Validation of Prouvé protocols using the automatic tool TA4SP [⋆]

Yohan Boichut[1], Nikolai Kosmatov[2], and Laurent Vigneron[3]

[1] LIFC, University of Franche-Comté, boichut@lifc.univ-fcomte.fr
[2] LIFC – CASSIS, INRIA Lorraine, kosmatov@loria.fr
[3] LORIA – University Nancy 2, vigneron@loria.fr

Abstract. We present a new process permitting to automatically analyze security protocols, specified in a very powerful language, PROUVÉ, describing the roles of the participants as real programs. We have built a translator from PROUVÉ specifications to a rule-based language used as input language by several very efficient protocol analyzers. This has permitted us to successfully validate confidentiality properties of several protocols with the TA4SP tool.
Keywords: security protocols, specification, verification, validation.

1 Introduction

The analysis of security protocols has been intensively studied in the last decade. Huge progresses have been achieved in several directions:

– from Alice-Bob specifications to detailed role-based specifications;
– from simple properties to verify, such as secrecy and authentication, to much more complex ones, such as type confusion, non-repudiation or temporal formulas;
– from manual and semi-automatic tools to automatic ones;
– from tools being able to rediscover attacks on a couple of old protocols, to tools analyzing all the protocols of the Clark-Jacob library and a large number of Internet security protocols;
– from verification techniques dedicated to a couple of toy protocols, to general techniques applicable to large real-life protocols and properties;
– from attack search to properties validation.

Among all the created tools, let us cite [4, 11, 13, 10, 7, 14, 6, 2, 12, 1].

This paper presents a work done as part of the PROUVÉ project. We have connected the PROUVÉ protocol specification language [9] to the validation tool TA4SP [3]. The PROUVÉ protocol specification language is not "yet another specification language"; this is one of the most advanced languages, designed for describing protocols very precisely. It looks more like a programming language than other specification languages, but still it is accessible to non specialists.

[⋆] Supported by the RNTL project 03V360 Prouvé, www.lsv.ens-cachan.fr/prouve/.

TA4SP performs the validation of secrecy properties for an unbounded number of sessions; its method consists in solving reachability problems. TA4SP is one of the four back-ends of the AVISPA tool [1]. Its input language is therefore an intermediate format (IF) common with the other back-ends. This IF is a rule-based specification language, describing protocol transitions and agents' knowledge evolution.

We have succeeded to write a translator that converts a large proportion of the PROUVÉ instructions into IF. This connection permits to analyze protocols with TA4SP without any modification in this tool. Note that the use of the other three AVISPA back-ends is then for free!

This paper is organized as follows. In Section 2, we present the PROUVÉ protocol specification language. In Section 3, we describe the translation of PROUVÉ specifications into IF specifications. Section 4 presents TA4SP and some experiments.

2 The Protocol Specification Language Prouvé

The purpose of the PROUVÉ protocol specification language is to give means to describe both protocols and the context in which they are used. A protocol specification in PROUVÉ can be composed of five main sections:

– *signature* for declaring the type of non-predefined constants and constructors,
– *axioms* for defining the semantics of message constructors,
– *roles* for describing the actions of each participant to the protocol,
– *variables* for the declaration of global variables,
– *scenario* describing the combination of roles to be considered.

The sub-language used to define the actions of each role is oriented on existing imperative programming languages. Unlike specification languages based on the simple Alice-Bob notation, PROUVÉ permits to write unambiguous and very precise actions. For example, the generation of a new nonce is explicitly written `new(my_nonce)`, while a technical analysis of an Alice-Bob specification is needed only to understand that a new nonce has to be generated. So, PROUVÉ users write specifications describing the exact behavior of each participant, and there is no need of any complicated compiler for analyzing the specification before transmitting it to a verification tool.

The design of the PROUVÉ protocol specification language has been inspired by the specification language of the AVISPA tool, HLPSL [5]. The main improvements are the possibility to have a global signature section, to explicitly write equational properties of message constructors and to declare global variables; scenarios can be much more complex in PROUVÉ than in HLPSL; and the actions in roles are described as a sequence of instructions in PROUVÉ, while they are described as non-deterministic transitions in HLPSL. Therefore, the PROUVÉ language allows to write deterministic fine-grained specifications, able to handle protocols at an implementation level, and not just as general specifications.

A simple example of specification is given below.

```
signature
    alice, bob, i: principal;
    alice_key, bob_key, intruder_key: pubkey;
end

role Alice (my_name: principal; bob_name: principal;
    my_key: pubkey; bob_pubkey: pubkey)
declare
    my_nonce, bob_nonce: nonce;
begin
    new(my_nonce);
    send(crypt(asym,bob_pubkey,[my_nonce,my_name]));
    recv(crypt(asym,my_key,[my_nonce,bob_nonce,bob_name]));
    send(crypt(asym,bob_pubkey,bob_nonce));
end

role Bob (my_name: principal; alice_name: principal;
    my_key: pubkey; alice_pubkey: pubkey)
declare
    my_nonce, alice_nonce : nonce;
begin
    recv(crypt(asym,my_key,[alice_nonce,alice_name]));
    new(my_nonce);
    send(crypt(asym,alice_pubkey,[alice_nonce,my_nonce,my_name]));
    recv(crypt(asym,my_key,my_nonce));
end

scenario
    begin
        parallel
            Alice (alice,bob,alice_key, bob_key)
            | Bob (bob,alice,bob_key,alice_key)
            | Alice (i,bob,intruder_key,bob_key)
            | Bob (i,alice,intruder_key,alice_key)
        end
    end
end
```

3 Translation from Prouvé into IF

This section describes the implementation of prouve2if, an automatic translator from Prouvé protocol specifications into IF, the input language of the ta4sp tool. We show how we have succeeded to represent most of the structured instructions of standard imperative languages (such as conditionals, iterations, matching) into transition rules.

Prouvé type	IF type
`message`	`message`
`int`	`nat`
`bool`	`bool`
`nonce`	`text`
`principal`	`agent`
`symkey`	`symmetric_key`
`pubkey`	`public_key`
`privkey`	`inv(public_key)`
`algo, symalgo`	—
`list`	—
`table`	—
tuple	pair*
association list	set of pairs

Fig. 1. Translation of the Prouvé types into IF.

3.1 Types, Constants and Variables

The basic Prouvé types are translated into IF as shown in Fig. 1. Since the type for a private key does not exist in IF, we translate it as `inv(public_key)`. The encryption and signature algorithms in IF are omitted.

Concerning structured types, lists and tables are not represented in IF. Association lists may be represented by sets of pairs. Tuples $[a_1, a_2, \ldots, a_{n-1}, a_n]$ ($n \geq 2$) are transformed into right-balanced couples $(a_1, (a_2, \ldots (a_{n-1}, a_n)_{\ldots}))$, and each couple is represented using the binary symbol `pair`. For example, the Prouvé tuple `[x,y,z]` is translated into IF as `pair(V_x,pair(V_y,V_z))`, where we suppose that `x,y,z` are translated as `V_x,V_y,V_z` respectively.

The translation of function types is obtained by direct translating of the function origin and end types. For example, the Prouvé declaration `exp: (pubkey, nonce) -> pubkey` can be translated into IF as `exp: public_key * text -> public_key`. In fact, some common functions and constants (such as `xor`, `exp`, `true`, `false`) are predefined in the IF prelude file and need not to be defined in the IF file.

In general, translating of variables (and in some cases constants and functions) may need their renaming. In Prouvé roles, identifiers may be defined in nested scopes (top-level declarations for all the protocol, input parameters for each role, internal nested declarations in each role). The same identifier may be declared several times with different types like in C. In IF, each declaration is global. This is why the first step of the translating from Prouvé into IF is a proper renaming of the identifiers already defined somewhere before. We add suffixes `_1`, `_2`, ... at the end of the redeclared identifiers in order to distinguish them from the first declaration.

Besides, all identifiers in Prouvé may begin with either a small or a capital letter, whereas in IF constants begin with a small letter and variables with a capital one. Therefore, we also add the prefix `V_` for variables and the prefix `c_` for constants during the translation.

```
role R (me: principal)
declare
    n: nonce;
    x,y: symkey;
begin
    new(n);
    recv([x,y]);
    send([exp(x,n),y,exp(y,n)]);
end
```

Fig. 2. An example of a role in Prouvé.

```
step step_R_1(V_me,V_n,V_x,V_y,SID,Dummy_V_n,Forever) :=
  state_role_R (V_me,1,Dummy_V_n,V_x,V_y,Forever,SID).
  iknows (start)
  =[exists V_n]=>
  state_role_R (V_me,2,V_n,V_x,V_y,Forever,SID)

step step_R_2 (V_me,V_n,V_x,V_y,SID,Dummy_V_x,Dummy_V_y,Forever) :=
  state_role_R (V_me,2,V_n,Dummy_V_x,Dummy_V_y,Forever,SID).
  iknows (pair(V_x,V_y))
  =>
  state_role_R (V_me,3,V_n,V_x,V_y,Forever,SID)

step step_R_3 (V_me,V_n,V_x,V_y,SID,Forever) :=
  state_role_R (V_me,3,V_n,V_x,V_y,Forever,SID)
  =>
  state_role_R (V_me,4,V_n,V_x,V_y,Forever,SID).
  iknows (pair(exp(V_x,V_n),pair(V_y,exp(V_y,V_n))))
```

Fig. 3. Translation of the role of Fig. 2 into IF rules.

3.2 Roles

Each Prouvé role is translated into several IF transition rules which describe the transitions and state changes of an agent playing this role. To illustrate the translating of roles, we consider the role of Fig. 2 and its translation into IF rules given in Fig. 3.

The state of an agent playing a given Prouvé role is defined in IF by a special predicate created for each role and declared in the `signature` section of the IF file. For the role of Fig. 2, the state predicate is:

```
state_role_R : agent * nat * text * symmetric_key * symmetric_key
               * bool * nat -> fact
```

The parameters of the state predicate for a role are put in the following order:

- the input parameters (defined in the role declaration),

– the current state number,
– the local variables of the role,
– the boolean flag Forever (explained below), and
– the session number necessary to the verification tools.

In the simplest case, each instruction of a role is translated into an IF rule (`step`) that may modify the current state number and the values of local variables. The first state has the value 1; it is the origin of the first rule. Any (legitimate) role execution starts in this state. The state value 0 is used to block the role execution; no transition starts from this state. Each rule starts with a unique rule header specifying as parameters the list of variables involved in the rule. We present the translating of the role instructions in more detail in the following sections.

3.3 Basic Instructions

Fig. 4 shows the translating of the most common instructions. We suppose that the instruction given in the left column appears in a role `R`, the state number of the role before this instruction is `A` and the translated IF rule has the number `n`. The new state number `B` is usually just the next available number, except some special cases described below.

Prouvé instruction	Translation into an IF rule
`send(expr)`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...)`
	`  =>`
	`  state_role_R(...,B,...).`
	`  iknows(expr)`
`new(x)`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...,Dummy_V_x,...)`
	`  =[exists V_x]=>`
	`  state_role_R(...,B,...,V_x      ,...)`
`x := expr`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...,Dummy_V_x,...)`
	`  =>`
	`  state_role_R(...,B,...,expr     ,...)`
`recv(pattern)`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...).`
	`  iknows(pattern)`
	`  =>`
	`  state_role_R(...,B,...)`
`fail`	`step step_R_n(...) :=`
	`  state_role_R(...,A,...)`
	`  =>`
	`  state_role_R(...,0,...)`

Fig. 4. Translating of the basic Prouvé instructions into IF rules.

The Prouvé instruction `send(expr)` sends the expression to other agents. It is translated into IF by the special predicate `iknows(expr)` stating that the message `expr` is added to the intruder knowledge. The intruder model used in IF being that of Dolev-Yao, the intruder has access to all the communication channels. So giving him a sent message is a short-cut for sending a message to another agent and intercepting it by the intruder. Thus a normal run of this message sending is the transfer of the message by the intruder to the official receiver.

The Prouvé instruction `new(x)` generates a new value for the variable `x` (which can be of type `int`, `nonce` or `message`). It is translated by `=[exists V_x]=>`, where `V_x` is supposed to be the translation of the variable `x` into IF. Since the value of the variable has changed along this transition, we use a new variable name `Dummy_V_x` to denote the old value.

Similarly, the assignment `x := expr` of the expression `expr` to the variable `x` changes its value to the new one.

The Prouvé instruction `recv(pattern)` blocks the execution of the role until a message that matches the given pattern is received. The constants and previously affected variables in the pattern are just compared to the values in the message. The variables that are not yet instantiated are instantiated along the transition if the matching is successful. Therefore the IF rule translating a `recv` instruction should change the values of such variables putting their `Dummy_` versions in the state predicate before the transition, like it was done for `new` and `:=`. We use the results of a static analysis of the role to check for each variable in a pattern whether this variable has already been instantiated.

The Prouvé instruction `fail` stops the execution of the role. To model this behavior in IF, we change the state to 0. Since no transition can start from this state value, the role execution is stopped. The instructions that follow a `fail` (and cannot be reached in some other way) are ignored.

3.4 The instruction `choice`

The Prouvé instruction

$$\texttt{choice}\ il_1\ \texttt{|}\ il_2\ \texttt{|}\ \texttt{...|}\ il_k\ \texttt{end}$$

executes one list of instructions non-deterministically chosen among the given lists $il_1, \ldots, il_k$, $k \geq 1$. Suppose that A is the state in which the translation of the previous instructions has finished. We proceed in the following way. Every instruction list il_j, $1 \leq j \leq k$, is translated independently into a sequence of IF transition rules starting in some state B_j and ending in some state C_j. The non-deterministic choice of the list to execute is modeled by the *silent transitions* $AB_1, \ldots, AB_k$ whose only effect is passing form the state A to B_j. The before part of these rules is the same.

Fig. 5a) illustrates the translating into IF rules in case $C_j \neq 0$ for all $1 \leq j \leq k$. In this case, in order to finish the translation in the same state, we choose the state C_1 as the last state of the `choice` instruction and add the silent transitions $C_2C_1, \ldots, C_kC_1$.

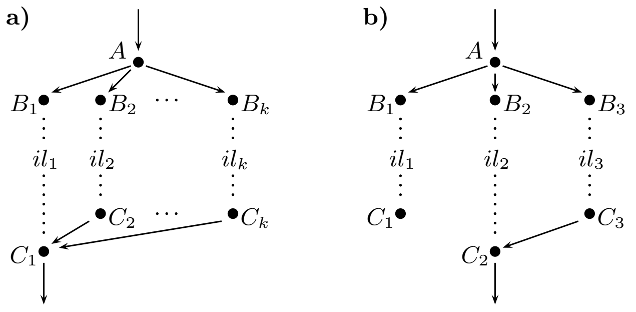

Fig. 5. Translating of `choice` il_1 `|` il_2 `|` ... `|` il_k `end` **a)** for $C_j \neq 0$, $1 \leq j \leq k$; **b)** for $k = 3$, $C_1 = 0$, $C_2 \neq 0$ and $C_3 \neq 0$.

If il_1 finishes in $C_1 = 0$, no transition can follow it. Suppose that we have $C_j \neq 0$ for some $1 \leq j \leq k$. To choose the last state C_i of the `choice` instruction in this case, we take the smallest i such that $C_i \neq 0$ and add the silent transitions $C_jC_i, \dots, C_jC_i$ for all $j \neq i$ with $C_j \neq 0$. For example, Fig. 5b) illustrates the translating into IF rules for $k = 3$, $C_1 = 0$, $C_2 \neq 0$ and $C_3 \neq 0$.

If $C_j = 0$ for all $1 \leq j \leq k$, no instruction can start in any C_j. The last state of the `choice` instruction is 0 independently of the chosen instruction list, and no transition rule is added after any C_j.

3.5 The instruction `if then else`

The Prouvé instruction

`if` *cond* `then` il_1 `( else` il_2 `)? fi`

executes the first list of instructions il_1 if the condition *cond* is verified, or the second one il_2 (or nothing, if the `else` part is not provided) if the condition is not verified.

Suppose that A is the state in which the translation of the previous instructions has finished. We proceed in the following way. Every instruction list il_j, $1 \leq j \leq 2$, is translated independently into a sequence of IF transition rules starting in some state B_j and ending in some state C_j. Fig. 6 illustrates the translating.

The difficulty of the translating for this instruction is the fact that the direct translating of the condition *cond* from Prouvé into IF may be impossible. Indeed, a condition in Prouvé can be any well-typed expression of boolean type, in particular, any composition of disjunctions (denoted in Prouvé by `||`), conjunctions (`&&`), negations (`not`), different (in)equalities (`=,<,>,=<,>=,!=`), other predicates and functions. An IF transition rule can only express a conjunction of conditions each of which is an equality (denoted in IF by `equal`), a less or equal inequality (`leq`) or their negations (`not`).

To be able to translate an `if` instruction for any condition *cond*, we need the following preparatory steps:

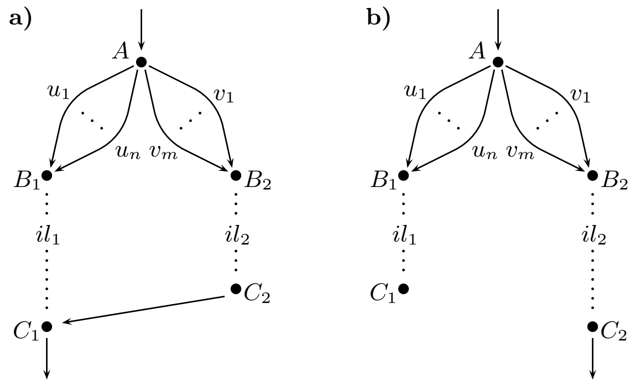

Fig. 6. Translating of `if` *cond* `then` il_1 `else` il_2 `fi` **a)** for $C_1 \neq 0$ and $C_2 \neq 0$; **b)** for $C_1 = 0$ and $C_2 \neq 0$.

```
step step_R_...(...) :=
   state_role_R(...,A ,...) & equal(V_x, 0)
=> state_role_R(...,B1,...)

step step_R_...(...) :=
   state_role_R(...,A ,...) & not(equal(V_x, 0)) & leq(V_y, 10)
=> state_role_R(...,B1,...)

step step_R_...(...) :=
   state_role_R(...,A ,...) & equal(c_g(V_x, V_y, V_z), true))
=> state_role_R(...,B1,...)
```

Fig. 7. Example of transition rules from A to B_1.

- To rewrite the inequalities `<,>,>=,!=` of *cond* in terms of `=,=<` and logical operations. For example, `x >= y` is replaced by `not(x =< y) || (x = y)`.
- To construct the DNF (disjunctive normal form) of the condition *cond*, that is a formula u_1 `||` ... `||` u_n, where the u_j are conjunctions of literals.
- To construct the DNF of `not(`*cond*`)`, that is a formula v_1 `||` ... `||` v_m, where the v_i are conjunctions of literals.

For every $1 \leq j \leq n$ we add an IF transition rule AB_1 guarded by u_j. Similarly, if the `else` part is provided, for every $1 \leq i \leq m$ we add a rule AB_2 guarded by v_i. Since every u_j, v_i is a conjunction of literals, it can be translated into one IF transition rule. A literal `L` that is not one of `=,=<` or their negation, is translated into `equal(L,true)`. For example, Fig. 7 shows the rules from A to B_1 obtained for the instruction:

```
if ( x=0 ) || ( not( x=0 ) && ( y=<10 ) ) || g(x, y, z) then...
```

where we suppose that the PROUVÉ identifiers `x`, `y`, `z`, `g` are translated into IF by `V_x`, `V_y`, `V_z`, `c_g` respectively.

The last state of the translation is chosen as explained before for the `choice` instruction and illustrated in Fig. 6. For example, if $C_1 \neq 0$ and $C_2 \neq 0$, we take C_1 as the last state of the `if` instruction and add the silent transition C_2C_1 (see Fig. 6a)).

3.6 Scenario

The scenario section of a protocol in PROUVÉ describes how the role instances are instantiated. For the moment, we translate only the basic scenario instructions:

- `R(...)` for a simple call of the role R (with the given parameters),
- `parallel R1(...) | ...| Rn(...) end` for the parallel executing of several roles,
- `forever R(...) end` for an infinite sequential loop.

The translating of these instructions is realized as follows. For one or several parallel role calls, we put the corresponding IF state predicates into the initial state in the `inits` section of the IF file. Each IF role is instantiated with the parameters given in the scenario. The `Forever` flag is set to `false`.

To model an infinite sequential loop for a role `R`, we put the corresponding IF state predicate into the initial state with the parameters (given in the scenario) and the `Forever` flag set to `true`. This flag is used as follows. Suppose that $A \neq 0$ is the last state in the translation of the role `R`. We add an additional rule for the role `R` that redirects it from the last state A into its initial state 1 if `Forever` is `true`:

```
step step_R_...(...) :=
   state_role_R(...,A,...,true,SID)
=> state_role_R(...,1,...,true,SID)
```

Any role execution may reach this last rule (if not blocked before) for any initial value of the `Forever` flag because all other rules do not provide (and do not change) the value of the variable `Forever`. This rule can be applied if and only if the role was started with the `Forever` flag set to `true`.

4 Verification using ta4sp

The TA4SP tool, whose method is detailed in [3], is one of the four official tools of the AVISPA tool-set [1]. The originality of this tool is to verify secrecy properties for an unbounded number of sessions. The problem of secrecy verification is handled as a problem of reachability of terms. Theoretically speaking, the protocol and some of the intruder's abilities are specified as a set of rewriting rules denoted by $\mathcal{R}$. The initial knowledge and some other abilities of the intruder as well as instances of sessions are represented by the language $\mathcal{L}(\mathcal{A}_0)$ of a tree

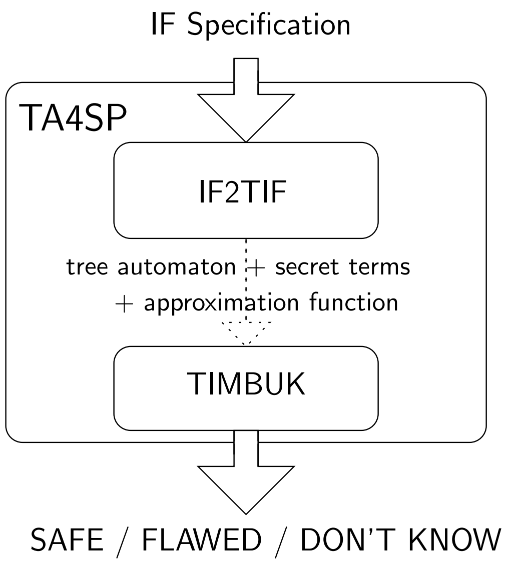

Fig. 8. The TA4SP tool

automaton $\mathcal{A}_0$. As in [8], the approach consists in computing the set of reachable terms from $\mathcal{L}(\mathcal{A}_0)$ using $\mathcal{R}$. This set represents the terms known by the intruder.

In general, the reachability problem is undecidable. The usual approach is to compute an over-approximation of the set of accessible terms. Thus, if a term does not belong to the over-approximation then this term is not accessible by rewriting from $\mathcal{L}(\mathcal{A}_0)$ using $\mathcal{R}$. In the framework of security protocols, a secrecy property is satisfied if all secret terms are not in the over-approximation. On the other hand, a property is not satisfied if at least one secret term belongs to an under-approximation.

Our contribution in [3] consists in applying the fully-automatic verification process described in [8], in particular for generating approximation functions. Specifying a protocol by rewriting rules and a tree automaton remains a tedious task and, moreover, is not well-adapted to a non-expert user.

The structure of the TA4SP tool is shown in Fig. 8. The TA4SP tool is made up of:

– `IF2TIF`, a translator from IF to a specification well-adapted to `TIMBUK+`, and
– `TIMBUK+`[4], a collection of tools for achieving proofs of reachability over term rewriting systems and for manipulating tree automata. This tool has been initially developed by Thomas Genet (IRISA – INRIA Rennes, France) and improved to handle our approximation functions.

TA4SP admits three possible outcomes for a protocol under verification:

[4] Timbuk is available at http://www.irisa.fr/lande/genet/timbuk/.

Protocol	Diagnostic
`NSPKL`	`SECURE`
`NSPK`	`INCONCLUSIVE`
`Needham Schroeder Symmetric Key`	`SECURE`
`Denning-Sacco shared key`	`SECURE`
`Yahalom`	`SECURE`
`TMN`	`INCONCLUSIVE`
`Andrew Secure RPC`	`SECURE`
`Wide Mouthed Frog`	`SECURE`
`Kaochow v1`	`SECURE`

Fig. 9. Verification results obtained by TA4SP.

- The protocol is flawed. This is detected if an under-approximation of the set of reachable terms contains a secret term. TA4SP tries to compute a sufficiently large under-approximation containing at least one of the secret terms.
- The protocol is secure. This is detected if the over-approximation contains no secret term.
- No conclusion can be drawn. This happens if the over-approximation contains a secret term while no under-approximation containing this term can be found.

In the examples of this paper, we restrict TA4SP to over-approximation computations.

As shown in Fig. 8, the TA4SP input is an IF specification. So, the translating of a PROUVÉ protocol into IF allows to verify it with TA4SP. Since the security properties are not yet handled by PROUVE2IF, the generated IF specification needs to be updated by hand in order to complete the initial state with the initial intruder's knowledge and to add the secrecy properties to check.

In Fig. 9, some results obtained with TA4SP are listed. Concerning the protocols `NSPK` and `TMN`, the conclusion `INCONCLUSIVE` is explained by the fact that both protocols are well-known to be flawed.

Notice that the protocols of Fig. 9 are taken from the security protocols open repository (SPORE[5]). The protocols analyzed for this paper do not use all the expressiveness of the PROUVÉ protocol specification language. However, their specification is much more detailed than when using the standard Alice-Bob notation; this guarantees that there is no possible ambiguity in the interpretation of the specification. Validating secrecy properties for such protocols is already an excellent result, and we plan to open TA4SP's horizon to more complicated data structures, algebraic operators, conditional rewriting rules and authentication properties, for considering more complicated and more fine-grained protocols.

[5] These protocols are available at http://www.lsv.ens-cachan.fr/spore/.

5 Conclusion and Future Work

We have described a new process for validating security protocols. It is based on an extremely powerful specification language, PROUVÉ [9], extending the possibilities of existing languages, such as HLPSL [5], for considering implementation-level specifications of real-life protocols. We have built a translator for transforming such specifications into IF, the rule-based language used as input by all the back-ends of the AVISPA tool [1]. Among these back-ends, we have used TA4SP [3] for validating confidentiality properties on several protocols.

This work illustrates the power of rule-based languages, able to represent most of the intructions of imperative programming languages. This connection beetween PROUVÉ and IF permits also to use for free the three other attack search engines of AVISPA.

We have succeeded to validate secrecy properties of several protocols. The extension of the translator PROUVE2IF to handle all the language PROUVÉ will permit to consider much more complex protocols, but also much more complex scenarios.

References

1. A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, 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.
2. A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocol Analysis Tool. In *Proceedings of CAV'02*, LNCS 2404, pages 349–354. Springer, 2002.
3. Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Automatic Verification of Security Protocols Using Approximations. Research Report RR-5727, INRIA-Lorraine - CASSIS Project, October 2005.
4. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
5. 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)*, volume 180, Linz, Austria, September 2004. Oesterreichische Computer Gesellschaft (Austrian Computer Society).
6. G. Denker and J. Millen. CAPSL integrated protocol environment. In *DARPA Information Survivability Conference (DISCEX 2000)*, pages 207–221. IEEE Computer Society, 2000.
7. B. Donovan, P. Norris, and G. Lowe. Analyzing a Library of Security Protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

8. T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proc. 17th CADE Conf., Pittsburgh (Pen., USA)*, volume 1831 of *LNAI*. Springer, 2000.
9. S. Kremer, Y. Lakhnech, and R. Treinen. The PROUVÉ manual: specifications, semantics, and logics. Technical Report 7, Projet RNTL PROUVÉ, December 2005.
10. G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998.
11. C. Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
12. J. Millen and G. Denker. MuCAPSL. In *DISCEX III, DARPA Information Survivability Conference and Exposition*, pages 238–249. IEEE Computer Society, 2003.
13. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1):85–128, 1998.
14. D. Song. Athena: A new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW '99)*, pages 192–202. IEEE Computer Society Press, 1999.