www.avispa-project.org

IST-2001-39252

Automated Validation of Internet Security Protocols and Applications

Deliverable D2.3: The Intermediate Format

Abstract

This deliverable introduces the Intermediate Format (IF), a tool-independent protocol specification language suitable for automated deduction. Specifications of security protocols and properties written in the High-Level Protocol Specification Language (HLPSL) are automatically translated in IF specifications, which are then given as input to the different back-ends that constitute the AVISPA tool for protocol analysis.

Deliverable details

Deliverable version: *v1.0*
Person-months required: *16*
Date of delivery: *31.08.2003*
Due on: *31.08.2003*
Classification: *public*
Total pages: *28*

Project details

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

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

1 Introduction

1.1 Architecture

This deliverable introduces the Intermediate Format (IF), a tool-independent protocol specification language suitable for automated deduction. As shown in Figure 1, which displays the architecture of the AVISPA tool, the HLPSL2IF translator automatically translates a HLPSL protocol specification provided by the user into an IF specification, which is then given as input to the different back-ends of the AVISPA tool. Hence, the main goal in the design of the IF was to provide a low-level description of the protocol that is suitable for automatic analysis (rather than being abstract and easy to read for human users like the HLPSL), and yet this format should be independent from the analysis methods employed by the various back-ends.

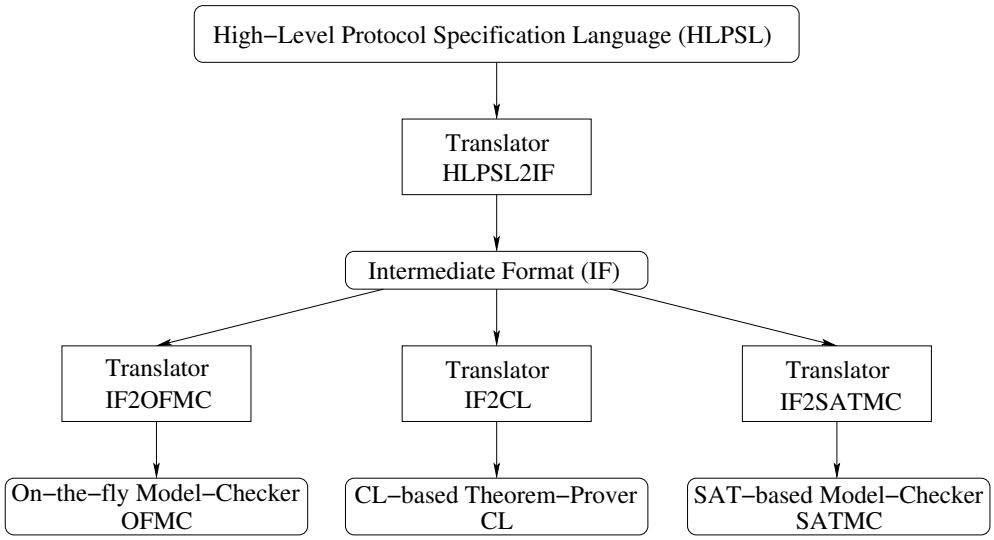

Figure 1: Architecture of the AVISPA tool

This document thus provides a basis for understanding the back-ends that work on the IF. It also provides a kind of "reference manual", i.e. the documentation for developers who plan to connect their own protocol analysis tools with the AVISPA tool by means of the IF.

1.2 Semantics

The IF describes a protocol in terms of rewrite rules describing an infinite-state transition system with an initial state, transition rules, and a state-based safety property, namely a goal (attack) predicate that defines if a

given state is an attack state or not.[1] In Section 4 below, we give a formal semantics that reflects this interpretation of the IF.

Note that the translation performed by the HLPSL2IF translator defines a semantics for the HLPSL in terms of the IF, which provides an alternative to the semantics of the HLPSL based on TLA (see Deliverable 2.1 [6]).

1.3 Novel Features

This document provides a complete description of the Intermediate Format IF. We defined a preliminary version of the IF as part of the AVISS project ("AVISS: Automated Verification of Infinite State Systems", FET-Open Project IST-2000-26410 [2, 7]). Since then, in order to be able to analyze the Internet security protocols and applications that we will consider in the AVISPA project, we have completely redesigned the language, as we illustrate below.

The most important new concept is the extension of the left-hand side of rules with conditions and negative facts, in order to allow for the explicit modeling of a wider class of protocols and properties in a natural way.

Note that the back-ends OFMC and CL [9, 10, 11, 15, 16, 17] are both based on the *lazy intruder*, a technique to symbolically represent the intruder-generated messages, which itself is based on unification [21, 12, 1, 22, 20, 19]. The use of negation in the context of unification is usually highly problematic; however, in Section 4.5, we show that this is not the case for our approach, as we have successfully managed to extend the lazy intruder to the extended language we defined. It is also worth pointing out that for the SATMC back-end [3, 4, 5], which is based on a declarative encoding into propositional logic, the introduction of negation does not require any change.

The introduction of negation also requires us to extend the intruder model. In the previous version of the IF, as is standard in protocol analysis, the intruder did not have the ability to generate fresh data himself. Since without negation the honest agents could not check that messages are different (i.e. they perform only equality checks), the intruder could always send the same messages again (as is done in approaches based on *data independence*, such as [24]). With inequalities allowed in rules, we must in general allow the intruder to generate an unbounded number of new messages. It turns out that this can be immediately handled by the back-ends based on the lazy intruder, without any additional cost (as shown below); SATMC, which is not based on the lazy intruder, can also easily handle, and profit from, the integration of such a rule by introducing additional constant

[1] An *attack trace* is a path that leads from the initial state to an attack state.

symbols (further details will be given in later deliverables, describing the different approaches and back-ends.)

We expect that, in order to analyze more complex protocols than the ones that we are currently considering according to the work-plan, we will need to further extend the IF. However, we also expect that most of these extensions will be straightforward. For example, integrating a new cryptographic operator requires only introducing a new symbol, new intruder rules, and new equations on this operator, while the rest of the IF syntax and semantics remains unchanged.

To make the IF flexible for such extensions, we use a *prelude* file that defines all protocol-independent aspects of the model, such as algebraic equations. The prelude file is shared between the HLPSL and the IF. The analysis tools that work on the IF may be specialized to the prelude file, in the sense that their method is based on this particular intruder model, and that changes to the prelude file would also require changes to the tools.

1.4 The HLPSL2IF Translator

The HLPSL2IF translator automatically translates a specification in HLPSL into an IF specification. The translator works as follows. First, it parses the HLPSL specification, checking that a number of conditions are met (e.g. that all used variables are declared). Then, it flattens the hierarchical structure of the role descriptions in HLPSL and translates them into step rules of the IF, describing the transitions honest agents can perform. (The intruder behavior is protocol-independent and specified as part of the prelude file.) The initial state of the IF is computed from the instantiation given in the HLPSL file (declaring which agents are to play which roles of the protocol with whom). Finally, the goals are computed as a state-based encoding of the properties given in the HLPSL file. A more detailed description of the translation process will be given as part of a later deliverable describing the AVISPA tool.

1.5 Organization

We proceed as follows. In Section 2, we give the syntax of the IF in BNF and explain the main features. In Section 3, we describe the prelude file. In Section 4, we describe the semantics of the IF as a transition system and a set of goal states; we also discuss the different kinds of infinity inherent in this format and the relation between negation and the lazy intruder. In Section 5, we illustrate the use of the IF, and in particular its novel features, by considering the example of the well-known Needham-Schroeder Public

Key Protocol (NSPK, [18, 23]), as well as variants of the protocol that could not be described in the previous versions of the IF. These examples are the IF translations of the respective HLPSL examples of Deliverable 2.1 [6].

2 The Syntax of the IF

We first give the entire BNF (with the usual conventions), and then give explanations and examples. The symbol for comments is %. The grammar has two start symbols, `Prelude` and `IFFile`, since the prelude file essentially has the same syntax as IF, but contains a different set of sections.

```
Prelude ::=  TypeSymbolsSection
             SignatureSection
             TypesSection
             EquationsSection
             IntruderSection

IFFile  ::=  SignatureSection
             TypesSection
             InitsSection
             RulesSection
             GoalsSection

TypeSymbolsSection ::= "section typeSymbols:" TypeList
SignatureSection   ::= "section signature:" SignatureSection0
TypesSection       ::= "section types:" TypeDeclaration*
EquationsSection   ::= "section equations:" Equation*
InitsSection       ::= "section inits:" ("initial_state"
                         Identifier ":="  State )+
RulesSection       ::= "section rules:" RulesDeclaration*
GoalsSection       ::= "section goals:" GoalDeclaration*
IntruderSection    ::= "section intruder:" RulesDeclaration*

RulesDeclaration ::= "step" Identifier "(" VariableList ")"
                       ":=" CNState ExistsVar? "=>" State
State   ::= Fact ("." Fact)*
CNState ::= NState ConditionList
ConditionList ::= ("&" Condition)*
Condition ::=  "equal(" Term "," Term ")"
            |  "leq(" Term "," Term ")"
```

```
            |  "not(" Condition ")"
NState ::= NFact ("." Nfact)*
NFact ::= Fact | "not(" Fact ")"
Fact ::= IF_Fact "(" TermList ")"

ExistsVar ::= "=[exists" VariableList "]"

GoalDeclaration ::= "goal" Identifier "(" VariableList ")"
                    ":=" CNState

Equation ::= Term "=" Term
Term ::= AtomicTerm
       | ComposedTerm
AtomicTerm ::= Constant
             | Variable
ComposedTerm ::= IF_Operator "(" TermList ")"

Constant ::= [a-z] [a-zA-Z0-9_]* | [0-9]+
Variable ::= [A-Z_] [a-zA-Z0-9_]*
Identifier ::= Constant

TypeDeclaration ::= AtomicTermList ":" Type
Type ::= IF_Type
       | IF_Operator "(" TypeList ")"
       | "{" ConstantList "}"

SignatureSection0 ::= SuperTypeDeclaration*
                    | FunctionDeclaration*
                    | PredicateDeclaration*
TypeStar ::= Type
           | Type "*" TypeStar
SuperTypeDeclaration ::= IF_Type ">" IF_Type
FunctionDeclaration ::= IF_Operator ":" TypeStar "->" Type
PredicateDeclaration ::= IF_Operator ":" TypeStar "->" Type

VariableList ::= Variable ("," Variable )*
TermList ::= Term ("," Term )*
TypeList ::= Type ("," Type )*
AtomicTermList ::= AtomicTerm ("," AtomicTerm)*

IF_Fact ::= "state_" Identifier | Identifier
```

```
IF_Operator ::= Identifier
IF_Type ::= Identifier
```

The IF specification of a protocol describes an *infinite-state transition system* by an *initial state*, a *transition relation*, and a set of *goal* (i.e. *attack*) *states*. Every state is a set of facts (e.g. the fact that the intruder knows a particular message), while the transition relation is given by *conditional rewrite rules* on sets. The prelude is a fixed file that contains all declarations that are protocol-independent, while each IF-file contains only declarations for a particular protocol.

2.1 Structure of an IF File

An IF file (and, similarly, a prelude file) consists of a sequence of sections.

Section Type Symbols (`TypeSymbolsSection`). In this section, all basic (message) types (like `nonce`) are declared.

In the sections signature (`SignatureSection`) and types (`TypesSection`), the types of variables, constants, function symbols and fact symbols are declared.

Section Signature (`SignatureSection`). This section contains declarations of the used function and fact symbols, and, more specifically, their types. Also it contains subtype declarations. This section is not necessary to define the semantics of the IF (the arity of the function and fact symbols is implicit when they are used consistently), but the type information can be helpful for some back-ends. Almost all the information in this section is protocol independent (and hence part of the prelude, not of the concrete IF file), however there is one exception: the signature of a state-fact describing the state of an honest agent is protocol-dependent.

Section Types (`TypesSection`). In this section, the types for all constants and variables can be specified. This implies that throughout an IF file an identifier cannot be used with two different types (while the scope of each variable is limited to the rule it appears in). Note that one may leave unspecified the type of some or all identifiers in order to obtain an *untyped model*.

Section Equations (`EquationsSection`). In this section, algebraic properties of the function symbols are specified.

The sections inits (`InitsSection`), rules (`RulesSection`), and intruder (`IntruderSection`) describe a transition system, and section goals (`GoalsSection`) describes a goal (or attack) predicate on states.

Section Inits (`InitsSection`). In this section, we specify one or more initial states of the protocol, and thus consider several parallel runs of the protocol. Deliverable D3.3 on *session instances* will illustrate how this section can be generated automatically by the HLPSL2IF translator.

Section Rules (`RulesSection`). In this section, we specify the transition rules of the honest agents executing the protocol. Note that, with respect to previous versions of the IF, we have extended the rules with conditions and negative facts, which are discussed in Section 2.3 below.

For the declaration of rules, we also use the following syntactic sugar. We assume that the `iknows` fact (`iknows(M)` means that the intruder knows the message `M`) is persistent, in the sense that if an `iknows` fact holds in a state, then it holds in all successor states (i.e. the intruder never forgets messages). Therefore, if an `iknows` fact appears in the left-hand side (LHS) of a rule, it should also be contained in the rule's right-hand side (RHS). To simplify the rules, however, we do not write the `iknows` facts that already appeared in the LHS. In other words, in our rules `iknows` is *implicitly persistent*, and we interpret the rules as if every LHS `iknows` also appears in the RHS.

Also, a rule can be labeled with a list of existentially quantified variables. Their purpose is to introduce new constants like fresh nonces. Note that in the previous version of the IF we instead used a method of creating unique terms; this is, of course, still possible, but the new specification language does no longer prescribe a particular method to create new constants.

Section Goals (`GoalsSection`). The goals are defined in terms of predicates on states and are conceptually not different from the LHS of rules (and we will define their semantics similarly); consequently, we allow also conditions and negative facts in the goals.

Section Intruder (`IntruderSection`). The rules in this section describe the abilities of the intruder, namely composition and decomposition of messages he knows (according to the standard Dolev-Yao intruder extended with the ability to exploit the specified algebraic properties of operators). As these abilities are again independent of the protocol, they are included in the prelude.[2]

[2]In the future, we will also allow for the specification of different intruder models,

2.2 Context-sensitive Properties

All used identifiers must be different from the IF keywords (`step`, `section`, `intruder`, `equal`, `leq`, `not`, `state`). The identifiers for types (`IF_Type`) used in declarations can only be those identifiers that have been introduced as type identifiers in the prelude. Identifiers for operators (`IF_Operator`) are only those that have been declared in the signature section of the prelude as having range type message. Similarly, fact symbols (`IF_Fact`) are only the ones declared in the signature section of the prelude or the IF file as having range type fact. The identifiers that name initial states, rules, or goals must be unique and distinct from all constants and variables and declared identifiers.

For a rule declaration, the variables in the variable list must contain exactly those variables that occur in the LHS of the rule and in the existential quantification. The variables of the RHS must be a subset of the variables in the positive facts of the LHS (excluding those variables that occur only in the conditions or the negative facts of the rule) and the existentially quantified variables. Analogous restrictions apply for initial states. More precisely, variables cannot occur in an initial state as it can be seen as the RHS of a rewrite rule that can be applied only once with an empty LHS.

2.3 Negation

We have extended the standard rewriting approach to consider rules that contain also negative facts and conditions. The conditions are conjunctions of equalities and inequalities on terms. Note that the truth value of a condition depends only on the substitution for the variables of the contained terms, while the truth value for positive and negative facts depends also on the current state (hence the conceptual distinction between facts and conditions).

The conditions can also contain comparisons between natural numbers, which are necessary for instance for protocols where agents loop according to a counter, but again their truth value depends only on the substitution for the variables of the contained terms and not on the current state.

2.4 Shared Variables and Sets

The analysis of security protocols and properties often requires us to model that an agent can remember a set of objects shared over all protocol sessions he participates in, e.g. the set of known public-key certificates of other agents,

e.g. for *Over-the-Air* protocols where the intruder is not able to intercept messages but can only insert and eavesdrop messages.

or the set of nonces the agent has seen. This can be readily modelled in the IF. For example, if `keyset_a` denotes the set of key certificates known to agent `a`, then the situation in which `keyset_a` contains a message `pair(b,kb)` can be represented by a state containing the fact `contains(keyset_a,pair(b,kb))`. This allows us to easily check if a certain element is or is not contained in a set, as well as to add and to remove elements from a set. In Section 5.3, we describe how such a set can be used to give a precise model of the Needham-Schroeder public-key protocol with key-server, where the agents ask the key-server for a key iff they do not know it yet.

2.5 Algebraic Equations

The introduction of algebraic equations is a subtle task as they affect both the unification algorithms and the abilities of the intruder to generate and analyze messages. Currently, we focus on the following operators and equations:

- Pairing is associative.
- Exponentiation (used for instance in Diffie-Hellman and RSA) commutes in the exponents, and inverse exponents cancel each other out.
- Bitwise XOR has the property of being associative, commutative, and self-inverse.

We will introduce further operators and/or equations if required by the protocols to be analyzed.

3 Prelude File

Here we give the entire prelude that we are currently using:

```
% PRELUDE AVISPA IF

section typeSymbols:

  agent, nonce, symmetric_key, public_key, function, set, table,
  nat, message, fact

section signature:

  message       > agent
```

```
  message       > nonce
  message       > symmetric_key
  message       > public_key
  message       > function
  message       > set
  message       > table

  pair     : message * message -> message
  crypt    : message * message -> message
  inv      : message -> message
  scrypt   : message * message -> message
  exp      : message * message -> message
  xor      : message * message -> message
  apply    : message * message -> message

  iknows   : message -> fact
  contains : message * message -> fact
  witness  : agent * agent * message * message -> fact
  request  : agent * agent * message * message -> fact
  secret   : message * message -> fact

section types:

  K,M,M1,M2,M3 : message

section equations:

  pair(M1,pair(M2,M3)) = pair(pair(M1,M2),M3)

  inv(inv(M)) = M

  exp(exp(M1,M2),M3) = exp(exp(M1,M3),M2)
  exp(exp(M1,M2),inv(M2)) = M1

  xor(M1,xor(M2,M3)) = xor(xor(M1,M2),M3)
  xor(M1,M2) = xor(M2,M1)
  xor(xor(M1,M1),M2) = M2


section intruder:
```

```
% generate rules

step gen_pair (M1,M2) :=
  iknows(M1).iknows(M2) => iknows(pair(M1,M2))
step gen_crypt (M1,M2) :=
  iknows(M1).iknows(M2) => iknows(crypt(M1,M2))
step gen_scrypt (M1,M2) :=
  iknows(M1).iknows(M2) => iknows(scrypt(M1,M2))
step gen_exp (M1,M2) :=
  iknows(M1).iknows(M2) => iknows(exp(M1,M2))
step gen_xor (M1,M2) :=
  iknows(M1).iknows(M2) => iknows(xor(M1,M2))
step gen_apply (M1,M2) :=
  iknows(M1).iknows(M2) => iknows(apply(M1,M2))

% analysis rules

step ana_pair (M1,M2) :=
  iknows(pair(M1,M2)) => iknows(M1).iknows(M2)
step ana_crypt (M1,M2) :=
  iknows(crypt(K,M)).iknows(inv(K)) => iknows(M)
step ana_scrypt (M1,M2) :=
  iknows(scrypt(K,M)).iknows(K) => iknows(M)

% Generating new constants of any type:

step generate (M) :=
  =[exists M]=> iknows(M)
```

4 The Semantics of the IF

In this section, we formally describe the semantics of the IF. Recall that, as we remarked above, the translation performed by the HLPSL2IF translator defines a semantics for the HLPSL in terms of the IF, which provides an alternative to the semantics of HLPSL based on TLA; see Deliverable 2.1 [6].

The basis of the semantics are terms, which are built from the constants and function symbols of the prelude and the IF files. To smoothly integrate the existential quantifier, we assume a set of fresh constants that is disjoint from all constants in the prelude and if file. For these constants, we assume a function *fresh* that maps a state and a set of variables to a substitution that

replaces the variables with constants that do not appear in the given state.

4.1 Types

Let *type* be a partial function from `AtomicTerm` to `Type` (see the grammar in Section 2) that yields for every constant and variable the respective type that has been declared, if any.

Note that our syntax allows also *composed* types, e.g.

$$\texttt{M} : \texttt{scrypt}(\texttt{symmetric\_key}, \texttt{pair}(\texttt{nonce}, \texttt{agent}))\ .$$

Such a variable declaration is used when the receiver is not supposed to analyze a certain message-part according to the protocol. For instance, in the case of the Otway-Rees protocol, A should send to B a message M that is encrypted with a key K_{AS} that is shared between A and a trusted server S. B has to forward this message M to S and cannot read it himself. Hence an intruder, impersonating A, can send any message in the place of M since B will not try to analyze it. For a *typed* model, however, we want B to accept M only if it is of the proper format (according to the protocol), i.e. if it is an encryption with a symmetric key and the contents after decryption are also of the proper format. In other words, even though B cannot decrypt the message, we assume that he can check whether the received message is of the correct type and reject it if not.

Semantically, let *op* be an n-ary `IF_Operator`, `M` a variable and $t_1, \ldots, t_n$ types (atomic or themselves composed). Then the declaration

$$\texttt{M} : op(t_1, ..., t_n)$$

is equivalent to the declarations

$$\texttt{M}_1 : t_1, \ldots, \texttt{M}_\texttt{n} : t_n$$

if $\texttt{M}_i$ (with $i = 1, \ldots, n$) are fresh variables (that do not appear in the IF file) and every occurrence of `M` in the IF file is replaced with the term $op(\texttt{M}_1, \ldots, \texttt{M}_\texttt{n})$.

One may hence see composed types as syntactic sugar, but they allow us to write the rules for an IF file independent of the question of typing.

4.2 Unification

We define unification on IF terms in the standard way, only that types and algebraic properties have to be respected.[3] Formally, the algebraic equations

[3]Unless the lazy intruder or other symbolic techniques are employed, all states are ground terms and hence we employ a special case of unification, *matching*, i.e. unification

induce an equivalence relation $\equiv$ on the term algebra, and a unifier of two terms is a substitution, such that (i) the substituted terms are equivalent with respect to $\equiv$, and (ii) the type of every substituted variable agrees with the type of the term it is replaced with. (In an untyped model, the types are not declared and hence do not constrain the unification.) As we adopt the standard notion of sorted unification, we will not go into further details here but refer the reader to [8].

Besides for the properties we have already discussed in Section 2.5, we also use the "." as an associative, commutative, and idempotent operator, i.e. we have:

$$t_1.(t_2.t_3) = (t_1.t_2).t_3$$
$$t_1.t_2 = t_2.t_1$$
$$t.t = t$$

Note, however, that these three properties cannot be specified as part of the equation section, as these properties work on facts rather than on messages. With these properties, the operator "." works as a set constructor for facts, and in the following we will consequently talk about sets, union, and set difference for facts as a shorthand.

4.3 Rule Application

For a substitution σ, we define $\sigma \models \mathit{Cond}$ on conditions as expected:

$\sigma \models t_1 = t_2$ iff $t_1\sigma = t_2\sigma$ (where t_1 and t_2 are arbitrary terms)
$\sigma \models t_1 \leq t_2$ iff $t_1\sigma \leq t_2\sigma$ (where t_1 and t_2 are natural numbers)
$\sigma \models \phi \wedge \psi$ iff $\sigma \models \phi$ and $\sigma \models \psi$ (where ϕ and ψ are conditions)
$\sigma \models \neg\phi$ iff not $\sigma \models \phi$ (where ϕ is a condition)

We define when a rule is applicable to a (ground) state by the function *matches* that takes as argument the LHS of the rule lhs and yields a function that maps a state s to the set of substitutions σ such that $lhs\sigma$ can be applied to s (this set is empty if the rule is not applicable). Note that the RHS of the rule can only contain variables from the positive facts of the LHS and the existentially quantified variables (which will be replaced by fresh constants below), therefore all substitutions that result from *matches* are ground and so are all successor states.

A LHS of a rule contains a set of positive and negative facts as well as a set of conditions, i.e. a set of equalities and inequalities as follows: the IF condition `equal`(t_1, t_2) represents equality of the terms t_1 and t_2, `not` represents negation of a condition, and $\leq (t_1, t_2)$ represents $t_1 \leq t_2$. For the

of two terms one of which is ground.

LHS *lhs* of a rule, we define the functions $PF(lhs)$ for the positive facts of the *lhs*, $NF(lhs)$ for the negative facts, $PC(lhs)$ for the positive conditions (i.e. without not), and $NC(lhs)$ for the negative conditions.[4]

$$\begin{aligned}
matches &: Rule_LHS \to (State \to 2^{Substitution}) \\
matches\ lhs\ s &= \{\sigma \mid ground(\sigma) \\
&\quad \wedge\, dom(\sigma) = vars(PF(lhs)) \cup vars(PC(lhs)) \\
&\quad \wedge\, PF(lhs)\sigma \subseteq s \wedge \sigma \models PC(lhs) \\
&\quad \wedge\, (\forall \rho.\, dom(\rho) = (vars(NF(lhs)) \cup vars(NC(lhs))) \setminus dom(\sigma) \\
&\quad \wedge\, NF(lhs)\sigma\rho \cap s = \emptyset \wedge \sigma\rho \models NC(lhs))\}
\end{aligned}$$

The intuition behind this definition is as follows: we consider every substitution σ such that under σ the positive facts can be unified with a subset of the current state (hence $PF(lhs)\sigma$ is necessarily ground) and the positive conditions are satisfied. Furthermore, for all ground substitutions ρ for the remaining variables, i.e. those variables that appear only in negative facts and in negative conditions, we postulate that none of the negative facts under $\sigma\rho$ is contained in the state and none of the conditions is satisfied for $\sigma\rho$. Note also that *matches* is applied in the same way for goal states (which are syntactically the same as a rule's LHS).

To define the semantics of a rule as a state-transition function, we use the applicability check *matches*; besides for this check, the conditions and the negative facts of the rule do not play any role: the transition itself is concerned only with the positive *facts* of the LHS of the rule, the existentially quantified variables, and the RHS.

$$\begin{aligned}
[\![\cdot]\!] &: Rule \to (State \to 2^{State}) \\
[\![(lhs, exVar, rhs)]\!](s) &= \{s' \mid \exists \sigma, \rho.\, \sigma \in matches\ lhs\ s \\
&\quad \wedge\, \rho = fresh(s, exVar) \\
&\quad \wedge\, s' = (s \setminus (PF(lhs)\sigma)) \cup (rhs\,\rho\sigma)\}
\end{aligned}$$

Note that here the semantics of a rule is defined as a state-transition function operating only on ground terms, i.e. s may not contain variables (otherwise the definition of the transition relation may not behave as one would expect); the resulting s' is then also ground, as the rules cannot introduce any new variables. However, one key feature of the IF is the support for symbolic methods, in particular the narrowing-style lazy intruder technique, which is hard to combine with any form of negation; in Section 4.5, we show

[4]Note that 2^S denotes the power-set of a set S.

that the form of IF rules we have defined here is indeed compatible with our lazy intruder approach.

The rest of the semantics is straightforward: we have one or more ground initial states and a transition relation; this defines an infinite-state transition system. A protocol, described by an IF file, is secure iff there are no reachable state s and goal g such that *matches* g s holds. We will refer to the transition system defined in this section as the *ground model*.

4.4 Kinds of Infinity in the Model

The transition system we have defined is, in general, infinite, where the following distinct kinds of infinity can occur:

- The transition system is by default untyped, so the complexity of messages that can occur is unbounded, and this implies that there are infinitely many different possible messages that can occur (and they are for instance stored in an agent state-fact).
- The number of steps that an honest agent can perform to execute a run of the protocol can be unbounded (e.g. loops that the agent can repeat an unbounded number of times).
- The number of parallel sessions that the agents can execute may be unbounded; although the initial state is a finite set of ground terms, there can be rules that create new state-facts that correspond to new sessions of agents in their initial states.
- The number of agents may be unbounded.

It is possible to obtain a finite-state transition system by bounding all of these parameters of the model. Moreover, using the lazy intruder technique allows us to decide the security question even without the first restriction on the complexity of messages [25].

4.5 Negation and the Lazy Intruder

We now illustrate how we can extend the lazy intruder approach with explicit negation (for a full account, see [9]; see also [1], which is, besides for ours, the only other lazy intruder approach that can handle a form of negation). The basic idea is to reduce the problem to what we already have (symbolic states and constraints) plus a conjunction/disjunction of inequalities on message terms. In this section, we assume for simplicity that the term algebra is free,

so that there is a most general unifier for every unification problem (i.e. all other unifiers are instances of the most general one).

The equivalent of the *matches* function of the ground model is now computed on a symbolic state s and a rule $lhs \Rightarrow rhs$ as follows:

1. Find a unifier for s and $PF(lhs)$. Without algebraic equations, there is always a most general unifier. If there is no unifier, then the rule is not applicable, else we apply the unifier to the state, the constraints and the rule.

2. For every equation $t = t' \in PC(lhs)$, try to unify t and t'. If this is not possible, then the rule is not applicable, else we apply the unifier to the state, the constraints and the rule.

3. Every negative fact $\texttt{not}(f) \in NF(lhs)$ induces a set of unifiers for f and some fact of the state. This set is empty if there is no fact that can be unified with the current state, and it contains the identity if f is directly contained in the current state.

   For every unifier, we specify the negation of the unifier as a disjunction of inequalities:

   $$\neg[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n] = \bigvee_{i=1}^{n} x_i \neq t_i \ .$$

   In the special case of the identity, we have $n = 0$ and hence the result is the neutral element of the disjunction, i.e. *false*.

4. As the rules contain only conjunctions of conditions, we now have a conjunction of disjunctions of inequalities. Note that we do not allow arbitrary boolean formulae of inequalities here.

5. The inequalities are handled in a straightforward way: every substitution that arises during the search is also applied to the inequalities. Also, we check if an inequality is a tautology or if it is unsatisfiable, and replace it with *true* or *false*, respectively. Straightforward rules reduce the boolean formula to eliminate conjunction or disjunction with *true* or *false*.

In [9], we have shown that a set of inequalities, which is reduced according to the last step of the above procedure, conjoined with a simple constraint set of the lazy intruder is always satisfiable: a variable on the LHS of a simple lazy intruder constraint set expresses the fact that in place of this variable the intruder may use any message he can generate, where the only difference

with respect to the "normal" approach is that with the lazy intruder the inequalities additionally demand that these messages are different. This reflects that in such a model with negation the intruder must be able to create an unbounded number of *different* messages. He can do this if he knows at least one message (e.g. his own name): he can create new messages for instance by concatenation of known ones.

We can see here that, with the introduction of negation, we need the intruder to be able to create new messages at any point. This feature comes for free with the lazy intruder approach: inequalities on variables, i.e. messages where it does not matter (yet) which value the intruder chooses, do not require further exploration.

The conditions in the IF rules that impose comparisons between natural numbers will be treated similarly, i.e. as additional constraints in the symbolic approach, where we check in every state that at least one solution for the constraints exists. Although the satisfiability problem is harder when allowing comparisons on natural numbers, it is still decidable [13, 14].

5 Example: the Needham-Schroeder Public-Key Protocol (NSPK)

As a concrete example, we give IF files for a number of variants of the well-known Needham-Schroeder Public Key Protocol (NSPK) [18, 23].

5.1 "Standard" NSPK

The first variant that we consider is the usual variant with the three-message exchange between Alice and Bob.

```
section signature:

  state_Alice : nat * agent * agent * public_key * public_key
                * nonce * nonce * nat -> fact
  state_Bob   : nat * agent * agent * public_key * public_key
                * nonce * nonce * nat -> fact

section types:

  A,B,a,b,i: agent
  KA,KB,ka,kb,ki: public_key
  0,1,2,3: nat
```

```
  SID: nat
  NA,NB,na,nb,ni: nonce

section inits:

  initial_state init1 :=
    iknows(i).
    % session 1 [A:a, B:b, KA:ka, KB:kb]
    state_Alice(0,a,b,ka,kb,ni,ni,1).
    state_Bob(0,b,a,kb,ka,ni,ni,2).
    iknows(a).iknows(b).iknows(ka).iknows(kb).
    % session 2 [A:a, B:i, KA:ka, KB:ki]
    state_Alice(0,a,b,ka,ki,ni,ni,3).
    iknows(ki).iknows(inv(ki))

section rules:

   step step0 (A,B,KA,KB,NA,SID) :=
    state_Alice(0,A,B,KA,KB,ni,ni,SID)
    =[exists NA]=>
    state_Alice(1,A,B,KA,KB,NA,ni,SID).
    iknows(crypt(KB,pair(NA,A))).
    secret(NA,B).
    witness(A,B,na,NA)

  step step1 (A,B,KA,KB,NA,NB,SID) :=
    state_Alice(1,A,B,KA,KB,NA,ni,SID).
    iknows(crypt(KA,pair(NA,NB)))
    =>
    state_Alice(2,A,B,KA,KB,NA,NB,SID).
    iknows(crypt(KB,NB))

  step step2 (A,B,KA,KB,NA,NB,SID) :=
    state_Bob(0,B,A,KB,KA,ni,ni,SID).
    iknows(crypt(KB,pair(NA,A)))
    =[exists NB]=>
    state_Bob(1,B,A,KB,KA,NA,NB,SID).
    iknows(crypt(KA,pair(NA,NB))).
    secret(NB,A).
    witness(B,A,nb,NB)
```

```
  step step3 (A,B,KA,KB,NA,NB,SID) :=
    state_Bob(1,B,A,KB,KA,NA,NB,SID).
    iknows(crypt(KB,NB))
    =>
    state_Bob(2,B,A,KB,KA,NA,NB,SID)

section goals:

  % weak authentication
  goal authenticate_A_B_NB (A,B,KA,KB,NA,NB,SID) :=
    state_Alice(2,A,B,KA,KB,NA,NB,SID).
    not(witness(B,A,nb,NB))
    & not(equal(B,i))

  % replay
  goal authenticate_A_B_NB_r (A,B,KA,KB,NA,NA2,NB,SID,SID2) :=
    state_Alice(2,A,B,KA,KB,NA,NB,SID).
    state_Alice(2,A,B,KA,KB,NA2,NB,SID2).
    & not(equal(B,i)) & not(equal(SID,SID2))

  % weak authentication
  goal authenticate_B_A_NA (A,B,KA,KB,NA,NB,SID) :=
    state_Bob(2,B,A,KB,KA,NA,NB,SID).
    not(witness(A,B,na,NA))
    & not(equal(A,i))

  % replay
  goal authenticate_B_A_NA_r (A,B,KA,KB,NA,NB,NB2,SID,SID2) :=
    state_Bob(2,B,A,KB,KA,NA,NB,SID).
    state_Bob(2,B,A,KB,KA,NA,NB2,SID2).
    & not(equal(A,i)) & not(equal(SID,SID2))

  goal secrecy (M,A) :=
    secret(M,A).iknows(M) & not(equal(A,i))
```

In this specification, we used two fact symbols, `state_Alice` and `state_Bob` for representing the local state of an honest agent in role Alice or Bob, respectively. The format of such `state` terms is protocol-dependent (therefore it is explicitly declared in the signature section): they allow us to represent all information relevant for participating in one session of NSPK in role Alice or Bob, namely the step number of the last executed protocol step, the

names of the involved agents, their public keys, their nonces, and a session identifier. This identifier is necessary to allow for several parallel sessions between the same agents, as it is similarly necessary in the case of HLPSL. Initially, the slots for the two nonces are filled with the dummy value `ni` (standing for "nonce of intruder"), which expresses that no nonce has been created or received yet.

The rules incorporate an optimization that we call *step-compression*, and which is employed by all the three back-ends of the AVISPA tool [3, 4, 5, 9, 10, 11, 15, 16, 17]. Step-compression is based on the idea (followed also in other approaches such as [1, 12, 19, 22]) that we can identify the intruder and the network: every message sent by an honest agent is received by the intruder and every message received by an honest agent comes from the intruder. Formally, we compose (or "compress") several steps: when the intruder sends a message, an agent reacts to it according to his rules, and the intruder diverts immediately the agent's answer. A bisimulation proof [9] shows that the model with such composed actions (which we present here) is "attack-equivalent" to the model with single (uncompressed) transitions (i.e. we end up in an attack state using composed transitions iff that was the case using uncomposed transitions).

The facts `witness` and `secret` are used to encode information necessary to keep track of the goal states (which represent a violation of the security properties): `witness(a,b,na,17)` means that the agent `a` has generated the nonce of value `17` as nonce identifier `na` for communication with `b`. The value `secret(17,a)` means that some honest agent has created the value `17` for the communication with an agent `a` (so the intruder may not find out `17`). This is specified by the secrecy goal: the intruder may not find out `M` if `M` is a secret that was generated by an honest agent for an agent `A`, unless `A` is the intruder.

The authenticate goals are much more subtle to express: for *weak* authentication, we need the fact that an agent has finished his part of the protocol, believing the received nonce really comes from the agent it appears to come from, while there is no witness fact that supports this belief. Again, the exception is that the other agent is the intruder acting under his real name. For *strong* (non-injective) authentication, we additionally need the fact that it is not a replay of the same messages, i.e. no agent may be made accept a second time the same nonce as coming from another agent, unless the other agent is the intruder.

5.2 NSPK with replay-protection

We now show how a variant of the NSPK can be expressed, where agents store all the nonces they have seen (generated or received from others) so that they will not accept any of these nonces as fresh from the other agents in future protocol runs.

The only changes are in the rules for agents generating nonces (and storing them) as well as receiving a supposed-to-be-fresh nonce (checking it is not yet stored and storing it).

```
section signature:

  state_Alice : nat * agent * agent * public_key * public_key
                * nonce * nonce * set * nat -> fact
  state_Bob   : nat * agent * agent * public_key * public_key
                * nonce * nonce * set * nat -> fact

section types:

  A,B,a,b,i: agent
  KA,KB,ka,kb,ki: public_key
  0,1,2,3: nat
  SID: nat
  Noncestore,noncestore_a,noncestore_b: set
  NA,NB,na,nb,ni: nonce

section inits:
  initial_state init1 :=
    iknows(i).
    % session 1 [A:a, B:b, KA:ka, KB:kb]
    state_Alice(0,a,b,ka,kb,ni,ni,noncestore_a,1).
    state_Bob(0,b,a,kb,ka,ni,ni,noncestore_b,2).
    iknows(a).iknows(b).iknows(ka).iknows(kb).
    % session 2 [A:a, B:i, KA:ka, KB:ki]
    state_Alice(0,a,b,ka,ki,ni,ni,noncestore_a,3).
    iknows(ki).iknows(inv(ki))

section rules:
  step step0 (A,B,KA,KB,NA,Noncestore,SID) :=
    state_Alice(0,A,B,KA,KB,ni,ni,Noncestore,SID)
    =[exists NA]=>
```

```
    state_Alice(1,A,B,KA,KB,NA,ni,Noncestore,SID).
    iknows(crypt(KB,pair(NA,A))).
    secret(NA,B).
    witness(A,B,na,NA).
    contains(NA,Noncestore)

  step step1 (A,B,KA,KB,NA,NB,Noncestore,SID) :=
    state_Alice(1,A,B,KA,KB,NA,ni,Noncestore,SID).
    iknows(crypt(KA,pair(NA,NB))).
    not(contains(NB,Noncestore))
    =>
    state_Alice(2,A,B,KA,KB,NA,NB,Noncestore,SID).
    iknows(crypt(KB,NB)).
    contains(NB,Noncestore)

  step step2 (A,B,KA,KB,NA,NB,Noncestore,SID) :=
    state_Bob(0,B,A,KB,KA,ni,ni,Noncestore,SID).
    iknows(crypt(KB,pair(NA,A))).
    not(contains(NA,Noncestore))
    =[exists NB]=>
    state_Bob(1,B,A,KB,KA,NA,NB,Noncestore,SID).
    iknows(crypt(KA,pair(NA,NB))).
    secret(NB,A).
    witness(B,A,nb,NB).
    contains(NA,Noncestore).
    contains(NB,Noncestore)
```

5.3 NSPK with key-server

We model the server as a "persistent" agent, whose `state`-fact never changes and that uniformly reacts to every incoming key-request with the appropriate certificate if there is such a certificate in his database.

The key-table the server possesses is modeled by a finite set of pairs of agents and their public keys. Alternatively, we could use a function to this end, which would simplify matters when the number of agents is not bounded.

```
section signature:

  state_Alice : nat * agent * agent * public_key * public_key
                * nonce * nonce * set * nat -> fact
  state_Bob   : nat * agent * agent * public_key * public_key
```

```
                * nonce * nonce * set * nat -> fact

section types:

  A,B,a,b,i: agent
  KA,KB,KS,ka,kb,ki,ks: public_key
  0,1,2,3: nat
  SID: nat
  Keyset,keyset_a,keyset_b,keyset_s: set
  NA,NB,na,nb,ni: nonce

section inits:

  initial_state init1 :=
    iknows(i).
    % session 1 [A:a, B:b]
    state_Alice(0,a,b,ka,ks,ni,ni,keyset_a,1).
    state_Bob(0,b,a,kb,ks,ni,ni,keyset_b,2).
    iknows(a).iknows(b).
    % session 2 [A:a, B:i]
    state_Alice(0,a,b,ka,ks,ni,ni,keyset_a,3).
    iknows(ki).iknows(inv(ki)).
    % whatever the agents happen to know initially
    % depending on instance, let's say
    contains(pair(i,ki),keyset_a).
    % initial server keyset={(a,ka),(b,kb),(i,ki)}
    state_Server(s,keyset_s).
    contains(pair(a,ka),keyset_s).
    contains(pair(b,kb),keyset_s).
    contains(pair(i,ki),keyset_s)

section rules:

  step step0_0 (A,B,KA,KB,KS,Keyset,SID) :=
    state_Alice(0,A,B,KA,KS,ni,ni,Keyset,SID).
    not(contains(pair(B,KB),Keyset))
    =>
    state_Alice(1,A,B,KA,KS,ni,ni,Keyset,SID).
    iknows(pair(A,B))

  step step0_1 (A,B,KA,KB,KS,NA,Keyset,SID) :=
```

```
    state_Alice(0,A,B,KA,KS,ni,ni,Keyset,SID).
    contains(pair(B,KB),Keyset)
    =[exists NA]=>
    state_Alice(2,A,B,KA,KS,NA,ni,Keyset,SID).
    contains(pair(B,KB),Keyset)
    iknows(crypt(KB,pair(NA,A))).
    secret(NA,B).
    witness(A,B,na,NA)

  step step1 (A,B,KA,KB,KS,NA,Keyset,SID) :=
    state_Alice(1,A,B,KA,KS,ni,ni,Keyset,SID).
    iknows(crypt(inv(KS),pair(B,KB)))
    =[exists NA]=>
    contains(pair(B,KB),Keyset).
    state_Alice(2,A,B,KA,KS,NA,ni,Keyset,SID).
    iknows(crypt(KB,pair(NA,A))).
    secret(NA,B).
    witness(A,B,na,NA)

  step step2 (A,B,KA,KB,KS,NA,NB,Keyset,SID) :=
    state_Alice(2,A,B,KA,KS,NA,ni,Keyset,SID).
    iknows(crypt(KA,pair(NA,NB))).
    contains(pair(B,KB),Keyset)
    =>
    state_Alice(3,A,B,KA,KS,NA,NB,Keyset,SID).
    iknows(crypt(KB,NB)).
    contains(pair(B,KB),Keyset)

  % <similar rules for role bob>

  step server (A,B,KB) :=
    iknows(pair(A,B)).
    state_Server(s,keyset_s).
    contains(pair(B,KB),keyset_s)
    =>
    state_Server(s,keyset_s).
    iknows(crypt(inv(ks),pair(B,KB)))
```

6 Conclusion

The IF is a low-level, simple but expressive language for specifying security protocols and their properties. IF specifications can be generated automatically by the HLPSL2IF translator from specifications written in the high-level language HLPSL [6]. The first version of IF was developed during the AVISS project and this new version is a complete re-design of the language, based on the experience the project partners have made with a variety of protocol analysis problems, in order to be able to analyze the Internet security protocols and applications that we will consider in the AVISPA project,

References

[1] R. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In C. Palamidessi, editor, *Proceedings of Concur'00*, LNCS 1877, pages 380–394. Springer-Verlag, 2002.

[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 *Proc. CAV'02*, LNCS 2404. Springer, 2002.

[3] A. Armando and L. Compagna. Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In *Proceedings of FORTE 2002*, LNCS 2529, pages 210–225. Springer-Verlag, 2002.

[4] A. Armando and L. Compagna. Abstraction-driven SAT-based Analysis of Security Protocols. In *Proceedings of SAT 2003*, LNCS 2919. Springer-Verlag, 2003. Available at `www.avispa-project.org`.

[5] A. Armando, L. Compagna, and P. Ganty. SAT-based Model-Checking of Security Protocols using Planning Graph Analysis. In *Proceedings of FME'2003*, LNCS 2805. Springer-Verlag, 2003.

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

[7] AVISS. Deliverable 1.3: Final project report. For more information on the AVISS project see `http://www.avispa-project.org/theproject.html`, 2002.

[8] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[9] D. Basin, S. Mödersheim, and L. Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. In E. Snekkenes and D. Gollmann, editors, *Proceedings of ESORICS'03*, LNCS 2808, pages 253–270. Springer-Verlag, 2003. Available at `http://www.avispa-project.org`.

[10] D. Basin, S. Mödersheim, and L. Viganò. Constraint Differentiation: A New Reduction Technique for Constraint-Based Analysis of Security Protocols. In V. Atluri and P. Liu, editors, *Proceedings of CCS'03*, pages 335–344. ACM Press, 2003. Available at `http://www.avispa-project.org`.

[11] D. Basin, S. Mödersheim, and L. Viganò. Constraint Differentiation: A New Reduction Technique for Constraint-Based Analysis of Security Protocols (Extended Abstract). In *Proceedings of SPV'03*. Available at `www.loria.fr/~rusi/spv.html`, 2003. Available at `http://www.avispa-project.org`.

[12] M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings of ICALP'01*, LNCS 2076, pages 667–681. Springer-Verlag, 2001.

[13] J. Büchi. Weak second-order arithmetic and finite automata. *Zentralblatt der mathematischen Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[14] J. Büchi. On a decision method in restricted second order arithmetic. In *Proc. 1960 Int. Congr. on Logic, Methodology and Philosophy*, pages 1–11. Stanford University Press, 1962.

[15] Y. Chevalier and L. Vigneron. A Tool for Lazy Verification of Security Protocols. In *Proceedings of ASE'01*. IEEE Computer Society Press, 2001.

[16] Y. Chevalier and L. Vigneron. Towards Efficient Automated Verification of Security Protocols. In *Proceedings of the Verification Workshop (VERIFY'01) (in connection with IJCAR'01), Università degli studi di Siena, TR DII 08/01*, pages 19–33, 2001.

[17] Y. Chevalier and L. Vigneron. Automated Unbounded Verification of Security Protocols. In E. Brinksma and K. G. Larsen, editors, *Proceedings of CAV'02*, LNCS 2404, pages 324–337. Springer-Verlag, 2002.

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

[19] R. Corin and S. Etalle. An Improved Constraint-Based System for the Verification of Security Protocols. In *Proceedings of SAS 2002*, LNCS 2477, pages 326–341. Springer-Verlag, 2002.

[20] M. Fiore and M. Abadi. Computing Symbolic Models for Verifying Cryptographic Protocols. In *Proceedings of CSFW'01*. IEEE Computer Society Press, 2001.

[21] A. Huima. Efficient infinite-state analysis of security protocols. In *Proceedings of the FLOC'99 Workshop on Formal Methods and Security Protocols (FMSP'99)*, 1999.

[22] J. K. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proceedings of the ACM Conference on Computer and Communications Security CCS'01*, pages 166–175, 2001.

[23] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. Technical Report CSL-78-4, Xerox Palo Alto Research Center, Palo Alto, CA, USA, 1978. Reprinted June 1982.

[24] A. Roscoe and P. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7:147–190, 1999.

[25] M. Rusinowitch and M. Turuani. Protocol Insecurity with Finite Number of Sessions is NP-complete. In *Proceedings of CSFW'01*. IEEE Computer Society Press, 2001. Available at `http://www.avispa-project.org`.