. However, the transition we are interested in is still not present because the data `exp(g, ni)` is not yet present in the intruder knowledge. In fact, in SPAN *deduction* mechanisms (e.g. deducing m from $\{m\}_K$ and K) are automatic but *construction* mechanisms (e.g. producing $exp(x,y)$ from x and y) are not. This is due to the fact that deduction converges whereas construction does not. Using a unification algorithm between constructable messages and messages likely to be received could have helped to prune the search space of data to construct. This is used in many tools in the AVISPA system for efficient attack detection. However, our purpose here is not to get automatically the smallest malicious message but rather to let the user build the attack s/he is interested in.

Hence, `exp(g, ni)` is not yet present in the intruder knowledge but it can be easily constructed and sent to `bob` using the intruder message composition interface of Figure 11. In the top of the interface, the user can compose new data from intruder knowledge. On this figure, the user has created a component message

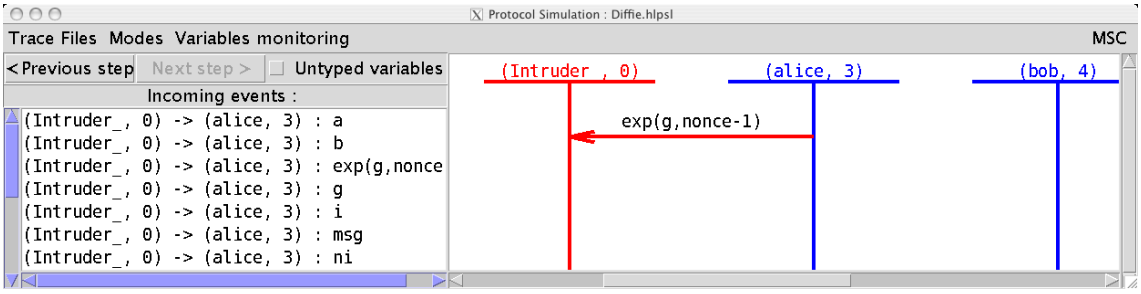


Figure 10: First step of man in the middle attack on Diffie-Hellman protocol

of type `exp` with two arguments: `g` and `ni` of the intruder knowledge which is recalled in the leftmost and rightmost windows. Clicking on the add button adds the new datum to the intruder knowledge: `exp(g,ni)` is now available to the intruder. In the second window of the middle row, the user can now select between message patterns likely to be received by a given agent. Here, the user has chosen to send a message to `bob` and the pattern is only `Y'`. Filling the form by double-clicking on the right data is enough to end the message composition and continue the attack construction.

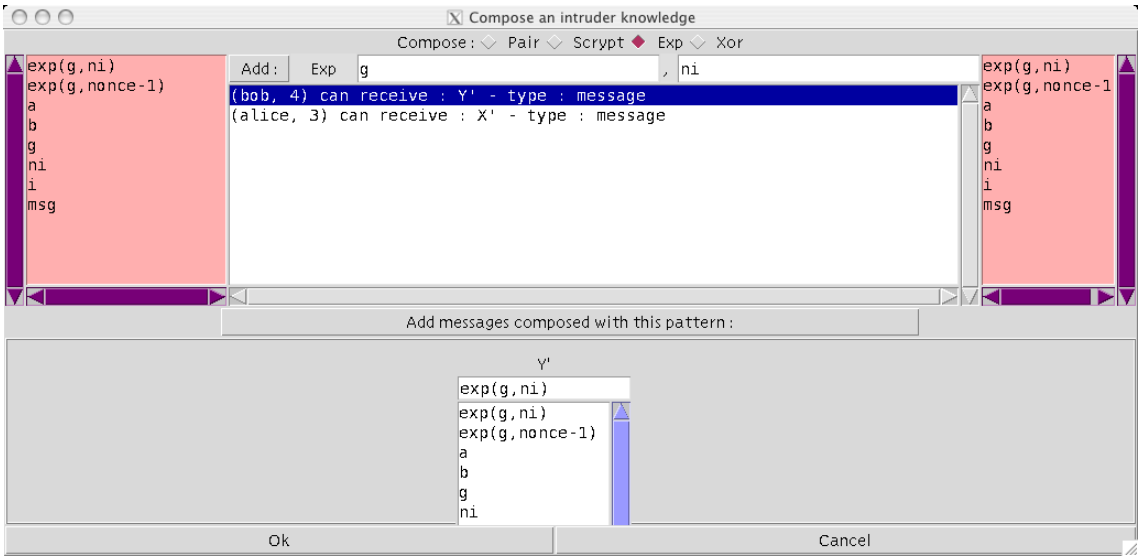


Figure 11: Intruder message composition interface

We can easily continue the construction of the attack until reaching the configuration of Figure 12, where no transition can be fired. We sum up and explain the content of this figure. At step 1, the message sent by `alice` is received by the intruder. At step 2, the intruder generates his half-key `exp(g,ni)` and sends it to `bob`. At step 3, the message sent by `bob` and containing its half-key `exp(g, nonce-2)` is received by the intruder. At step 4, the intruder uses the same half-key than at step 2 and sends it to `alice`. Finally, at step 5, the agent `alice` sends the secret `nonce-3` encoded with the key: `exp(exp(g,ni),nonce-1)`. Until now the intruder has not built the shared key `exp(exp(g,ni),nonce-1)` with the agent `alice` yet. As a result, the intruder cannot obtain the secret `nonce-3`. This can be seen on Figure 13(a) which shows the content of the intruder knowledge window. Using the composition tool as in Figure 11, the user can build `exp(exp(g,nonce-1),ni)` from `exp(g,nonce-1)` and `ni` both known by the intruder. SPAN automatically deduces that `exp(exp(g,nonce-1),ni)` is equivalent to `exp(exp(g,ni),nonce-1)` and can thus obtain `nonce-3` from {`nonce-3`}_`exp(exp(g,ni),nonce-1)` since it has the inverse key. The

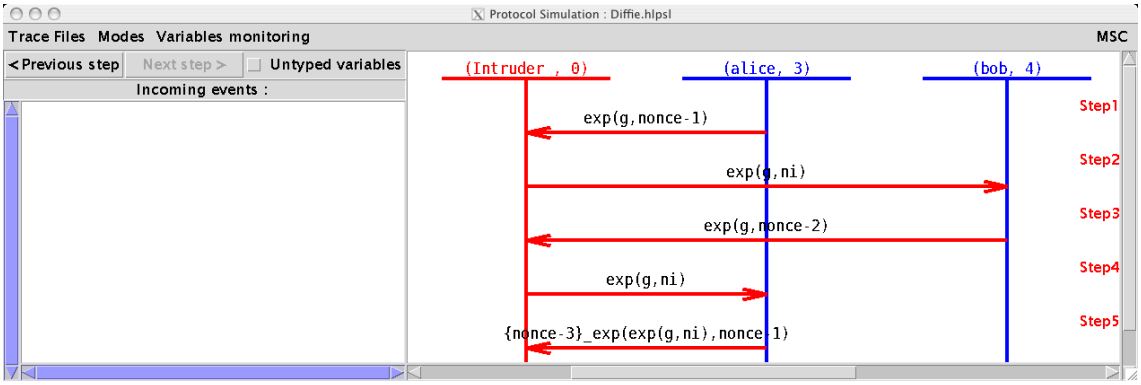
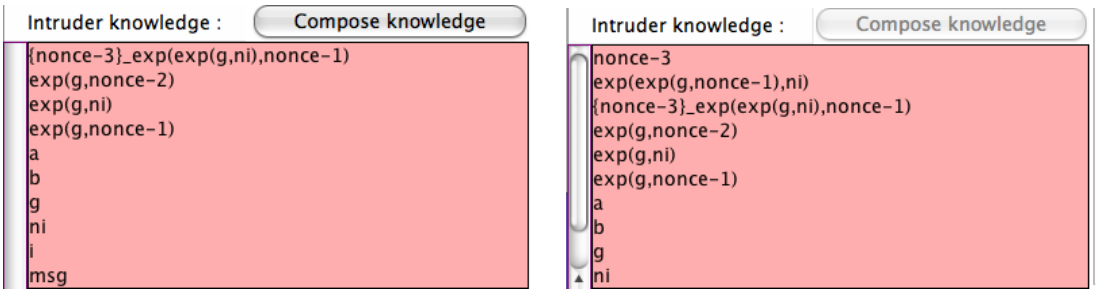


Figure 12: Alice sends the secret `nonce-3`

resulting value of the intruder knowledge containing `nonce-3` is given Figure 13(b).



(a) Before key composition

(b) After decoding

Figure 13: Intruder knowledge evolution using manual construction and automatic deduction

Finally, using the intruder message composition interface on Figure 14, we can see that `bob` waits for a message of the form `{Nsecret'}_exp(exp(g,ni),nonce-2)`. The key `exp(exp(g,ni),nonce-2)` can easily be constructed using `exp(g,nonce-2)`, `ni` and the fact that `exp(exp(g,nonce-2),ni)` is equivalent to `exp(exp(g,ni),nonce-2)`. Then, the message pattern shown in the middle of the figure can be filled up with the relevant elements, i.e. `nonce-3` for `Nsecret'` and `exp(exp(g,nonce-2),ni)` for the key. After filling the pattern, the message can be sent and the protocol attack completed as shown on Figure 15.

Thus, we have shown that SPAN can be used to reconstruct a specific attack from an HLPSL specification. Note that attacks can be saved and replayed. This makes collaborative development of protocols easier. For instance, it is possible to send a particular attack trace to a protocol designer and let him load and replay it. This is useful to illustrate a particular behavior or weakness of a protocol under development.

5 Experiments and further works

By recreating visual information from HLPSL text specifications, SPAN reconciles formal development and the usual way of designing protocols in the industry. This increases the trust of the protocol designers in the automated tools. Moreover, this provides an easy way to build UML style examples and documentation diagrams, not speaking about pedagogic applications. At least, it provides one more good reason to make the effort of writing a full HLPSL specification.

We have applied SPAN to all the protocols of the AVISPA Library and to a new protocol developed by THOMSON called USDP [GGHC06, CHKD06]. During the meetings we had with the protocol designers, we used SPAN to interactively produce MSCs and tune the HLPSL specifications to what they expected of

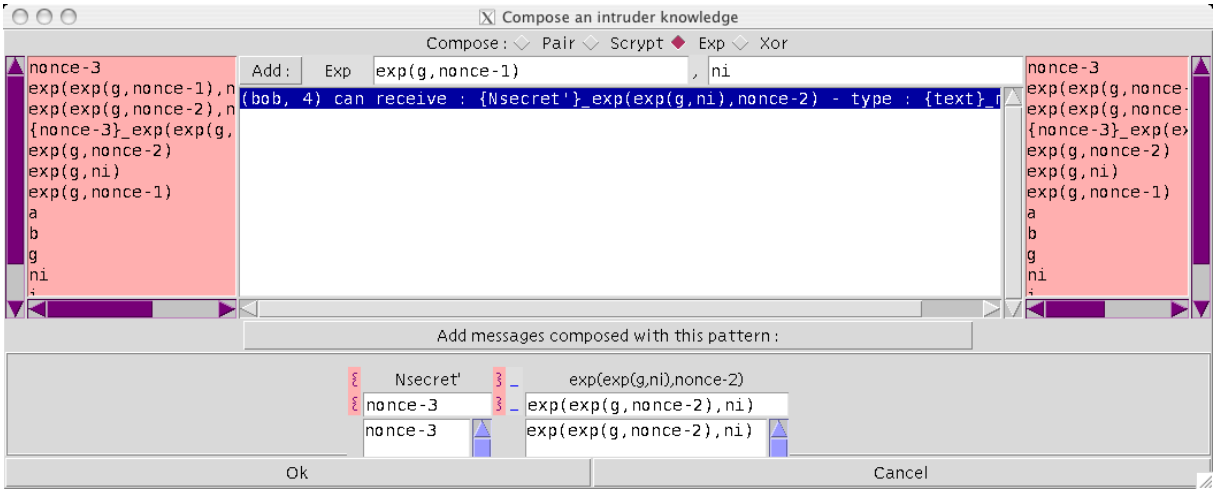


Figure 14: Intruder message composition interface for the last step of the attack

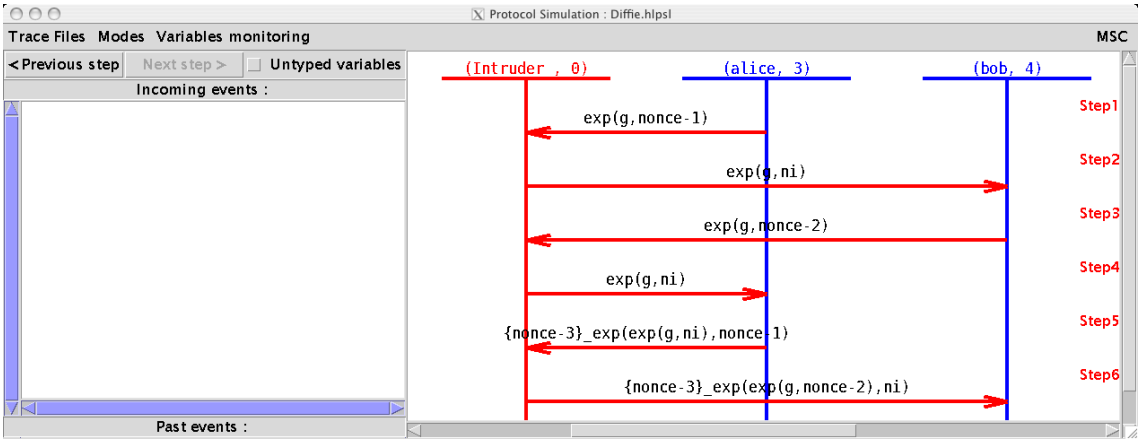


Figure 15: The complete man in the middle attack: the intruder forwards the secret `nonce-3` to `bob`

the protocol. Despite we already agreed on its HLPSL text, this tool permits to quickly reveal and correct misunderstandings remaining in the first version of the specification of USDP. We also observed that using SPAN gives more confidence to the protocol designers on the final HLPSL specification. Finally, the MSCs commonly used by those engineers in the technical documents and patents on protocols can be produced automatically by SPAN from the interactive animation of the HLPSL specification.

Since formal modeling and animation have already demonstrated their interest on the analysis of existing protocols, the aim is now to use HLPSL and SPAN earlier in the development phase of THOMSON's security protocols. In particular, we plan to study the impact of this methodology on the ways of communicating, explaining, experimenting and analyzing a cryptographic protocol.

In further developments, we also would like to automatically build MSCs from the attack traces given by AVISPA verification tools.

References

[ABB+05] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinow-

itch, J. Santos Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification, CAV'2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285, Edinburgh, Scotland, 2005. Springer.

[AC05] Alessandro Armando and Luca Compagna. An optimized intruder model for sat-based model-checking of security protocols. *Electr. Notes Theor. Comput. Sci.*, 125(1):91–108, 2005.

[BHK05] Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Automatic verification of security protocols using approximations. Research Report RR 5727, INRIA, 2005. in revision for Journal of Automated Reasoning.

[BMV05] David A. Basin, Sebastian Mödersheim, and Luca Viganò. Ofmc: A symbolic model checker for security protocols. *Int. J. Inf. Sec.*, 4(3):181–208, 2005.

[CCC+04] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A high level protocol specification language for industrial security-sensitive protocols. In *Proceedings of Workshop on Specification and Automated Processing of Security Requirements (SAPS)*, Linz, Austria, 2004.

[CHKD06] O. Courtay, O. Heen, M. Karroumi, and A. Durand. Secure Device Pairing under Realistic Conditions. In *Industrial Track, Proceedings of Applied Cryptography and Network Security*, pages 41–54, 2006.

[DY83] D.Dolev and A. Yao. On the security of public key protocols. In *Proc. IEEE Transactions on Information Theory*, pages 198–208, 1983.

[GG06] Y. Glouche and T. Genet. SPAN – a Security Protocol ANimator for AVISPA – User Manual. IRISA / Université de Rennes 1, 2006. 20 pages. `http://www.irisa.fr/lande/genet/span/`.

[GGHC06] Y. Glouche, T. Genet, O. Heen, and O. Courtay. A Security Protocol Animator Tool for AVISPA. In *ARTIST-2 workshop on security of embedded systems, Pisa (Italy)*, 2006.

[HT03] D. Harel and P. S. Thiagarajan. Message sequence charts. *UML for Real: Design of Embedded Real-time Systems*, 2003.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[NS78] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *CACM*, 21(12):993–999, 1978.

[Tur06a] Mathieu Turuani. The cl-atse protocol analyser. In Frank Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 2006.

[Tur06b] Mathieu Turuani. The cl-atse protocol analyser. In *RTA*, volume 4098 of *LNCS*, pages 277–286. Springer, 2006.

[WBL06] M. Westergaard and K. Bisgaard Lassen. The britney suite animation tool. In *ICATPN*, volume 4024 of *LNCS*, pages 431–440. Springer, 2006.