Available online at www.sciencedirect.com
SCIENCE DIRECT®

ELSEVIER Electronic Notes in Theoretical Computer Science 135 (2005) 59–77

Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Towards an Independent Semantics and Verification Technology for the HLPSL Specification Language

Alexey Gotsman [1] Fabio Massacci [1] Marco Pistore [1]

Dip. Informatica e Telecom.
Università di Trento
Trento, Italy

Abstract

We present an algorithm for the translation of security protocol specifications in the HLPSL language developed in the framework of the AVISPA project to a dialect of the applied pi calculus. This algorithm provides us with two interesting scientific contributions: at first, it provides an independent semantics of the HLPSL specification language and, second, makes it possible to verify protocols specified in HLPSL with the applied pi calculus-based ProVerif tool. Our technique has been implemented and tested on various security protocols. The translation can handle a large part of the protocols modelled in HLPSL.

Keywords: Security protocols, verification, specification languages, process algebras

1 Introduction

The last decades have seen a major interest in the verification of security protocols. In almost every theoretical computer science conference there are papers on (un)decidability results and automated reasoning and security conferences show a steady stream of papers on applying this or that verification technology to this or that security protocol.

This steady stream of results has been matched by an equally tumultuous and unruly family of languages for protocol specification. These proposals

[1] Email: {`gotsman, massacci, pistore`}`@dit.unitn.it`

doi:10.1016/j.entcs.2005.06.004

have essentially flowed from a shared concern [2]:

> . . . the formal verification of security protocols became the subject of intense research in the last decade. . .
>
> However, it became apparent that the formalization process itself was a serious bottleneck in the design process. At first, formalizing a protocol was hardly doable by somebody different from the proposer of the formal method itself. Second, the ambiguity of the goals of the protocol made it possible to find "attacks" with formal methods that security analysts will never regard as such [. . .]. Indeed, a security analyst can easily define a "security violation" in terms of sent, received and missing messages, while the language gap between him and the formal method makes it difficult for him to formally and exactly capture what is needed.

So the aim of specification languages was essentially to close the gap between the informal languages in which protocols are specified in security research papers and standard bodies and the formal languages used by experts in formal verification. Among the uncountable number of specification languages proposed in the literature, we cite ALSP [2], BRUTUS [11], CAPSL and its variants [8,9] or its intermediate format CIL [12], CASPER [19], CVS [14], HLPSL [10], NAPTRL [21], the applied pi calculus-based specification languages including ProVerif [6] just to name a few proposals based on different target verification technologies.

Looking retrospectively at these specification languages, it is fair to say that they seldom enjoyed a wide diffusion beyond the proposers or their close scientific collaborators. At first one may think that this was due to the different expressive power of the various languages. However, this would only be justified for general programming languages. Instead, here we have many different languages for the same specialized application domain and, moreover, for almost the same identical set of protocols. We argue that *the major reason behind the lack of diffusion of many specification languages was the lack of an independently motivated and documented semantics.* Loosely speaking, the only available translation was the (not-so-well-documented) translation into the authors' own favorite formal method. In some cases (e. g. CASPER), the protocol description required the direct knowledge and input of construct from the authors' target method. The consequence was that using a different method as a target would be fairly difficult.

The High Level Protocol Specification Language (HLPSL) developed in the framework of the AVISPA project [2] is somehow of an exception. Indeed, while it maintains the customary translation from the protocol description language

[2] `http://www.avispa-project.org`

to the proposers' "proprietary" format (the IF language), it has a first draft of an independently motivated semantics, namely based on Lamport's temporal logic of actions [18].

The HLPSL language is the input language of the AVISPA tool. In more detail, the AVISPA tool takes as input an HLPSL specification, translates it to IF and analyzes the result by invoking state-of-the-art back-ends, which return attacks (if any) to the user. Since different tools are suited for solving different problems, it is relevant to integrate HLPSL with other tools for verification of security protocols. So far the available back-ends of the AVISPA tool have been based on three different technologies that cover a large share of the research arena in security protocol verification:

- bounded model checking and satisfiability [3];
- on-the-fly model checking [5];
- term rewriting [17];
- abstraction-based verification [7].

Letting aside static verification technologies, the major absence in this list is an approach based on a process algebra.

In this paper we propose a way to translate specifications in a subset of HLPSL to the dialect of the applied pi calculus that is supported by the ProVerif tool [6]. This makes it possible to verify secrecy, weak authentication, and strong authentication properties of HLPSL specifications by invoking ProVerif on the result of the translation. It completes the number of available formalisms and technologies available for HLPSL.

It has a further advantages: building on the available draft TLA specifications we provide an independent semantics based on the applied pi-calculus translation of HLPSL. This independent semantics can be used also to clarify a number of ambiguities in the original draft specifications. Indeed, our translation work, done with the support of the University of Genova's members of the AVISPA project, has pointed out a number of semantical issues in the specification and an error in the initial draft semantics for the freshness of nonces.

Our translation is directly based on the source HLPSL language and the TLA semantics rather than the "proprietary" IF language because some attacks can be lost while translating to the IF language. This approach makes it possible to independently check for semantical bugs in the specifications of HLPSL as well as in the translation from HLPSL to IF: protocol attacks must be reflected both in the original AVISPA tools and in our proposed combined verification technology.

The paper is organized as follows. Section 2 introduces the subset of

```
role $B(\mathcal{F})$ played_by $pl$ def=
  init $Init$
  local $\mathcal{L}$
  accept $Acc$
  transition
    $lb_1$. $ev_1$ =|> $act_1$
    $lb_2$. $ev_2$ =|> $act_2$
    ...
    $lb_n$. $ev_n$ =|> $act_n$
end role
```

```
role $P(\mathcal{F})$ def=
const $\mathcal{C}$
composition
  $R_1(\mathcal{A}_1) \wedge \ldots \wedge R_n(\mathcal{A}_n)$
end role
```

Fig. 1. Generic structure of HLPSL roles

HLPSL that we use. In section 3 we describe the dialect of the applied pi calculus to which the translation is performed. Section 4 describes the algorithm of the translation. Section 5 explains how ProVerif can be used to verify the outcome of the translation while section 6 gives some details of the implementation and the experimental results on a selection of cryptographic protocols that were modelled in HLPSL in the framework of the AVISPA project.

2 The HLPSL Language

In this section we briefly describe the subset of HLPSL that is handled by our translation algorithm. We refer the reader to [10] for a fuller description of the language.

Protocol specifications in HLPSL are divided into roles. Some roles (the so-called basic roles) serve to describe the actions of one single agent in a run of the protocol or subprotocol. Others (composed roles) instantiate basic roles to model an entire protocol run, a session of the protocol between multiple agents, or the protocol model itself. This latter role is called the main role.

Figure 1 shows the structure of a basic role (denoted with B) and of a composed role (denoted with P) in HLPSL. $\mathcal{F}$ is the set of formal arguments, $\mathcal{L}$ is the set of local variables, pl is the agent playing B. We require that $\mathcal{F} \cap \mathcal{L} = \emptyset$.

In HLPSL, both variables and constants are typed. We will use the following types: `agent`, `channel`, `public_key`, `text`, `nat`, `bool`, `symmetric_key`, `message`, `function`, `protocol_id`. Certain types (`text` and `channel`) may have attributes enclosed within parentheses. For example, `text(fresh)` represents the type of freshly generated nonces and `channel(dy)` the type of channels that correspond to Dolev-Yao (DY) intruder model [13]. In our

$t ::=$		HLPSL terms
	V	variable
	V'	new value of the variable V
	c	constant
	$t_1.t_2$	concatenation of t_1 and t_2
	$\{t_1\}t_2$	encryption of the message t_1 by the key t_2
	$f(t)$	application of the function stored in the variable or constant f (of type `function`) to the term t
	$\mathrm{inv}(t)$	the private key corresponding to the public key t

Fig. 2. The syntax of HLPSL terms

subset of HLPSL we consider only DY-channels, so by default all channels implicitly have the `dy` attribute.

Terms in HLPSL are constructed as it is shown in figure 2. Here V' is used to denote the value of the variable V after the execution of a transition, while V denotes the value of the variable before the execution. Besides, for a variable V with the `fresh` attribute V' denotes a newly generated value for this variable. Note also that $\{t_1\}t_2$ is considered to be public-key encryption, if t_2 is x or $\mathrm{inv}(x)$, where x is a variable or a constant of the type `public_key` and shared-key encryption otherwise. In the former case t_2 is called a public-key expression.

The `init` section assigns initial values to local variables and has the following structure:

$$Init \triangleq \bigwedge_{i=1}^{n} (V_i = c_i)$$

where V_i are variables, c_i are constants. Let $\mathcal{I}$ be the set of variables assigned values in the `init` section. We require that $\mathcal{I} \subseteq \mathcal{L}$.

A transition is defined by a label lb_i, a trigger event ev_i, and an action act_i. Each transition is triggered whenever its guard event predicate is satisfied and fires immediately. In case when a few events are satisfied, one of the transitions corresponding to them is chosen non-deterministically and executed.

We require that each basic role have a local variable `State` of type `nat` that intuitively represents the state of the participant of the protocol and that

ev_i and act_i have the following form

$$ev_i \triangleq \texttt{State} = s \wedge Comp \wedge RMsg$$

$$act_i \triangleq \texttt{State}' = s_1 \wedge SMsg \wedge Ass \wedge Witness \wedge WRequest \wedge Request \wedge Secret$$

(the order of conjuncts is not fixed, but the set of conjuncts is). We will also require that $\texttt{State} \in \mathcal{I}$.

It is worth noting that the HLPSL specification does not make such a restriction on the form of transitions but we found this assumption extremely useful. The requirement that each role have its own state explicitly declared allows us to optimize the translation of HLPSL specifications and map back the results of the verification into the protocol domain. Furthermore, we found out that *all* publicly available HLPSL specifications are defined using the `State` variable in the way just said.

Let us now describe the parts of a transition.

$Comp$ is a set of comparisons and is defined to be

$$Comp \triangleq \bigwedge_{i=1}^{n} (t_i = m_i)$$

where t_i and m_i are HLPSL terms without primed variables.

$RMsg$ and $SMsg$ are the sets of receive actions and send actions respectively. They have the following form:

$$RMsg \triangleq \bigwedge_{i=1}^{n} c_i(t_i)$$

$$SMsg \triangleq \bigwedge_{i=1}^{n} d_i(m_i)$$

where c_i and d_i are variables or constants of type `channel`, t_i and m_i are HLPSL terms.

Ass is the set of assignments of new values to variables and has the following structure:

$$Ass \triangleq \bigwedge_{i=1}^{n} (V_i' = m_i)$$

where V_i are variables, m_i are HLPSL terms.

$Witness$, $WRequest$, $Request$, and $Secret$ are conjunctions of expressions of the forms $\texttt{witness}(a, b, p, m)$, $\texttt{wrequest}(a, b, p, m)$, $\texttt{request}(a, b, p, m)$, and $\texttt{secret}(m, a)$ respectively. These expressions are called goal facts and are used to express correctness requirements. Intuitively, a goal fact is a predicate that becomes true at the moment he appears on the right-hand side of the transition being executed. Here a and b are variables or constants of the type `agent`, m is an HLPSL term and p is a protocol identifier – a constant of the type `protocol_id` that is declared in the main role and is used to determine the correctness requirement that corresponds to the goal fact. Intuitively, $\texttt{witness}(a, b, p, m)$ means that the agent a wants to execute the protocol with the agent b using m as the value for the authentication identifier p. $\texttt{wrequest}(a, b, p, m)$ or $\texttt{request}(a, b, p, m)$ mean that the agent a accepts the value m and now relies on the guarantee that b exists and agrees with him on this value for the authentication identifier p. Whether to use `wrequest` or `request` depends on the type of correctness requirement being formulated (see section 5). We use `wrequest` when we verify weak authentication and `request` when we verify strong authentication. $\texttt{secret}(m, a)$ means that the honest player of the role claims that the term m is a secret that the agent a is allowed to know.

The `accept` section has the form

$$Acc \triangleq \texttt{State} = s_a$$

where s_a is the state corresponding to the successful completion of the protocol in which the role is participating.

In composed roles we have no `transition` section. Rather, we have a section entitled `composition` that instantiates other roles R_i (with sets of arguments $\mathcal{A}_i$) running in parallel. Besides, a composed role may define a set of constants $\mathcal{C}$.

In addition, the main role must have declared a constant `i` of the type `agent` that represents the intruder. It may also have a declaration of the form `knowledge(i) =` $\mathcal{K}$, which describes the initial knowledge of the intruder. Here $\mathcal{K}$ is a set of HLPSL terms.

Correctness requirements for the protocol, which are called goals, are given in their own section. Three types of goals are supported:

- Secrecy of an HLPSL term m: `secrecy_of` m. This means that m must not be known to the intruder.
- Weak authentication: A `weakly authenticates` B `on` p. This means that whenever an agent b playing the role B executes $\texttt{wrequest}(b, a, p, m)$, it is true that either $a = \texttt{i}$ or a is an agent playing the role A and $\texttt{witness}(a, b, p, m)$ has been executed previously. This corresponds to Lowe's notion of non-injective agreement [20].
- Strong authentication: A `authenticates` B `on` p. In comparison with the

t ::=		terms
	x	variable
	$f(t_1, \ldots, t_n)$	constructor application
P ::=		processes
	nil	deadlock
	$x!\langle t\rangle.P$	output
	$x?(y).P$	input
	if $t_1 = t_2$ **then** P **else** Q	comparison
	if $t_1 \neq t_2$ **then** P **else** Q	comparison
	$P_1 \vert P_2$	parallel composition
	$P_1 + P_2$	choice
	$(\nu x)P$	restriction
	let $x = g(t_1, \ldots, t_n)$ **in** P **else** Q	destructor application
	event $e(t_1, \ldots, t_n).P$	event e
	event_ex $e(t_1, \ldots, t_n)$	executed event e

Fig. 3. The syntax of the target language

previous goal this adds the requirement that each `wrequest` executed by an agent playing B correspond to unique `witness` executed by an agent playing A. This corresponds to Lowe's notion of injective agreement [20].

3 The Target Applied Pi Calculus

As the target language of our translation we use a dialect of the applied pi calculus similar to the one presented in [1]. This dialect extends the classical pi calculus defining cryptographic primitives by reduction relations. Its syntax is reported in figure 3.

The variable x is bound in P in the processes $x?(y).P$, $(\nu x)P$, and

$$\textbf{let } x = g(t_1, \ldots, t_n) \textbf{ in } P \textbf{ else } Q$$

We will use $(\nu\mathcal{M})$, where $\mathcal{M} = \{x_1, \ldots, x_n\}$, as syntactic sugar for $(\nu x_1) \ldots (\nu x_n)$

Tuples

Constructor: tuple, $n\text{tuple}(t_1, \ldots, t_n)$

Destructors: projections, $i\text{th}_n(n\text{tuple}(t_1, \ldots, t_n)) \rightarrow t_i$

Shared-key encryption

Constructor: encryption of t_1 by the key t_2, $\text{sencrypt}(t_1, t_2)$

Destructor: decryption, $\text{sdecrypt}(\text{sencrypt}(t_1, t_2), t_2) \rightarrow t_1$

Public-key encryption

Constructors: encryption of t_1 by the public key t_2, $\text{pencrypt}(t_1, t_2)$

the private key, corresponding to the public key t, $\text{inv}(t)$

Destructor: decryption, $\text{pdecrypt}(\text{pencrypt}(t_1, t_2), \text{inv}(t_2)) \rightarrow t_1$

One-way hash functions

Constructor: application of the function f to the term t, $\text{hashfun}(f, t)$

Fig. 4. Constructors and destructors

and **let** $x = g(t_1, \ldots, t_n)$ **in** P as syntactic sugar for

$$\textbf{let } x = g(t_1, \ldots, t_n) \textbf{ in } P \textbf{ else nil}$$

The syntax assumes that a set of constructors, destructors, and events are defined. Constructors are used to build terms. Destructors are used in processes to manipulate terms. They are partial functions on terms that processes can apply. The process **let** $x = g(t_1, \ldots, t_n)$ **in** P **else** Q tries to evaluate $g(t_1, \ldots, t_n)$; if this succeeds, then x is bound to the result and P is executed, else Q is executed. Evaluation of g is defined by a set of reduction rules of the form $g(t_1, \ldots, t_n) \rightarrow t$.

We use destructors rather than equational theories of the applied pi calculus because this approach allows for more efficient verification and is used in the ProVerif tool with which the outcome of our translation is verified.

For purposes of our translation we define constructors and destructors for representing data structures and cryptographic primitives as it is shown in figure 4. We define **let** $(x_1, \ldots, x_n) = t$ **in** P as syntactic sugar for

$$\textbf{let } x_1 = 1\text{th}_n(t) \textbf{ in } \ldots \textbf{ let } x_n = n\text{th}_n(t) \textbf{ in } P$$

Moreover, we will abbreviate $n\text{tuple}(t_1, \ldots, t_n)$ to $(t_1, \ldots, t_n)$.

$$
\begin{array}{ll}
(P|Q)|R \equiv P|(Q|R) & P|0 \equiv P \\
(\nu a_1)(\nu a_2)P \equiv (\nu a_2)(\nu a_1)P & !P \equiv P|!P \\
(P+Q)+R \equiv P+(Q+R) & P+0 \equiv P \\
(\nu a)(P|Q) \equiv P|(\nu a)Q, \text{ if } a \text{ is not free in } P &
\end{array}
$$

Fig. 5. Structural congruence

$$
\begin{array}{ll}
a!\langle t\rangle.Q|a?(x).P \to Q|P\{t/x\} & P \to Q \Rightarrow P|R \to Q|R \\
P' \equiv P, P \to Q, Q \equiv Q' \Rightarrow P' \to Q' & P \to Q \Rightarrow (\nu a)P \to (\nu a)Q \\
\textbf{if } t = t \textbf{ then } P \textbf{ else } Q \to P & \textbf{let } x = t \textbf{ in } P \to P\{t/x\} \\
\textbf{if } t_1 = t_2 \textbf{ then } P \textbf{ else } Q \to Q, \text{ if } t_1 \neq t_2 & P \to Q \Rightarrow P+R \to Q+R \\
\textbf{if } t \neq t \textbf{ then } P \textbf{ else } Q \to Q & \\
\textbf{if } t_1 \neq t_2 \textbf{ then } P \textbf{ else } Q \to P, \text{ if } t_1 \neq t_2 & \\
\textbf{let } x = g(t_1,\dots,t_n) \textbf{ in } P \textbf{ else } Q \to P\{t'/x\} \text{ if } g(t_1,\dots,t_n) \to t' & \\
\textbf{let } x = g(t_1,\dots,t_n) \textbf{ in } P \textbf{ else } Q \to Q & \\
\quad \text{if there exists no } t' \text{ such that } g(t_1,\dots,t_n) \to t' & \\
\textbf{event } e(t_1,\dots,t_n).P \to \textbf{event\_ex } e(t_1,\dots,t_n)|P &
\end{array}
$$

Fig. 6. Reduction

Events are used in authentication goals. The constructs **event_ex** are useful to remember what events were executed. We define three events (witness, wrequest, and request) with four arguments each, which correspond to HLPSL goal facts with the same names. In addition, we define event player(a, r) meaning that the agent a plays the role r. The roles of these events will become clear in the latter sections.

Formal semantics of the applied pi calculus is given in terms of structural congruence and reduction. The structural congruence for the applied pi calculus is the smallest congruence that satisfies the equations in figure 5. The axioms for the reduction are listed in figure 6. We also let $\text{inv}(\text{inv}(t)) = t$.

4 Translation algorithm

The translation of the composed roles in an HLPSL specification is done by flattening their structure starting from the main role. In this way we obtain only instantiations of basic roles with constants as arguments.

Our translation of basic roles uses the `State` variable. A basic role is translated to a set of processes each acting as the role in a particular state.

There are two important details to note about the translation algorithm.

First, since we are using DY intruder model, it is irrelevant from which channel we receive a message or to which channel we send it. Therefore, in the translation of receive and send actions we receive messages from and send messages to a predefined channel c, which is a free constant unknown to the intruder.

Second, we will use the restriction operator of the pi calculus to model the generation of new values for variables with the `fresh` attribute.

Now we will describe the translation algorithm. We will denote the translation of a fragment R of an HLPSL specification as $[\![R]\!]$ and will use the notation for parts of an HLPSL specification introduced in section 2.

First of all we describe the translation of HLPSL terms. Their translation is defined according to their structure:

$$[\![V]\!] \triangleq V,\ [\![V']\!] \triangleq V,\ [\![c]\!] \triangleq c,\ [\![t_1.\ \ldots\ .t_n]\!] \triangleq ([\![t_1]\!], \ldots, [\![t_n]\!])$$

$$[\![\{m\}t]\!] \triangleq \begin{cases} \text{pencrypt}([\![m]\!], [\![t]\!]), \text{ if } t \text{ is a public-key expression;} \\ \text{sencrypt}([\![m]\!], [\![t]\!]),\ \text{otherwise} \end{cases}$$

$$[\![f(t)]\!] \triangleq \text{hashfun}([\![f]\!], [\![t]\!]),\ [\![\text{inv}(t)]\!] \triangleq \text{inv}([\![t]\!])$$

The instantiations of the composed roles in an HLPSL specification form a tree with the main role as the root. To translate the composed roles we flatten this tree by repeatedly substituting the texts of composed roles for their instantiations and actual arguments for formal arguments, and using the parallel composition operator of the pi calculus to translate parallel compositions of role instantiations. Besides, we rename all the constants declared in the composed roles to avoid name clashes and declare them as free constants unknown to the intruder. In addition, for each basic role instantiation we introduce a constant representing an identifier for this instantiation. We add a formal argument SID to each basic role and assign the instantiation identifier to this argument in each instantiation. The instantiation identifier we be used while translating strong authentication goals.

Having done these preliminary steps, we can generate the translation of the main role (and all the composed roles) in the following way:

(i) First, we generate the process that outputs the initial knowledge of the intruder on the channel c:

$$c!\langle t_1\rangle.\ \dots\ .c!\langle t_k\rangle$$

where $\{t_1, \dots, t_k\} = \mathcal{K}$.

(ii) Second, for each basic role R we introduce a free constant unknown to the intruder representing an identifier for this role. Since after flattening we obtain only the instantiations of basic roles with constants as arguments, we can determine (using `played_by` declaration) for each basic role the set of agents playing the role. For each basic role identifier r and each agent a playing the role denoted with r we generate the processes of the form

$$\mathbf{event}\ \mathrm{player}(a, r)$$

and join them (as well as the process obtained on the previous step) with the . operator.

(iii) Finally, we output the result of flattening of the tree of composed roles using the **let** operator to assign values to formal arguments of basic roles.

A basic role B is translated to the following process:

$$B \stackrel{\triangle}{=} (\nu\mathcal{L}\backslash\mathcal{I})(B_{s_0}\{c_1/V_1, \dots, c_n/V_n\})$$

where s_0 is the value given to the `State` variable in the `init` section, V_i are the variables initialized in the `init` section by c_i.

For each of the values s of the `State` variable, assigned to it in the HLPSL text of the role we create a process B_s acting as the role in the state s. Let $Tr(s)$ be the set of all the transitions that have the conjunction $\texttt{State} = s$ on their left-hand sides and let $\mathcal{N}_k$ be the set of fresh variables updated by the k-th transition (i. e. the set of those variables that have the `fresh` attribute and are primed on the right-hand side of the transition). Then

$$B_s \stackrel{\triangle}{=} \sum_{k\in Tr(s)} [\![ev_k]\!](\nu\mathcal{N}_k)[\![act_k]\!]$$

The translation of events and actions is defined as follows:

$$[\![\texttt{State} = s \wedge Comp \wedge RMsg]\!] \stackrel{\triangle}{=} [\![Comp]\!][\![RMsg]\!]$$

$$[\![\texttt{State} = s_1 \wedge SMsg \wedge Ass \wedge Witness \wedge WRequest \wedge Secret]\!] \stackrel{\triangle}{=}$$
$$[\![Witness]\!].[\![Ass]\!][\![SMsg]\!].[\![WRequest]\!].[\![Request]\!].[\![Secret \wedge \texttt{State} = s_1]\!]$$

In addition, we define $B_{s_a} = \mathbf{nil}$ where s_a is the value of the `State` variable in the `accept` section.

Let us describe the translation of parts of a transition.

The translation of *Comp* is the following:

$$[\![Comp]\!] \triangleq \textbf{if } [\![t_1]\!] = [\![m_1]\!] \textbf{ then } \ldots \textbf{ if } [\![t_n]\!] = [\![m_n]\!] \textbf{ then}$$

In our translation of receive actions we first receive messages to newly created variables (using the applied pi calculus primitives) and then match their structures against the ones given in the HLPSL specification. This latter part of the process is defined by the mapping $\mathbf{M}(\cdot,\cdot)$, in which the first argument is the name to which the received term is bounded and the second argument is the HLPSL term describing its structure. I. e.

$$[\![RMsg]\!] \triangleq c?(t_1^*).\ \ldots\ .c?(t_n^*).\mathbf{M}(t_1^*, t_1).\ \ldots\ .\mathbf{M}(t_n^*, t_n)$$

If t_i is a primed variable, then t_i^* is this variable without the prime. Otherwise, it is a new variable that has not been used previously in the process of translation. $\mathbf{M}(\cdot,\cdot)$ is defined according to the structure of HLPSL terms in the following way:

$$\mathbf{M}(m, V) \triangleq \textbf{if } m = V \textbf{ then}$$

$$\mathbf{M}(m, c) \triangleq \textbf{if } m = c \textbf{ then}$$

$$\mathbf{M}(m, m_1.\ \ldots\ .m_n) \triangleq \textbf{let } (m_1^*, \ldots, m_n^*) = m \textbf{ in } \mathbf{M}(m_1^*, m_1) \ldots \mathbf{M}(m_n^*, m_n)$$

$$\mathbf{M}(m, \{t\}x) \triangleq$$

$$\begin{cases} \textbf{let } t^* = \text{pdecrypt}(m, \text{inv}(x)) \textbf{ in } \mathbf{M}(t^*, t), & \text{if } x \text{ is a public-key expression;} \\ \textbf{let } t^* = \text{sdecrypt}(m, x) \textbf{ in } \mathbf{M}(t^*, t), & \text{otherwise} \end{cases}$$

m_i^* and t^* are defined as before (with respect to m_i and t). Besides, $\mathbf{M}(m, V')$ yields the empty formula and is discarded.

The translation of *SMsg* is defined as follows:

$$[\![SMsg]\!] \triangleq c!\langle[\![m_1]\!]\rangle.\ \ldots\ .c!\langle[\![m_n]\!]\rangle$$

We define the translation of *Ass* to be

$$[\![Ass]\!] \triangleq \textbf{let } V_1 = [\![m_1]\!] \textbf{ in } \ldots \textbf{ let } V_n = [\![m_n]\!] \textbf{ in}$$

`witness` and `wrequest` goal facts are translated in the following way

$$[\![\texttt{witness}(a, b, p, m)]\!] \triangleq \textbf{event } \text{witness}(a, b, p, [\![m]\!])$$

$$[\![\texttt{wrequest}(a, b, p, m)]\!] \triangleq \textbf{event } \text{wrequest}(a, b, p, [\![m]\!])$$

A `request` goal fact has the following translation

$$[\![\texttt{request}(a,b,p,m)]\!] \overset{\triangle}{=}$$

$$c!\mathrm{sencrypt}((a,b,p,m,SID),sidkey).\mathbf{event}\ \mathrm{request}(a,b,p,[\![m]\!])$$

Here *sidkey* is a free constant unknown to the intruder.

A conjunction $G_1 \wedge \ldots \wedge G_n$ of `witness`, `wrequest`, and `request` goal facts G_i is translated to $[\![G_1]\!].\ \ldots\ .[\![G_n]\!]$.

The translation of the remaining part of a transition is defined by the following recursive equations:

$$[\![\texttt{secret}(m,a_1) \wedge \ldots \texttt{secret}(m,a_n) \wedge R]\!] \overset{\triangle}{=}$$

$$\mathbf{if}\ a_1 \neq \texttt{i}\ \mathbf{then}\ \ldots\ \mathbf{if}\ a_n \neq \texttt{i}\ \mathbf{then}\ (c!\mathrm{sencrypt}(r_m,[\![m]\!]).[\![R]\!])$$
$$\mathbf{else}\ [\![R]\!]\ \ldots\ \mathbf{else}\ [\![R]\!]$$

$$[\![\texttt{State} = s_1]\!] \overset{\triangle}{=} B_{s_1}$$

Here R is a conjunction of HLPSL expressions, r_m is a special constant created for each message m declared secret and used in formulating secrecy goals. While translating *Secret* we group `secret` goal facts by the expression that is being made secret. Note, that we have to guard each event with the restriction that neither of the participants is the intruder `i` because the correctness requirements are meaningless in the this case. We can create an additional process defined as $[\![R]\!]$ to avoid the duplication of the code.

It is worth noting that currently the semantics of HLPSL is not clearly defined. Therefore, in some cases we had to make the decision of how to interpret HLPSL constructs. In this cases we tried to follow the semantics that is implemented in the AVISPA tool.

For example, in our subset of HLPSL it is not possible to model a situation in which a role receives a private key through a channel and then decrypts a message using this private key, i. e. a role has static knowledge. This is due to the fact that the private key must be known to the role at the beginning of its execution and written in the text of the role using the inv constructor. This is compatible with the AVISPA tool, which does not support processing of such a situation at the moment.

One more issue that we had to resolve was whether the values for variables with the `fresh` attribute should be generated each time the transition containing the variable is performed or each time the role is instantiated. In our translation we create the values as fresh names in the applied pi calculus each time the transition is performed.

5 Verification with ProVerif

The result of the translation described in the previous section can be verified with the ProVerif tool [6], which is a theorem proving system based on an intermediate representation of the protocol by a set of Horn clauses (a logic program). ProVerif has the input language similar to the one described in section 3. The differences between these two languages are the following. ProVerif does not support the + operator, but offers built-in unification in the **let** operator. Besides, ProVerif allows for declaring free names in specifications, which can be public (known to the intruder) and private (unknown to the intruder). The model of the intruder in ProVerif is DY with the exception that some constructors can be declared private, so that the intruder cannot use them to create new messages. Moreover, ProVerif allows for specifying equations between terms. However, the treatment of equations in ProVerif is still preliminary, which may result in non-termination of the tool in certain cases.

ProVerif allows for specifying different types of goals (which are called queries). For our translation we will use only some of them.

Queries are build out of facts, which can have the following form:

- $\texttt{attacker:}t$ – is true if and only if the intruder may know t.
- $t_1 = t_2$ – is true if and only if the terms t_1 and t_2 are equal.
- $\texttt{ev:}f(x_1, \ldots, x_n)$ – is true if and only if the event $f(x_1, \ldots, x_n)$ has been executed.

Given facts F and $F_{i,j}$ $(1 \le i \le k,\ 1 \le j \le k_i)$ we can construct a query

$$F \texttt{ ==> } (F_{1,1} \ \& \ \ldots \ \& \ F_{1,j_1}) \mid \ldots \mid (F_{k,1} \ \& \ \ldots \ \& \ F_{k,j_k})$$

which is true if and only if, when F is true, then for some i $F_{i,m}$ is true for all m such that $1 \le m \le j_i$. The query can also be a fact F.

For example, the query

$$\texttt{query attacker:}t$$

is true when the intruder may know t and the query

$$\texttt{query ev:}f_1(x_1, \ldots, x_n) \texttt{ ==> ev:}f_2(y_1, \ldots, y_n)$$

is true when, if the event $f_1(x_1, \ldots, x_n)$ has been executed, then the event $f_2(y_1, \ldots, y_n)$ must have been executed before it.

In our translation we declare inv as a private constructor with the equation $\mathrm{inv}(\mathrm{inv}(x)) = x$, the channel c as a public free name and all the other constants as private free names.

A secrecy goal `secrecy_of` m in HLPSL is translated to

$$\texttt{query attacker:}r_m$$

since from the translation algorithm it follows that the intruder can know r_m if and only if it knows m. The need for r_m is due to the fact that one needs a free name to be used in `query` and the terms declared secret cannot be used for this purpose because they are created inside processes.

A weak authentication goal A `weakly authenticates` B `on` p in HLPSL is translated to the set of queries of the form

$$\texttt{query ev:wrequest}(b, y, p, x) \texttt{ ==>}$$

$$(\texttt{ev} : \texttt{witness}(y, b, p, x) \ \& \ \texttt{ev:player}(y, aid)) \mid y = i.$$

for each agent b playing the role B. Here aid is the role identifier of the role A. Agents playing roles and role identifiers are determined while translating composed roles. We have to put the variable y as the second argument to `wrequest` in this query instead of an agent playing the role A because we have to take into account the situation in which `wrequest` is executed with a non-existing agent (created by the intruder) as its second argument.

A strong authentication goal A `authenticates` B `on` p is translated to the set of queries including those for the corresponding weak authentication goal (with `request` substituted for `wrequest`) and queries of the form

$$\texttt{query attacker:}(\mathrm{sencrypt}((b, a, p, x, y_1), sidkey),$$

$$\mathrm{sencrypt}((b, a, p, x, y_2), sidkey)) \texttt{ ==> } y_1 = y_2.$$

for each agent b playing the role B and agent a playing the role A. Since the messages of the form $\mathrm{sencrypt}((b, a, p, x, y), sidkey)$ are sent only when the corresponding `request` event is executed, these additional queries ensure that no data will be used twice for authentication in two different sessions (defined by different SID values y_1 and y_2) and, hence, ensure strong authentication. We have to use the queries of the form `query attacker:`t because ProVerif does not allow for using conjunctions of facts on the left-hand side of the `==>` operator. We have to encrypt the data used in the `request` event with $sidkey$ unknown to the intruder so that the intruder cannot use them.

Our implementation translates HLPSL specifications from the subset of HLPSL described in section 2 to the language of ProVerif. Since ProVerif does not support the $+$ operator, the translation can be performed only for the roles that have the conjunct `State` $= s$ at most in one transition for each s. However, this is sufficient to verify most of the protocols that were modelled in HLPSL.

6 Implementation and Experiments

We have implemented the translation described in the previous sections and used ProVerif to verify the generated specifications.

Our experimental results are summarized in figure 7. The implementation was tested on some of the protocols modelled in HLPSL as a part of the AVISPA project [4]. For each protocol the table shows its short name (as it appears in [4]), the time it took to translate it to the applied pi calculus and verify it with ProVerif, the number of properties verified, the type of the authentication goals (weak or strong) and whether any attacks have been found. The tests have been performed on a computer with 2 Intel Xeon 3.2Ghz processors running Red Hat Enterprise Linux 3.0. Our translator was implemented in GNU C++ and we used ProVerif 1.13 compiled with Ocaml 3.08.

ProVerif has not terminated on the specification of ISO3 protocol. For all the other protocols we found known attacks against flawed protocols, and proved the correctness of the correct ones. The attacks were the same as the attacks found by the AVISPA tool (using OFMC as the back-end).

¿From figure 7 it can be concluded that the verification is very efficient for all protocols except ISO3. Non-termination of ProVerif on ISO3 is due to the fact that ProVerif is a theorem proving system that may not terminate in some cases.

7 Conclusions and Future Work

We have described an algorithm for the translation of HLPSL specifications to the dialect of the applied pi calculus supported by the ProVerif tool and presented its implementation. We illustrated the usability of the algorithm and the implementation by reasoning on a variety of HLPSL specifications.

The algorithm provides us with two interesting scientific contributions: at first it provides an independent semantics of the HLPSL specification language and second makes it possible to verify protocols specified in HLPSL with the ProVerif tool.

Our current research activity proceeds on the following directions.

First, we are extending the translator to deal with the whole HLPSL. In particular, we intend to add support for complex types, sequential composition of roles, and the translation of roles with forks in computation.

Second, we are going to formally prove our translation algorithm correct. It is not possible at the moment because both syntax and semantics of HLPSL are currently under development. We intend to provide such a proof on the

Protocol	Time (s)	Properties			Attacks found
		Secrecy	Auth.	Type of auth.	
NSPK	0.04	2	2	Weak	Yes
NSPK-Lowe	0.07	2	2	Strong	No
SHARE	0.09	1	2	Weak	Yes
EKE	0.08	1	2	Weak	Yes
Chapv2	0.08	1	2	Strong	No
ISO1	0.02		1	Strong	Yes
ISO2	0.04		1	Strong	No
ISO3	–		2	Weak	–
ISO4	0.03		2	Strong	No
UMTS-AKA	0.05	1	2	Strong	No

Fig. 7. Experimental results

basis of the formal definition of the HLPSL semantics in temporal logic of actions [18] that is currently under development.

Finally, we are working to identify more sophisticated protocols to assess the scalability of our approach and to use other verification engines for the applied pi calculus. A possible approach is to use the HAL model checker [16,15]. This approach is particularly convenient since HAL can handle fresh names generation in an effective way.

References

[1] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, 2005.

[2] L. C. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Trans. Comput. Logic*, 2(4):542–580, 2001.

[3] A. Armando and L. Compagna. SATMC: a SAT-based model checker for security protocols. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA 2004)*, volume 3229 of *LNCS*, pages 730–733. Springer-Verlag, 2004.

[4] AVISPA. Deliverable D6.2: Specification of the problems in the high-level specification language, 2005. http://www.avispa-project.org/d6-2.pdf.

[5] D. Basin, S. Modersheim, and L. Vigano. An on-the-fly model-checker for security protocol analysis. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS 2003)*, pages 253–270, 2003.

[6] B. Blanchet. Automatic verification of cryptographic protocols: a logic programming approach. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 1–3. ACM Press, 2003.

[7] Y. Boichut, P.-C. Heam, O. Kouchnarenko, and F. Oehl. Improvements on the Genet and Klay technique to automatically verify security protocols. In *Proceedings of the 3rd International Workshop on Automated Verification of Infinite States Systems (AVIS'04)*, pages 1–11, 2004.

[8] S. Brackin, C. Meadows, and J. Millen. CAPSL interface for the NRL protocol analyzer. In *Proceedings of IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET-99)*. IEEE Computer Society Press, 1999.

[9] S. H. Brackin. An interface specification language for automatically analyzing cryptographic protocols. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 40–51, 1997.

[10] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drieslma, J. Mantovani, S. Modersheim, 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 2004)*, 2004.

[11] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Trans. Softw. Eng. Methodol.*, 9(4):443–487, 2000.

[12] G. Denker, J. K. Millen, A. Grau, and J. Kuster Filipe. Optimizing protocol rewrite rules of CIL specifications. In *Proceedings of 13th IEEE Computer Security Foundations Workshop (CSFW-13)*, pages 52–62, 2000.

[13] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

[14] A. Durante, R. Focardi, and R. Gorrieri. CVS: a compiler for the analysis of cryptographic protocols. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 203–212, 1999.

[15] G. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, 2003.

[16] G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, and G. Ristori. Verifying mobile processes in the HAL environment. In *Proceedings CAV'98*, volume 1427 of *LNCS*. Springer Verlag, 1998.

[17] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In *Proceedings of 7th International Conference on Logic for Programming and Automated Reasoning (LPAR'2000)*, volume 1955 of *LNCS*, pages 131–160. Springer-Verlag, 2000.

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

[19] E. G. Lowe. Casper: A compiler for the analysis of security protocols. *J. Comput. Security*, 6:18–30, 53–84, 1998.

[20] G. Lowe. A hierarchy of authentication specifications. In *CSFW '97: Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, page 31. IEEE Computer Society, 1997.

[21] P. Syverson and C. Meadows. A formal language for cryptographic protocol requirements. *Des. Codes Cryptography*, 7(1-2):27–59, 1996.