AVISS

Automated Verification of Infinite State Systems

FET-Open Assessment Project IST-2000-26410

(May 1, 2001 — April 30, 2002)

Deliverable D1.3: Final project report

(DRAFT)

March 26, 2002

Contents

1. Introduction

This document is a draft of the final report of the AVISS project (IST-2000-26410). It describes all the results obtained in the project, bringing up-to-date and extending the previous deliverables. A final version of this report will be available after the end of the project (after April 30, 2002).

1.1 Background and motivation

Experience over the last twenty years has shown that, even assuming perfect cryptography, the design of protocols for secure electronic transactions and electronic commerce is highly error-prone and that conventional validation techniques based on informal arguments and/or testing are not up to the task [6,18,26,46]. It is now widely recognized that only verification tools based on formal methods can provide the level of assurance required, and a number of formal methods have been proposed for rigorously analyzing security protocols, e.g. [5, 7, 8, 10–12, 14, 17, 19, 20, 22, 27, 28, 30, 32, 34–36, 39, 43]. The current generation of formal method tools are based on either interactive verification or automatic finite-state model-checking. The interactive tools require a considerable investment of time by expert users and provide no support for error detection when the protocols are flawed. The automatic tools generally require strong assumptions that bound the information analyzed, so that an infinite-state system can be approximated by a finite-state one. Moreover, the current level of automated support scales poorly and is insufficient for the validation of realistic protocols. Finally, tuning the specification (such as building relevant approximations) and tuning the tool's parameters often requires significant expertise too.

While exploratory work has shown the potential of automated deduction techniques in this setting, most of them have never been tested comprehensively across a large range of problems. A FET OPEN assessment project provided thus a necessary preliminary step before starting a wide-scale industrial involvement within an RTD project.

1.2 Project objectives and results

The AVISS project lays the foundations of a new generation of verification tools for automatic error detection for e-commerce and related security protocols; this paves the way to turning the prototype we have developed into a mature technology, whose application in the industrial setting will be ascertained in a follow-up, full RTD project with industry involvement.

1.2.1 Project phases.

To assess the potential of this technology, we have structured our project into two phases:

1. A *development phase* aimed at the design and implementation of a prototype verification tool incorporating inference engines based on three promising automated deduction techniques: on-the-fly model-checking based on lazy data-types, theorem-proving with constraints, and model-checking based on propositional satisfiability checking.

2. An *analysis phase* aimed at measuring the success of the assessment project by thoroughly testing and evaluating our prototype tool (and the techniques) by applying it to the protocols in the Clark/Jacob library [18], which contains 51 protocol verification problems.

1.2.2 The prototype AVISS tool.

The prototype AVISS tool for security protocol analysis has the architecture shown in Figure 1.1. The *High-Level Protocol Specification Language* HLPSL is a language close to that used in textbooks and by engineers. Given a specification of a protocol verification problem in the HLPSL (i.e. a description of a protocol together with a security property to check), the HLPSL2IF translator translates this specification into the more detailed, tool-independent format suitable for automated deduction, called the *Intermediate Format* (IF). Specifications in the IF are then translated into tool-specific encodings that are fed into the inference engines that implement the selected automated deduction techniques. The three back-ends of the tool that we have developed in the context of the AVISS project are:

- The on-the-fly model-checker OFMC developed by the Freiburg group.
- The theorem prover based on constraint logic CL developed by the Nancy group.
- The model-checker based on propositional satisfiability checking SATMC developed by the Genova group.

Whenever the input protocol is flawed and when the analysis carried out by the tool completes successfully, the tool will return as a counter-example an execution trace witnessing an attack on the protocol.

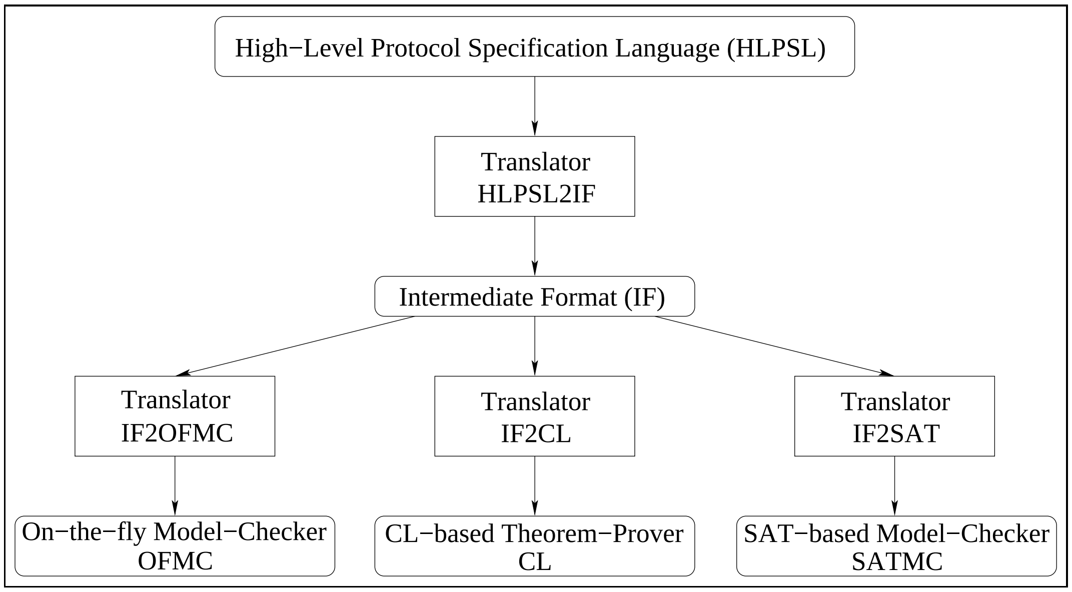

High−Level Protocol Specification Language (HLPSL)
Translator HLPSL2IF
Intermediate Format (IF)
Translator IF2OFMC
Translator IF2CL
Translator IF2SAT
On−the−fly Model−Checker OFMC
CL−based Theorem−Prover CL
SAT−based Model−Checker SATMC

Figure 1.1: Architecture of the prototype AVISS tool

Ease of tool integration was an important design consideration for the AVISS tool. We currently have implemented three back-ends for performing complementary automated protocol analysis techniques, and the IF provides a specification language that allows for the integration of other existing tools as further back-ends into the AVISS architecture. As we discuss in Section 8, we have begun to investigate in this direction and the first results are very promising; further investigations will be subject of future work.

Success criteria		Objectives	Results
Coverage	(number of protocols specified)	80%	90%
Effectiveness	(number of flaws detected)	70%	91 %
Performance	(CPU time)	< 1 hour on 70%	< 1 min on 91%

Table 1.1: Summary of the project results (according to the success criteria)

1.2.3 Project results.

The results we describe here demonstrate the success of our assessment project: our prototype tool more than satisfies the project objectives, according to the success criteria adopted as a measure of the assessment, i.e.

Coverage: the number of protocols, security properties, and types of security threats that can be specified in the tool's input specification language.

Effectiveness: the number of insecure protocols detected as flawed and the number of different types of attacks on the insecure protocols detected.

Performance: the time spent by the tool to detect errors in the protocols.

The AVISS tool achieves the following results, which we display in a summarized form in Table 1.1.

Coverage: 46 of 51 of the protocols in the Clark/Jacob library [18] are specifiable in the tool's input specification language HLPSL, which amounts to a coverage of 90% (80% was requested).

Effectiveness: the tool detects attacks in 91% of the protocols in the library known to be insecure (70% was requested). More specifically, 35 out of the 51 protocols are flawed, and we can find a flaw in 32 of them, including a previously unknown flaw (the remaining 3 protocols can not be specified in HLPSL yet, as explained in Section 7).

Performance: the tool can find the attack for 91% of the flawed protocols in the Clark/Jacob library (i.e. the 32 flawed protocols that can be modeled), in less than one minute of CPU time on a 1.4 GHz processor (the criterion required that at least 70% of the (flawed) protocols were processed in less than 1 hour of CPU time).

1.2.4 Project web-page, deliverables and publications.

Detailed information about the project is available at the AVISS web-page:

`www.informatik.uni-freiburg.de/~softech/research/projects/aviss`

The web-page contains pointers to the previous project deliverables, to the joint system description, and to the papers the partners have written documenting the project results:

- **The AVISS Security Protocol Analysis Tool**.
  Alessandro Armando, David Basin, Mehdi Bouallagui, Yannick Chevalier, Luca Compagna, Sebastian Mödersheim, Michaël Rusinowitch, Mathieu Turuani, Luca Viganò, Laurent Vigneron. January 2002. (System description submitted to CAV'02.)

- **Automatic SAT-Compilation of Security Problems**.
  Alessandro Armando and Luca Compagna. January 2002.

- **Automated Unbounded Verification of Security Protocols.**
  Yannick Chevalier and Laurent Vigneron.
  Technical Report 4369, INRIA (France), January 2002. (Submitted to CAV'02.)

- **A Tool for Lazy Verification of Security Protocols.**
  Yannick Chevalier and Laurent Vigneron.
  In Proceedings of ASE-2001: The 16th IEEE Conference on Automated Software Engineering, San Diego (CA), November 2001. IEEE CS Press. Long version available as Technical Report A01-R-140, LORIA, Nancy (France).

- **Towards Efficient Automated Verification of Security Protocols.**
  Yannick Chevalier and Laurent Vigneron.
  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, Siena (Italy), June 2001. Also available as Technical Report A01-R-046, LORIA, Nancy (France).

- **Protocol Insecurity with Finite Number of Sessions is NP-complete.**
  Michaël Rusinowitch and Mathieu Turuani.
  In 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia (Canada), June 2001. Also available as Technical Report A01-R-051, LORIA, Nancy (France).

A paper on the on-the-fly model-checker is currently in preparation.

1.2.5 The graphical web-based interface

Available from the project web-page is also the graphical web-based interface that we have designed for the AVISS verification tool. This graphical web-based interface has a number of useful features. It ensures a permanent access to the latest version of the tool. Moreover, installation and tool configuration are not required for its use. Therefore, the tool is accessible to a large number of users whose feedback will enable us to further enhance the system.

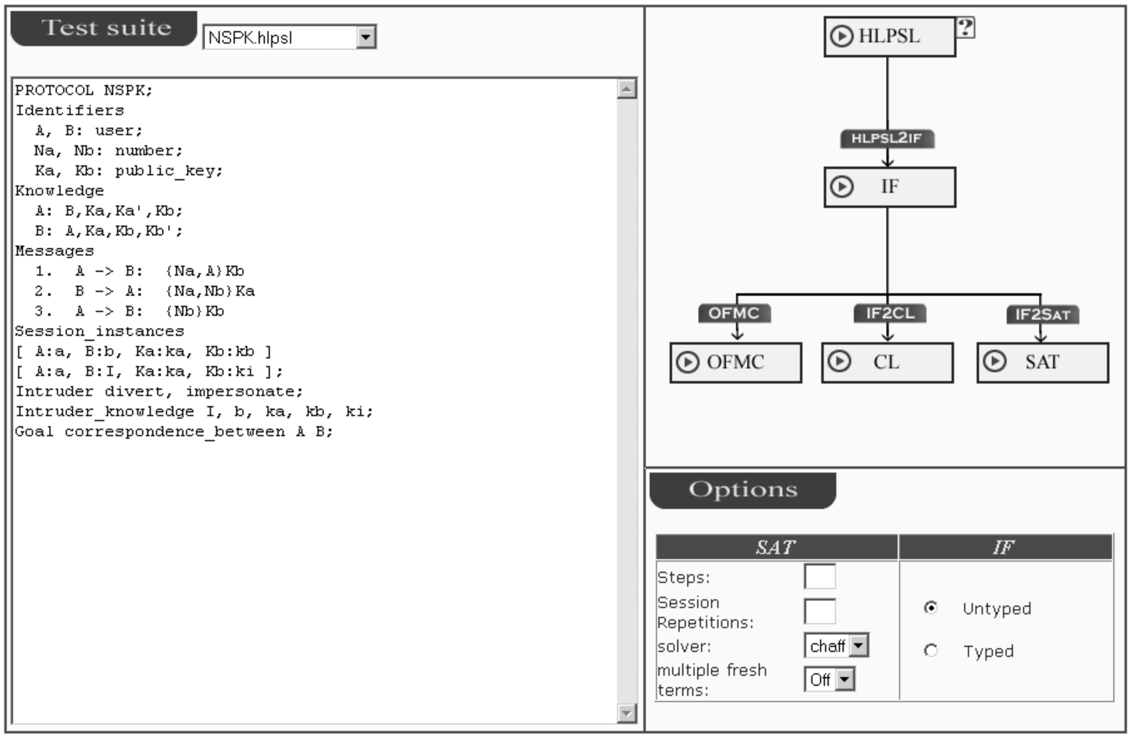
Figure 1.2 shows a snapshot of the interface, which, as displayed on the right, imports the tree structure of the project architecture. In particular, there are four main icons corresponding to the HLPSL2IF translator and the three implemented back-ends (OFMC, CL and SATMC). As shown on the left, input protocols are either selected from a given test-suite, including a number of protocols in the library [18], or they are defined by the user. The interface also provides user guidelines.

When the user, after having selected a particular protocol, selects a tree node by clicking on one of the icons, a form is submitted to the web-server. (The user can also select a number of options for the IF and for the SATMC back-end; these options are explained in the corresponding sections.) This form contains data about the input protocol, the selected node and the execution arguments. This data is processed by a server side-script that dynamically builds shell commands: the tree branch starting from the HLPSL node to the selected node is equivalent to a sequence of shell commands, and this sequence is executed by the web-server and its output is displayed on the browser. More precisely, clicking on the IF icon yields the corresponding HLPSL2IF output, and the tree leaves yield the output of the back-ends.

This interface integrates the different back-ends in a seamless way, and therefore increases the usability of the AVISS tool. It thus provides an effective prototype for a future advanced platform.

1.3 Organization of this report

The structure of this final report reflects the architecture of our tool. We begin by describing the high-level protocol specification language HLPSL in Section 2, and then consider the Intermediate Format IF in Section 3. Section 4 describes two important optimizations based on the IF format: a more efficient model of the intruder's behavior, called the lazy-intruder, and the step-compression method, which reduces the search space by excluding certain protocol interleavings that need not



```
PROTOCOL NSPK;
Identifiers
  A, B: user;
  Na, Nb: number;
  Ka, Kb: public_key;
Knowledge
  A: B,Ka,Ka',Kb;
  B: A,Ka,Kb,Kb';
Messages
  1.  A -> B:  {Na,A}Kb
  2.  B -> A:  {Na,Nb}Ka
  3.  A -> B:  {Nb}Kb
Session_instances
[ A:a, B:b, Ka:ka, Kb:kb ]
[ A:a, B:I, Ka:ka, Kb:ki ];
Intruder divert, impersonate;
Intruder_knowledge I, b, ka, kb, ki;
Goal correspondence_between A B;
```

SAT		*IF*
Steps:		[✓ Untyped]
Session Repetitions:		Typed
solver:	chaff	
multiple fresh terms:	Off	

Figure 1.2: Snapshot of the graphical web-based interface

be considered. These improvements are shared by the back-ends and are hence first explained in general, tool-independent form. Section 5 describes the common output format adopted by our back-ends. The details of the three deduction techniques and the back-ends implementing them are given in Section 6. Section 7 presents the results of our experiments with the library of protocols, and demonstrates that the project achieved its objectives. We discuss relevant related work in Section 8. Finally, in Section 9, we draw conclusions and briefly discuss the full RTD project with industry involvement that we envision as a natural follow-up to this assessment project.

2. The High-Level Protocol Specification Language HLPSL

2.1 Introduction

In this section, we define the syntax of the High Level Protocol Specification Language (HLPSL) that we use to study protocols. A *protocol* is defined by a finite set of roles, and by messages these roles exchange. A *role* is defined by the messages this role sends and expects to receive during a protocol execution and by the knowledge needed to do so.

The protocol is studied by using a set of *principals*, who are instances of the roles. We consider two kinds of principals, the *honest* ones and the *dishonest* ones. The honest ones perform only the actions planned in the protocol specification, while the dishonest ones may also perform other actions, such as sending messages they should not send, or altering the messages they are supposed to send. As is standard, we assume, without loss of generality, that all the dishonest principals cooperate and are all "merged" under the identity of the intruder. The *intruder* is a principal who tries to violate one (or more) of the properties the protocol is claimed to ensure. The intruder is always present, even if he is known by no honest principal. The actions he can perform are also specified in the HLPSL.

We now first give an example of a protocol specification in HLPSL, and then give the commented grammar of the HLPSL in two parts: first the message syntax and then the protocol specification syntax. Note that we have designed the HLPSL syntax to be close to the protocol specifications that can be found in text-books or in technical drafts.

2.2 Overview of the HLPSL syntax

Consider, as an example, the HLPSL specification for the Needham-Schroeder Public-Key protocol given in Figure 2.1. A protocol specification consists of the following parts:

- the name of the protocol;
- the identifiers used to specify the protocol;
- the knowledge each role in the protocol is supposed to have at the beginning of a protocol session;
- the sequence of messages of the protocol;
- the definition of an initial state in which roles as well as their initial knowledge are instantiated with actual principals together with the initial value of the knowledge items;
- the intruder's capabilities;
- the intruder's knowledge, a possibly empty list of values;

PROTOCOL NSPK;
IDENTIFIERS
  A, B : `user`;
  Na, Nb : `number`;
  Ka, Kb : `public_key`;
KNOWLEDGE
  $A : B, Ka, Ka', Kb$;
  $B : A, Ka, Kb, Kb'$;
MESSAGES
  1. $A \rightarrow B$: $\{Na, A\}Kb$
  2. $B \rightarrow A$: $\{Na, Nb\}Ka$
  3. $A \rightarrow B$: $\{Nb\}Kb$
SESSION_INSTANCES
  $[A : a; B : b; Ka : ka; Kb : kb]$
  $[A : a; B : I; Ka : ka; Kb : ki]$;
INTRUDER Divert, Impersonate;
INTRUDER_KNOWLEDGE I, b, ka, kb, ki;
GOAL Correspondence_Between A B ;

Figure 2.1: HLPSL specification of the Needham-Schroeder Public-Key protocol NSPK

- the goal of the intruder, i.e. the flaw we are looking for.

We now indicate how to specify each of these sections in HLPSL syntax.

2.3 The HLPSL grammar

2.3.1 Keywords and identifiers

In all grammars in this document, we denote terminal symbols by using the `typewriter font` and non-terminal symbols by using the *slanted font*. Note that the parser is case-insensitive on keywords, but is case-sensitive on identifiers and instances.

It is possible to comment the specification by using the % symbol. Anything following a % symbol in the protocol specification will be ignored until the end of the line.

Tokens are separated by spaces, newline or tabulation. The grammar doesn't rely on white-spaces to apply a rule. However, there are some places where the lexicographic analysis may be mislead into wrongly parsing a keyword as an identifier. In these places, tokens have to be separated, using either a white space (at least one space, newline or tabulation) or a comment. We have indicated these places using the symbol ␣.

There is a special meaning associated with the prime character `'` in identifiers. If an identifier ends with this character, it is assumed that this identifier refers to a private key (corresponding to the public key denoted by the same identifier without a terminating prime).

The data types of HLPSL are the following:

```
   id_type ::=  number
             |  user
             |  public_key
             |  symmetric_key
             |  function
             |  table
     ident ::=  [a-zA-Z][a-zA-Z_0-9]*[']?
ident_list ::=  ident
             |  ident , ident_list
       int ::=  [0-9]+
```

The meaning of the different data types is the following:

- `number` represents any piece of data without any special property.
- `user` represents names of roles.
- `public_key` represents the type of public keys and, implicitly, private keys. Only the public keys need to be declared in the identifier section. The private key associated to a public key K is written K`'`.
- `symmetric_key` represents symmetric keys. This type is now handled internally exactly in the same way as `number` since composed symmetric keys are permitted.
- `function` represents a *hash* function or algorithm. We do not distinguish here between the function name and the algorithm computing this function. Function names can be sent in messages.
- `table` represents a table of public keys. The main use of tables is for protocols describing a public key server. In this case, upon receiving a principal name, the server can, for example, reply with the public key of this principal.

In the HLPSL, we have two kinds of identifiers that are syntactically both *ident*. The first kind is *protocol variables*, e.g. role names like A or names of protocol items like Na, that are used to describe the protocol independent of the concrete values used for these variables. For example, there may be a concrete agent a, which runs several sessions where he plays either role A or role B of the NSPK. The other kind is *instances* of these protocol variables, like a.

To easily distinguish between roles and their instances, throughout this document we will denote protocol variables by an identifier starting with an upper-case letter like A and B, and instances will be denoted by identifiers that start with a lower-case letter like a and b. The only exception is the name of the intruder, denoted by I. Note that the intruder may legally play a role in the protocol (under his real name).

Also, we use the terms *role* (or equivalently: *user*) for the participant names in the protocol description, e.g. A, and the terms *principal* (or equivalently: *agent*) to refer to instantiations of such a role, e.g. a.

Note that almost the complete protocol specification is free from instantiated identifiers with the exception of the sections in which the session instances and the intruder knowledge are declared.

2.3.2 Messages

The non-terminal symbol *msg* appears in three different parts of the specification: the knowledge part, the message part, and the intruder knowledge part. In the first two cases, all identifiers appearing in a message should be protocol variables, in the latter case it should be instances (declared in the session instances part).

The messages are defined as follows:

```
msg ::=  ident
      |  msg , msg
      |  ( msg )
      |  {msg } msg
      |  ident [ ident ]
      |  ident [ ident ]'
      |  ident ( msg )
      |  msg XOR msg
```

These rules are prioritized according to the order in which they are given, e.g. *m1* , *m2* `XOR` *m3* is interpreted as *m1* , (*m2* `XOR` *m3*).

The informal meaning of the constructors used is the following:

- `_,_` is the concatenation, by pairing, of two message terms.
- `(_)` can be used to compose message parts into a single one. It can be used for a composed key, and for `XOR` encryption.
- `{_}_` expresses the encryption of the first term by the second one. The type of the encryption (asymmetric or symmetric) is given by the type of the key used for encryption. Encryption is asymmetric in case of a public key or a table, and is symmetric otherwise.
- `_[_]` is valid whenever the first argument is of type `table` and the second argument is a user. In this case, `T[A]` represents the public key of user `A` given in the table `T`. The private key of `A` is written `T[A]'`.
- `_(_)` is valid whenever the first argument is of type `function`, and it expresses the application of this function with arguments being given by the second term.
- `_XOR_` expresses that the two given terms are XOR-ed. Note that this operator is only included for future versions of the AVISS tool; currently only the CL tool has made experiments with modeling the properties of this operator, see Section 6.2.

2.4 Protocol specification

A valid protocol specification in HLPSL is a text that can be parsed using the following rule:

```
protocol_description ::=  PROTOCOL␣ident ;
                          identifiers_declaration
                          knowledge
                          messages
                          role_declaration
                          session_declaration
                          intruder_definition
                          goal_list
```

We now explain how to obtain these non-terminal symbols.

2.4.1 Protocol name

The *name* of the protocol is simply given by:

PROTOCOL name_of_the_protocol;

Note that name_of_the_protocol can be any identifier, and is currently not used thereafter.

2.4.2 Identifiers declaration

The identifiers used for the description of the protocol are given using, for example:
Identifiers
Kas, Kbs, Kab : `symmetric_key`;
...
declares three new identifiers, Kas, Kbs and Kab, of type `symmetric_key`. The types are among the following: `user`, `public_key`, `symmetric_key`, `number`, `function` and `table`.

More formally, the identifiers are declared using the following rules:

```
       identifiers_declaration ::=  IDENTIFIERS␣
                                    variables_declarations_list
   variables_declarations_list ::=  variables_declaration
                                 |  variables_declaration
                                    variables_declarations_list
         variables_declaration ::=  ident_list : id_type ;
```

Note that in HLPSL, unlike other protocol specification languages, there is no need to explicitly specify which identifiers are considered to be *fresh*, i.e. generated during the protocol run, like nonces and session keys. We interpret as fresh the objects that are not declared in the initial knowledge of any role. The creator of the fresh item is the sender of the first message the item appears in. In case of a freshly created public key, the corresponding private key is also created at the same time and by the same user.

Note also that we currently model, as it is often done, timestamps as nonces. This is only an approximation, which is however sufficient to find attacks in a number of protocols employing timestamps as fresh objects, when the attacks do not rely on particular properties of the timestamps. As discussed in Section 6.2, we are currently working on a more refined model of timestamps, which will allow us to express the *timeliness* of timestamps, i.e. that they are only valid for a certain period.

2.4.3 Knowledge declaration

We then define the initial knowledge each role of the protocol has before each session of the protocol as follows:
Knowledge
role1 : `List_of_terms`;
role2 : . . .
The syntax is straightforward. Roles always know at least their name; it is therefore unnecessary to specify it in the list of terms.

The following rules all have the side-condition that the identifiers used in the *ident_list* must be identifiers declared as users in the *identifiers_declaration* section.

```
                    knowledge ::=  KNOWLEDGE␣
                                   declaration_knowledge_list
   declaration_knowledge_list ::=  declaration_knowledge
                                |  declaration_knowledge
                                   declaration_knowledge_list
        declaration_knowledge ::=  ident_list : knowledge_list ;
               knowledge_list ::=  msg
```

2.4.4 Messages list

The *messages* sent and received during a session of the protocol are specified as follows:

MESSAGES
  1. *Sender* $\rightarrow$ *Receiver* : *msg*
  ...

where messages are numbered from 1 to n, *Sender* and *Receiver* are among the agents declared in the identifiers part of the specification, and *msg* is a message constructed as described above. It is mandatory that the receiver of message n is also the sender of message $n+1$ (except in the case of the last message). Note that this is not a restriction on the number of protocols that we can study: suppose receiver of message n is A and sender of message $n+1$ is B; then we can insert between these two messages an extra dummy message in which A sends to B her own name; this dummy message doesn't really change the protocol (i.e. it doesn't introduce or exclude any attacks), but ensures the property that the sender of a protocol step is the receiver of the previous one.

The HLPSL2IF translator checks if, given this list of messages and the initial knowledge of the roles as defined in the previous section, the protocol is executable. A protocol is said to be *executable* if all the roles, according to their initial knowledge and to the messages they receive, are able to compose the messages they have to send.

In the last of the following rules, the identifiers *ident* must be of type `user`.

messages	::=	`MESSAGES␣` *declaration_message_list*
declaration_message_list	::=	*declaration_message*
	|	*declaration_message declaration_message_list*
declaration_message	::=	*int. ident -> ident : msg*␣

2.4.5 Instances declarations

Once a generic specification of a protocol has been given using the above statements, we construct a particular scenario under which we study the protocol, and which consists in one or more instances of protocol principals. This scenario is built using two different kinds of declarations for principals in the initial state, namely session instances and, as a special case, role instance declarations.

Session instances declarations. Session instances allow us to specify groups of communicating users. More specifically, they allow us to specify scenarios of protocol executions where particular principals are ready to communicate with each other. Session instances are defined in the following way:

SESSION_INSTANCES
  $[A : a_1 ; B : b_1 ; \ldots]$
  $[A : a_2 ; B : b_2 ; \ldots]$
  ...;

In this example, there will be a first session, in which the role A is played by the principal a_1 (the other roles are similarly instantiated), the role B by the principal b_1, and another one in which the role of A is played by a_2. A same instance can be shared by different sessions. For example, we can have $a_1 = a_2$.

Note that the intruder may legally play a role in the protocol (under his real name), and hence be declared as a principal the session and role declarations.

Role instances declarations. Role instances are a restriction of session instances; their purpose is to provide the user with a facility to specify a more restrictive scenario than this is possible with session instances. This restricted scenario can lead to a much smaller search space and can hence increase efficiency. Note that replacing a role instance with an appropriate session instance never excludes any attacks.

As previously seen, it is possible to instantiate a bunch of roles at the same time using the SESSION_INSTANCES declaration. It is also possible to specify the principals running the protocol one at a time using the following declaration:

ROLE :
  $A\ [A\ :\ a_1\ ;\ B : b_1\ ;\ \ldots],$
  $B\ [A\ :\ a_2\ ;\ B : b_2\ ;\ \ldots],$
  $\ldots;$

In this example, one declares that the principal a_1 will play the role A, and that the principal b_2 will play the role B.

The following rules describe the different possibilities to declare a principal in the initial state. In the *role_instance* rule, *ident* is the name of the role whose instance is being defined. It must be an identifier of type `user` declared in the identifiers part, and be a sender or a receiver of at least one message in the message sequence.

In the *instance* rule, the first *ident* must be an identifier of type `user` declared in the identifiers part. The value of the second one is any valid identifier.

```
         role_declaration  ::=  ROLE :
                                role_declaration_list ;
    role_declaration_list  ::=  role_instance
                             |  role_instance , role_declaration_list
            role_instance  ::=  ident session_instance
      session_declaration  ::=  SESSION_INSTANCES
                                session_declaration_list ;
 session_declaration_list  ::=  session_instance session_declaration_list
                             |  session_instance
         session_instance  ::=  [ instance_list ]
            instance_list  ::=  instance
                             |  instance ; instance_list
                 instance  ::=  ident : ident
```

2.4.6 Definition of the intruder

The intruder's possible actions. The actions the intruder can perform are given in the INTRUDER field of the specification:

INTRUDER *Abilities_List*;

where the abilities of the intruder are any combination of the following:

- *Impersonate.* The intruder can send messages (built from his knowledge) under the identity of any principal, i.e. he can masquerade as any other principal.
- *Divert.* The intruder can intercept messages sent by one principal to another. The intercepted messages do not reach their destination, and their content can be analyzed by the intruder.
- *Eavesdropping.* The intruder can analyze the content of all the messages on the network sent by honest principals, even if they have not been sent to him.

The combination of abilities corresponding to the Dolev-Yao intruder model is defined with both Impersonate and Divert. In this case, it is always possible to simulate eavesdropping by simply replaying the messages intercepted.

The intruder knowledge. The initial knowledge of the intruder is defined as follows:

INTRUDER_KNOWLEDGE *List_of_Instantiated_Terms*;

The list of instantiated terms is either empty or a message. In the latter case, the *ident* upon which messages are built do not range over *identifiers*, but over *instances*. These instances are defined either in the SESSION_INSTANCES or ROLE parts of the specification as instances of identifiers.

We assume that the intruder also always knows at least his name, I, and at least one fresh value of each type (public/private key-pair, shared key, nonce). This allows the intruder to build a dummy message for any type.

Moreover, if the intruder plays a role of the protocol according to the session instances, he must also initially have the initial knowledge associated with that role, or precisely: the appropriate instantiations of the knowledge.

The following rules describe how to specify the intruder in the HLPSL.

```
intruder_definition ::= intruder_abilities
                        intruder_knowledge
```

```
intruder_abilities ::= INTRUDER␣abilities_list ;
     abilities_list ::= abilities
                     |  abilities_list , abilities
          abilities ::= divert
                     |  impersonate
                     |  eaves_dropping
```

```
intruder_knowledge ::= INTRUDER_KNOWLEDGE␣msg;
                    |  INTRUDER_KNOWLEDGE ;
```

2.4.7 Goals

The following goals may be defined in the HLPSL. If more than one goal is defined, the flaw searched for is a violation of one or more of these goals.

```
goal_list ::= goal
           |  goal
              goal_list
     goal ::= GOAL␣goal_description ;
```

```
      goal_description ::= correspondence_goal
                        |  secrecy_goal
                        |  short_term_secrecy_goal
                        |  authenticate_goal
   correspondence_goal ::= CORRESPONDENCE_BETWEEN␣ident , ident
          secrecy_goal ::= SECRECY_OF␣ident_list
short_term_secrecy_goal ::= SHORT_TERM_SECRET␣ident_list
     authenticate_goal ::= ident␣AUTHENTICATE␣ident␣ON␣ident_list
```

Correspondence goal. We specify a correspondence goal as a (symmetric) property between two principals as follows:

GOAL Correspondence_Between A B ;

It expresses that for each instantiation of the roles A and B, say a and b, whenever a or b finishes his part of the session, then the other principal has at least begun his part of the session.

Authenticate goal. During the first experiments, it turned out that correspondence goals are a declarative way to specify protocol goals, however many flaws are so subtle that they can not be detected by this kind of goal. For instance, some protocols actually ensure correspondence (i.e. the intruder can never successfully masquerade as an other agent), however the intruder may trick agents into accepting old or manipulated session keys and nonces. Hence we wanted to design a goal that allows to specify that every item M that is accepted by an honest agent is actually genuine, where genuine means that sender and receiver agree both on each other's name and the value of M, and this value is not a replay of a previous communication.

Hence we have developed a more versatile form of goal, the authenticate goal. It is expressive enough to subsume the concept of correspondence goals, though it is at first sight maybe less intuitive and easy to use. However, during the experimentation it turned out that this goal is very helpful in detecting all kinds of protocol weaknesses, and let to the discovery of a previously unknown flaw, as explained in Section 7.2.

It is possible to express an authentication goal between two roles A and B on any message part exchanged in the protocol:

GOAL A authenticate B on Na, Nb ;

The authenticate property expresses that whenever the principal a playing the role A has finished his part of the protocol session, and believes he has communicated with principal b playing the role B, then

1. there exists a principal b playing the role B. This principal believes he has sent some messages to a principal a playing the role A;
2. during the communication, two values na and nb were sent by b, standing respectively for Na and Nb;
3. principal a has received two values na' and nb', standing respectively for Na and Nb, which he believes were sent by b;
4. $na = na'$ and $nb = nb'$;
5. the values na and nb have been sent under these conditions at least as many times as they have been received. (Note that if na and nb are fresh items, e.g. nonces or session keys, then this condition implies that the element never has been used before.)

Long-term secrecy goal. It is possible to specify that the intruder should never know a term M by using the following construct:

GOAL Secrecy_Of M ;

where M is an identifier, either fresh or present in the initial knowledge of at least one role. This goal is violated whenever the intruder learns the value of an instance of M.

We recall the intruder may play a role, according to the instances in the initial state. In this case, M is not considered to be secret (in spite of the goal declaration) whenever the intruder, acting honestly, eventually knows the value of M.

Short-term secrecy goal. Finally, it is possible to express that a secret term may be compromised by the intruder, possibly by an off-line cryptographic attack, long after the session the secret belongs to is over. This is done by using a goal of short-term secrecy. The short-term secrecy goal behaves exactly like the long-term secrecy goal, except that the secret value is given to the intruder after the session is over. Once the value is given, it is no longer considered to be secret. This goal is specified in the following way:

GOAL Short_Term_Secret Kab ;

2.5 The translator HLPSL2IF

2.5.1 Overview

The aim of the HLPSL2IF translator is to map HLPSL into IF and thereby associate an operational semantics to protocol specifications in terms of rewrite rules. In order to make this translation, a nontrivial semantical analysis of the specification is done. The most interesting part of this analysis is the management of the knowledge of the principals, which we will now discuss in detail.

Note that the grammar of the HLPSL given above describes the input language of the HLPSL2IF translator, however it is possible that a specification conforming to this grammar is rejected by the translator if the semantic analysis described below finds an error in the protocol specification. Such an error can be protocol identifiers that have not been declared or similar minor mistakes, or, most notably, if the protocol can not be executed by honest agents, since they don't have the necessary knowledge to compose all messages they have to send.

The HLPSL2IF translator is written in Objective Caml, a language of the ML family, and it consists of more than 200kb of source code. The sources are split in 14 modules. The executable is compiled statically, thus enabling its use also on platforms where Objective Caml is not installed.

2.5.2 Knowledge management

The knowledge of a role in a protocol is augmented at each step where it receives a message or where it generates fresh data for preparing a new message. In order to guarantee that the protocol can be executed, we have to check that all the messages can be composed and sent to the right role. We assume that roles check the messages they have received so far.

The knowledge of each role can be decomposed into three parts:

- the initial knowledge, declared at the beginning of the protocol,
- the acquired knowledge, obtained by decomposition of the received messages,
- the generated knowledge, created for composing a message (fresh knowledge).

A protocol is executable if each role can compose the messages it is supposed to send. For some messages, a role will reuse parts of some previously received messages. Hence a role has to update its knowledge as soon as it receives a message: it has to store the new information, and to check if this new material can be employed for decoding ciphered texts from former messages that could not be decoded before by lack of the right key. We ensure knowledge updating by applying a decomposition function on the knowledge of a role until a fix-point is reached. Note that this function is also able to handle composed keys in messages.

The knowledge of roles is used twice during the construction of a receive/send rule, first when the role receives a message, in order to know which parts of this message it can access or analyze, and second when it sends a message, in order to check whether it has enough information to compose this message. Two functions are used to achieve these tasks, namely the *compose* and the *expect* function. Since *expect* relies on *compose*, we first describe the latter function, which describes the computation of a message M composed by a role U at step i. The knowledge of the role is the union of his initial knowledge, of pieces of information it got in the received messages until step $i-1$ (included), and of the fresh identifiers it has created so far. In the list below, the first possible case is applied (where if a Fail exception is raised in some recursive call of compose then it is the output of the function):

$$\begin{array}{rcll}
compose(U, M, i) & = & t & \text{if } M \text{ is known by } U \text{ and named } t \\
compose(U, \langle M_1, M_2\rangle, i) & = & \langle compose(U, M_1, i), compose(U, M_2, i)\rangle & \\
compose(U, \{M\}_K, i) & = & \{compose(U, M, i)\}_{compose(U,K,i)} & \\
compose(U, T[A], i) & = & compose(U, T, i)[compose(U, A, i)] & \\
compose(U, T[A]', i) & = & (compose(U, T, i)[compose(U, A, i)])' & \\
compose(U, M, i) & = & fresh(M, x_{time}) \text{ if } M \text{ is a fresh value at step i} & \\
compose(U, M, i) & = & \mathtt{Failure-of-Compose} & \text{otherwise}
\end{array}$$

Beside for composing his own messages, a role has also to verify information it receives in messages: if it is supposed to receive some piece of information it already knows, it has to check that what it receives is what it had recorded. A role is also aware of the shape (or external structure) of the messages it is supposed to receive. In a realistic implementation of the protocol it will check that everything it can access in some received message corresponds to what it expected at this step.

These checks are performed by the following function, where a role U tries to compose an expected message M using its knowledge before step i, the step when it will receive this message. All unknown ciphers are replaced by new variables.

Note that K stands for a public key, K' for a private key (possibly through the use of a table), and SK for a symmetric key. Like for *compose* the first successful case in the definition below is applied, but unlike for *compose* the *expect* function never fails.

$$\begin{array}{rcll}
expect(U, M, i) & = & compose(U, M, i) & \text{if no } \mathtt{Failure\text{-}of\text{-}Compose} \\
expect(U, \langle M_1, M_2\rangle, i) & = & \langle expect(U, M_1, i), expect(U, M_2, i)\rangle & \\
expect(U, \{M\}_K, i) & = & \{expect(U, M, i)\}_{compose(U,K',i)'} & \text{if no } \mathtt{Failure\text{-}of\text{-}Compose} \\
expect(U, \{M\}_{K'}, i) & = & \{expect(U, M, i)\}_{compose(U,K,i)'} & \text{if no } \mathtt{Failure\text{-}of\text{-}Compose} \\
expect(U, \{M\}_{SK}, i) & = & \{expect(U, M, i)\}_{compose(U,SK,i)} & \text{if no } \mathtt{Failure\text{-}of\text{-}Compose} \\
expect(U, M, i) & = & x_{U,M,i} & \text{otherwise}
\end{array}$$

2.5.3 Translator options

Files written in the HLPSL are parsed and compiled using the HLPSL2IF translator, which accepts different flags which affect its output. The translator is called by:

```
hlpsl2if [options] file
```

where `file` is the file containing the HLPSL specification of the protocol to be analyzed. If no options are given, the HLPSL2IF translator performs only the check if the protocol is executable. When the option

```
    --all
```

is given, the translator produces all the IF rules that describe the protocol. Alternatively, one can use any combination of the following options to generate only a part of the IF rules:

- `--init` prints the initial state of the protocol description. This includes descriptions of the principals and the intruder knowledge as given in the intruder knowledge part of the HLPSL specification of the protocol.

- `--rules` prints, for each message in the protocol description, a rule describing the actions a role, played by an honest principal, carries out when it receives the message and sends a reply.
- `--intruder` prints the intruder rules. These rules are set in accordance with the specified intruder abilities.
- `--goal` prints rewrite rules corresponding to the negation of the property given as a goal in the HLPSL description of the protocol.
- `--simplif` prints simplification rules for the intruder knowledge.

As explained in Section 3, there is a restricted variant of the standard model, the *typed* model. In this model, honest principals only accept type correct messages (hence excluding type-flaw attacks). This variant of the rules can be obtained by the option

```
--typed
```

which can be used independently of the other options.

3. The Intermediate Format IF

3.1 Introduction

Architecture. The Intermediate Format (IF) is the interface between the various tools: as shown in Figure 1.1, the HLPSL2IF translator automatically translates HLPSL protocol descriptions provided by the user into the IF, so that the different analysis tools can work on it. 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), and yet this format should be as independent as possible from the employed analysis methods of the various tools.

This section provides a framework for understanding the various tools that work on the IF. It also provides a kind of "programmer-manual", i.e. the documentation for developers who plan to connect their own protocol analysis tools with the IF and our system.

The IF describes a protocol in terms of rewrite rules of first-order equational logic; equivalently, it can be regarded as an infinite-state transition system with an initial state, transition rules, and a goal (attack) predicate that defines if a given state is an attack or not (i.e. a state-based safety property).

Semantics. The semantics of the HLPSL can be defined in terms of the IF: assuming that we already have defined an unambiguous semantics for the IF (which we do below), then the translation performed by the HLPSL2IF translator defines an operational semantics for the HLPSL. This translation reflects the general model of protocols underlying the HLPSL; for instance the system currently allows a restricted, typed variant of the default model (in this variant, agents only accept type-correct messages) as an option of the HLPSL2IF translation.

It remains, hence, to define an unambiguous semantics for the IF. Since the IF is formulated as rewrite rules, it already has a simple and straightforward semantics as an equational theory. In other words, we can see the IF as a fragment (resulting from a syntactic restriction) of equational logic. However, this straightforward semantics is not fully satisfactory. This is because the tools that work on the IF are not required to fully implement a semi-decision procedure for equational logic; moreover, we actually have a closer intuition on the meaning of each term in the IF rules than that they are simply terms in equation logic. We thus need a *tighter* semantics in which function symbols are interpreted (i.e. every term is associated with a particular meaning, which can then be exploited by the different tools). As remarked in the original project proposal, we will consider here only the straightforward equational semantics and give a preliminary definition of the tighter semantics, whose formal definition we postpone to future work.

Overview. We start with describing the *Generic IF* in Section 3.2: this is a super-set of the IF language actually used by the HLPSL2IF translator; the Generic IF is however easy to define and gives an overview of the concepts used, and it will be stable even when only small details of the HLPSL2IF model need to be changed. In Section 3.3, we replace the grammar rules of the Generic IF with more specific rules that give a tighter description of the output language of HLPSL2IF. For every term we give an informal semantics and describe which mechanisms are employed; the verification tools taking an IF specification as input may exploit this description by replacing the mechanisms with a specific solution that works efficiently in the context of the chosen analysis method.

Outlook. As remarked above, in future work we will focus on giving a formal semantics for the terms and concepts used in the IF, extending the informal description we give here. This is necessary both if one wants to prove the correctness of optimizations that the analysis tools perform, and if one wants to formally reason about the properties of the employed model and prove, for instance, equivalence with alternative models.

3.2 The Generic IF

We present the grammar of the Generic IF in an extended BNF style. Terminal symbols are displayed in `typewriter` font, non-terminal symbols in *slanted* font, and EOL denotes the End-Of-Line character. Comments are lines that start with two hashes `##`; they are not displayed in the grammar. A single hash `#` denotes additional information that the OFMC back-end and the SATMC back-end read and exploit (the CL back-end ignores this information). Since lines beginning with a single `#` must have a fixed form, we include their definition in the grammar we give below.

An IF-file starts with a flag specifying if the underlying model is untyped or typed (see below for details), followed by a list of labeled rules:

```
   IFFile ::= TypeFlag EOL
              LabeledRule*
 TypeFlag ::= # option=untyped | # option=typed
```

Each rule is preceded by a label that gives the rule a name and a category (where we assume that all rules of an IF-file have unique names):

```
 LabeledRule ::= Label EOL
                 Rule EOL
       Label ::= # lb=RuleName , type=RuleCategory
    RuleName ::= Alphanum+
    Alphanum ::= [ a-zA-Z0-9_ ]
RuleCategory ::= Protocol_Rules | Invariant_Rules | Decomposition_Rules
                 | Intruder_Rules | Init | Goal
```

The rule categories have the following meanings:

- `Init` represents the initial state.
- `Protocol_Rules` are rules that describe the behavior of the honest agents.
- `Goal` is a description of the attack states.
- `Decomposition_Rules` and `Invariant_Rules` describe static aspects of the model (e.g. that the intruder can find out the content of a message if he has the appropriate key).

A rule can either be a state (e.g. initial state or goal state) or a state transition; a state is a multi-set of facts, separated by the associative, commutative, and idempotent operator “.”.

```
 Rule ::= State EOL
          => EOL
          State
        | State
State ::= Fact | Fact . State
```

The semantics of a state transition is the following: if the left-hand side (LHS) of the rule can be matched (or unified, in the case of the lazy intruder; see Section 4.1) against a subset of the current state (i.e. every required fact currently holds), then the rule can *fire*, i.e. the identified subset of the state is replaced with the right-hand side (RHS) of the rule, where all variables in the rule's RHS are substituted according to the unifier. Intuitively, a protocol has a flaw if from the initial state some state is reachable via the state transition rules, such that for this state one of the goal states is satisfied.

There are some rules of type `Simplification`. These rules bring a set of facts into a normal form (as will be explained in detail in the Section 3.3) and are hence also called normalization rules. The goal-check (i.e. checking whether a state is a goal state) may only be applied to states that are normalized according to all normalization rules, i.e. when no further simplification of the state is possible.

The IF uses the following kinds of facts (which we will explain in more detail in the following sections):

- *MessageFact*, describing a message that was sent but has not yet been received;
- *Principal*, expressing the local state of an honest agent;
- *IntruderKnowledge*, expressing that the intruder has learned a certain message;
- *TimeFact*; giving the current time point; and
- The facts *Secret*, *Give*, *Witness*, and *Request* help to keep track of the goals, e.g. *Secret* states that a message is a secret belonging to a certain session. These kinds of facts will be explained in detail in Section 3.3.

```
             Fact ::= Principal | MessageFact | IntruderKnowledge | TimeFact
                    | Secret | Give | Witness | Request
        Principal ::= w(Message,Message,Message,Message,Message,Message,Message)
      MessageFact ::= m(Message,Message,Message,Message,Message,Message)
IntruderKnowledge ::= i(Message)
         TimeFact ::= h(Message)
           Secret ::= secret(Message,f(Message))
             Give ::= give(Message,f(Message))
          Witness ::= witness(Message,Message,Message,Message)
          Request ::= request(Message,Message,Message,Message)
```

Messages are modeled by appropriate terms. Here atomic messages are modeled as constants, concatenation and different kinds of encryption as binary function symbols, and type information as unary function symbols. Handling type information inside the message terms allows the rules to explicitly use the type information (as will be explained in the section about the typed version, i.e. Section 3.3.1).

Message terms (as used in rules) may also contain variables for any subterm. To allow a distinction between names of constants (atomic messages like `Na`) and variables names (like `xNa`), we require that variable names start with the character `x`. Note that the scope of a variable is always the rule in which it is used.

```
 Message  ::=  Constant | Variable | Number
            |  TypeInfo(Message)
            |  CryptOp(Message, Message)
            |  Message'
            |  s(Message)
 CryptOp  ::=  crypt | scrypt | c | funct | rcrypt | tb
 TypeInfo ::=  mr | nonce | pk | sk | fu | table
 Constant ::=  [ a-zA-Z^x ] Alphanum*
 Variable ::=  xAlphanum+
 Number   ::=  [ 0-9 ]+
```

The cryptographic operators have the following meaning:

- `crypt` denotes asymmetric encryption (with a public or private key).
- `scrypt` denotes symmetric (shared-key) encryption.
- `c` denotes concatenation.
- `funct` denotes the application of a cryptographic (hash) function.
- `rcrypt` denotes XOR-encryption; as pointed out in the HLPSL documentation above, the XOR operator is not yet integrated into the AVISS tool and its support is only preliminary.
- `tb` denotes looking up the public key of an agent in a key-table.

Note that here we do not assume any properties of the operators, in particular we do not assume `c` to be associative. (All properties of the operators are introduced by appropriate rewrite rules in the IF.)

The type functions have the following meaning: `mr`: principal name, `nonce`: nonce, `pk`: public key, `sk`: shared key, `fu`: (hash) function, `table`: table of public keys.

Note that private keys are not a type of their own, but denoted by adding a prime symbol `'` to a public key, i.e. `pk(ka)'` represents the private key for `pk(ka)` and `tb(table(t),mr(a))'` is the private key belonging to `mr(a)`'s public key in the key-table `t`.

3.3 A closer description of the IF

The Generic IF we described in the previous section is a super-set of the actual output language of the HLPSL2IF translator. We now take a closer look and replace some grammar rules of the Generic IF with more specific ones. We proceed in a bottom-up fashion since we assume the reader by now already has a rough idea of the IF. Please note that all above grammar rules still apply as long as they are not replaced with a more restrictive definition.

3.3.1 Model of messages

Atoms. Atomic messages are either constants or variables. A constant always carries type information, like `sk(kab)`, while variables are untyped, i.e. they can represent *any* message. (More about the typing will be explained below.)

Many constants are messages that are created during the protocol run, e.g. fresh nonces. In the model, a fresh item is represented by a message term of a special form. This form is used to ensure two important properties: first, the item has never been used before (i.e. it is different from all previous items); second, the intruder may not guess the item (i.e. he can only find out an item when he sees it in cleartext). To ensure these properties, we use terms of the form

TypeInfo`(c(`*Constant*`,`*time*`))`

where *TypeInfo* is the type of fresh item, e.g. `Nonce`, *constant* is the name the item has in the protocol description, e.g. `Na`, and *Time* is the time point when the item was generated. For instance, `nonce(c(Na,`5`))` stands for a nonce that was created at time point 5 by an honest agent for the purpose of the item `Na` of the protocol.

Note that both required properties are ensured by this construction: all generated nonces and keys are represented by different terms, since they have either different names or creation time. Further, since there is a type information around the item, it is an atom; hence there is no way for the intruder to compose this term (since the result of a composition can not be an atom).

As a further special case, the intruder shall also have the ability to generate fresh items of all types; however the intruder doesn't really need *fresh* items since the honest agents never compare nonces they receive with all the old nonces they have seen before. So whenever the intruder uses a nonce from his own knowledge, he can always use the same nonce. The straightforward construction is to give the intruder an own 'fresh' item of each type in his initial knowledge, where 'fresh' means only that it must be represented by a term different from all other messages. This is achieved by terms of the form *TypeInfo*`(c(ni,ni))`.

Atomic	::=	*TypeInfo*(*Const*) | *Variable*
Const	::=	*Constant* | `c(`*Constant*`,`*Time*`)` | `c(ni,ni)`
Time	::=	*Number* | `xTime` | `s(`*Time*`)`

Typed model of messages. The HLPSL2IF translator has an option that provides an alternative model, the *typed* model, which is a restriction of the standard model. The typed model may be exploited by analysis tools, like the SATMC back-end, which cannot use the standard, unrestricted model (see Section 6.3 for more details). Other approaches, like the OFMC back-end and the CL back-end, may employ the restricted typed model to speed-up the search for flaws.[1]

Roughly speaking, in the typed model, the agents only accept type-correct messages; this prevents *type-flaw attacks*, as shown in the following example, the Otway-Rees protocol:

1. $A \rightarrow B : M, A, B, \{Na, M, A, B\}_{Kas}$
2. $B \rightarrow S : M, A, B, \{Na, M, A, B\}_{Kas}, \{Nb, M, A, B\}_{Kbs}$
3. $S \rightarrow B : M, \{Na, Kab\}_{Kas}, \{Nb, Kab\}_{Kbs}$
4. $B \rightarrow A : M, \{Na, Kab\}_{Kas}$

Suppose the intruder has recorded the first message of an honest agent A. He can't decrypt the encrypted part, but he can send it as message 4 to A, posing to be B. A will decrypt the message

[1]Note, however, that OFMC and CL both apply a more efficient intruder model, the lazy intruder; using this model the typed version typically doesn't constitute a significant restriction as explained in detail in Section 6.1.5.

(as it is encrypted with the correct key) and find in the first position her nonce Na; hence, A will accept the rest of the message, i.e. the term M, A, B, as the new session key Kab. Since the intruder knows these items (they were sent in clear text in the first message), he now knows the session key which will be used by A for the communication with B.[2]

The typed model excludes type-flaws like this one since, for example, the session key should be of type shared key, while the actual content of the message faked by the intruder is a concatenation of items of several types (a nonce and two principal names). The idea how to express the typed model is to give type information to variables. For instance, in the untyped model, the pattern for the message expected by A in step 4 of the Otway-Rees protocol is

`scrypt(sk(kas),c(nonce(na),xKab))` ,

i.e. the key `xKab` can be any message term; in the typed model the pattern is

`scrypt(sk(kas),c(nonce(na),sk(xKab)))` ,

i.e. the key `xKab` can only be a constant of type shared key.

With respect to the grammar above, the difference in the grammar for the typed version is only that the rule for variable

$$\boxed{\textit{Variable} ::= \texttt{x}\textit{Alphanum}^{+}}$$

is replaced by the rule

$$\boxed{\textit{Variable} ::= \textit{TypeInfo}(\texttt{x}\textit{Alphanum}^{+})}$$

From the infinite set of messages the intruder can generate from his knowledge, an analysis tool may only consider those messages that can be accepted by an honest agent as a valid protocol message. Since the typed model restricts this set of acceptable messages, a tool working on the typed model may consider a heavily reduced universe of possible messages that contains only the type-correct messages in the format of the protocol steps. Note that this language is finite if one restricts the setting to a finite (bounded) set of constants.

Composed messages. In the IF (as opposed to the "larger" Generic IF), message terms are built from atomic messages by concatenation, inverting (i.e. applying the prime operator to public keys), and composition with the cryptographic operators. However, we can exclude some terms that don't make sense even in the untyped model: asymmetric encryption only makes sense when the first argument is a public or a private key (i.e. a term that can be inverted), and applying hash functions (key-tables) only makes sense when the first argument is of type function (table). Note that for symmetric encryption and concatenation any message can be used as the first argument (e.g. for modeling composed keys).

The grammar for composed messages is:

[2] Note that such an attack may be prevented by the actual implementation of the protocol, e.g. if all protocol items are assigned a particular length (in bits) and the messages to be matched, in this case Kab and M, A, B, have different lengths.

$$\begin{array}{rcl}
\mathit{Message} & ::= & \mathit{Composed} \mid \mathit{Atomic} \\
\mathit{Composed} & ::= & \mathit{Concatenation} \mid \mathit{SymmetricCypher} \mid \mathit{XOR} \mid \mathit{PublicCypher} \\
 & \mid & \mathit{Function} \mid \mathit{Keytable} \mid \mathit{KeytableApp} \\
\mathit{Concatenation} & ::= & \texttt{c(}\mathit{Message},\mathit{Message}\texttt{)} \\
\mathit{SymmetricCypher} & ::= & \texttt{scrypt(}\mathit{Message},\mathit{Message}\texttt{)} \\
\mathit{XOR} & ::= & \texttt{rcrypt(}\mathit{Message},\mathit{Message}\texttt{)} \\
\mathit{PublicCypher} & ::= & \texttt{crypt(}\mathit{Invertible},\mathit{Message}\texttt{)} \\
\mathit{Invertible} & ::= & \texttt{pk(}\mathit{Const}\texttt{)} \mid \texttt{pk(}\mathit{Const}\texttt{)'} \mid \mathit{KeytableApp} \mid \mathit{Variable} \mid \mathit{Variable}\texttt{'} \\
\mathit{Function} & ::= & \texttt{funct(fu(}\mathit{Const}\texttt{),}\mathit{Message}\texttt{)} \mid \texttt{funct(}\mathit{Variable},\mathit{Message}\texttt{)} \\
\mathit{Keytable} & ::= & \texttt{table(}\mathit{Const}\texttt{)} \mid \mathit{Variable} \\
\mathit{Agent} & ::= & \texttt{mr(}\mathit{Constant}\texttt{)} \mid \mathit{Variable} \\
\mathit{KeytableApp} & ::= & \texttt{tb(}\mathit{Keytable},\mathit{Agent}\texttt{)} \mid \texttt{tb(}\mathit{Keytable},\mathit{Agent}\texttt{)'}
\end{array}$$

Note that, for the typed version, a further restriction can be made: the type information for variables, e.g. in the definition of *Invertible*, can only be of those types that are allowed for the constants, e.g. only `pk` and `tb` in the case of *Invertible*.

3.3.2 Model of honest agents

The behavior of an agent is determined by his local state, called also *Principal* fact, and the transition rules that describe a transition of an agent's state (where messages are sent and received).[3]

$$\begin{array}{rcl}
\mathit{Principal} & ::= & \texttt{w(}\mathit{Step},\mathit{Agent},\mathit{Agent},\mathit{MsgList},\mathit{MsgList},\mathit{Boolean},\mathit{Session}\texttt{)} \\
\mathit{Step} & ::= & \mathit{Number} \mid \mathit{Variable} \\
\mathit{MsgList} & ::= & \texttt{etc} \mid \mathit{Variable} \mid \texttt{c(}\mathit{Message},\mathit{MsgList}\texttt{)} \\
\mathit{Boolean} & ::= & \texttt{true} \mid \texttt{false} \mid \mathit{Variable} \\
\mathit{Session} & ::= & \texttt{s(}\mathit{Session}\texttt{)} \mid \mathit{Number} \mid \mathit{Variable}
\end{array}$$

A *Principal* fact contains all the information from which the agent's behavior depends. The first parameter is the protocol step the agent is in; by convention, an agent in step i awaits the ith message of the protocol; the step in the initial state of the initiator of a protocol is 0.

The second parameter is the sender from which the agent expects the next message; in this position there can be a variable to indicate that either the agent is the initiator (and hence doesn't expect a message) or the sender of the message isn't determined.

The third parameter is the name of the agent which is described by the *Principal* fact.

The fourth parameter is a list of messages: the acquired knowledge of the agent, i.e. all information he stored during the run of the protocol (e.g. nonces he has generated and needs to check later).

The fifth parameter is also a message list, and it describes the long-term knowledge of the agent. The long-term knowledge is never changed by any rule; it contains typically agent names, long-term keys and so forth. Note that in this position we do not necessarily store all information that the agent needs to execute the protocol, but only that which is important for the applicability of the agent's rules. For instance, in a protocol with only public key encryption one might leave the fact implicit that every agent knows his private key, since for the formulation of the rules it is sufficient to check that each message is encrypted with the correct public key.

[3]Note that in the typed model, the *Variable* in the rules for *Step*, *MsgList*, *Boolean*, and *Session* has no type information (recall that in the typed version, all variables have a type constraint). However we didn't want to make the grammar unnecessarily complicated.

The sixth parameter is a boolean flag only used in conjunction with a search strategy, i.e. tools may safely ignore it. It will be explained below together with the impersonate rules.

Finally, the seventh parameter gives the number of the session the agent is in. This is a unique number for every session instance specified in the HLPSL file. Note that our model supports an unbounded number of consecutive runs of the same session, i.e. after agents have finished a run of the protocol they can always start a new run. To give different runs of the same session a unique identification, for every new run we use the `s` function; i.e. `s(1)` is the second run of session number `1` (and not session `2`).

More generally, all transition rules always have the same number of principal terms on the LHS and on the RHS, in other words, the transition rules can neither introduce nor remove principals from the state (but only change the local state of the principal). Hence there is no bound on how many sessions a principal can perform, and therefore the number of fresh items that can be generated is also unbounded. This is what makes our model infinite.

The messages sent by honest agents (and also by the intruder) are stored in a message fact:

MessageFact ::= `m(`*Step*`,`*Agent*`,`*Agent*`,`*Agent*`,`*Message*`,`*Session*`)`

This fact has the meaning that a message was sent by some agent and has not been received not yet; hence, our model is asynchronous (as sending and receiving of a message are not joined in one atomic operation).

MessageFact has a similar format as the *Principal* fact: the first parameter is again a step number.

The second parameter gives the real sender of the message; even when the intruder fakes a message impersonating other agents, this field contains the truth about the message origin. However, in the message rules for honest agents, if a *MessageFact* appears on the LHS of a rule, then this field always contains a variable, since no agent can see the real sender. Hence, the tools may ignore this field as well.

The third parameter is the official sender, i.e. the person that the real sender claims to be. Either official and real sender are identical or the real sender is the intruder (honest agents never claim to be somebody else).

The fourth parameter is the receiver of the message, i.e. the person the message is intended for. Honest agents can only receive messages that were intended for them.

The fifth parameter is the sent message itself, and the sixth parameter is the session number.

The behavior of the honest agents is described by transition rules of a specialized form that we call *MessageRule* (the other symbols introduced in the following grammar rule for IF rules will be explained below):

```
        Rule ::= InitialState | MessageRule | IntruderRule
               |  GoalState | SimplificationRule
InitialState ::= h(xTime). State EOL
 MessageRule ::= h(s(xTime)). MessageFact? . Principal EOL
                 => EOL
                 h(xTime). MessageFact? . Principal . GoalFact* EOL
```

A *MessageRule* expresses that an honest agent is prepared to perform a step if he receives a certain message and currently is in an appropriate local state. He reacts by updating his local state (storing information he has learned from the received message and the fresh items he has generated) and sending a reply message. The *MessageFact* is optional on either side: there is no

MessageFact on the LHS of the rule, if the step in question is the initial step of the protocol (so the agent doesn't await a message before he can start the protocol); further, there is no *MessageFact* on the RHS of the rule whenever the step in question is the final step of the protocol (so the agent doesn't send a reply message after receiving a message).

The *MessageRule* presented already reflects an optimization: by default, one would expect individual rules for receiving messages and for sending the reply. However, in an asynchronous model we can assume that every agent directly answers to the messages he received, as opposed to a synchronous model, where interception of messages by an intruder is modeled as if the respective agent received the message but never reacts to it — this is not necessary in the asynchronous model.

Additionally, on the RHS there can appear several facts relevant for keeping track of the goals; these are called *GoalFact* and will be explained below.

As an example, we give here the rules for agent A in the NSPK protocol. We give the message rule where A sends the initial message, and the message rule where A receives the second message from B, answers with the third message, and accepts the run as completed (for the sake of simplicity we use a list notation for the knowledge of agents):

```
h(s(xTime)).w(0,xinit,xa,[],[xa,xka,xka',xb,xkb],xbool,xc)
=>
h(xTime).m(1,xa,xa,xb,crypt(xkb,c(xa,nonce(c(Na,xTime)))),xc).
w(2,xb,xa,[nonce(c(Na,xTime))],[xa,xka,xka',xb,xkb],true,xc)

h(s(xTime)).w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],xbool,xc).
m(2,xr,xb,xa,crypt(xka,c(xNa,xNb)),xc2)
=>
h(xTime).m(3,xa,xa,xb,crypt(xkb,xNb),xc2).
w(0,xinit,xa,[],[xa,xka,xka',xb,xkb],true,xc)
```

3.3.3 Model of the intruder

The intruder behavior is independent from the protocol; according to the intruder abilities specified in the HLPSL file, the intruder can perform the following actions:

- *Eavesdropping.* The intruder can see all the messages on the network sent by honest principals, even if they have not been sent to him.
- *Divert.* The intruder can prevent sent messages on the network from reaching their destination, i.e. the intruder can intercept the messages sent by honest principals.
- *Impersonate.* The intruder can send messages (built from his knowledge) under the identity of any principal, i.e. he can masquerade as any other principal.

Besides for these abilities, the intruder can always compose and analyze messages in his knowledge according to the model of Dolev and Yao [21].

IntruderRule ::= *EavesDropping* | *Divert* | *Impersonate* | *Analysis*

Since this behavior is protocol-independent, every tool that works on the IF can have a method-specific solution to model the intruder. For instance, as we discuss in detail in Section 4, the CL and the OFMC back-ends implement the *lazy intruder*, an efficient model of the intruder that reduces the high branching introduced by the impersonate rules using a symbolic representation of the possible messages the intruder could generate.

It is however necessary to specify the basic intruder model (without tool-specific concepts and optimizations) in the equational logic formalism. The HLPSL2IF translator hence generates the following static, protocol-independent, rules (recall that facts of the form `i(`*Msg*`)` denote that the intruder currently knows *Msg*):

```
If eavesdropping is selected:
EavesDropping ::= h(s(xTime)).m(x1,x2,x3,x4,x5,x6) EOL
                  => EOL
                  h(xTime).m(x1,x2,x3,x4,x5,x6).
                  i(x2).i(x3).i(x4).i(x5) EOL
                | h(s(xTime)).m(x1,x2,x3,mr(I),x5,x6) EOL
                  => EOL
                  h(xTime).i(x2).i(x5) EOL
```

The first alternative of this rule describes the intruder reading the content of a message (but not removing it from the network); the second alternative describes the intruder receiving a message as a regular participant of the protocol.

```
If divert is selected:
Divert ::= h(s(xTime)).m(x1,x2,x3,x4,x5,x6) EOL
           => EOL
           h(xTime).i(x2).i(x3).i(x4).i(x5) EOL
```

The generated impersonate rules are described below; they also subsume the intruder's ability to construct messages from his knowledge. For the analysis of the knowledge, the HLPSL2IF translator always produces the following rules:

```
Analysis ::= i(c(x1,x2)) EOL
             => EOL
             i(x1).i(x2) EOL
           | i(crypt(x1,x2)).i(x1') EOL
             => EOL
             i(x2).i(x1') EOL
           | i(crypt(x1',x2)).i(x1) EOL
             => EOL
             i(x2).i(x1) EOL
           | i(scrypt(x1,x2)).i(x1) EOL
             => EOL
             i(x2).i(x1) EOL
```

Although, as explained above, the behavior of the intruder is protocol-independent, in the IF we chose to specialize the impersonate rules to the given protocol to realize a standard search technique: it is sufficient to model an intruder who only generates messages that can actually be received by an honest agent — all other messages need not be considered. Hence, the impersonate rules allow the intruder to construct only meaningful messages in the sense of the given protocol and are therefore protocol-dependent.

For instance, suppose A is waiting for the reply message from B in the NSPK protocol, namely A expects to receive a message of the form $\{N_A, N_B\}_{K_A}$ where N_A is a nonce which A has just sent to B and N_B is arbitrary. The intruder has now two basic possibilities to form a message that matches the expectations: (a) if he knows A's public key and the nonce N_A, he can construct the message himself, and (b) if he earlier recorded a message of this form (which he couldn't decrypt), he may now replay it. Accordingly, we have two rules in this example:

```
w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],true,xc).    A is waiting for B
i(xka).i(xNa).                                     intruder can compose the message and
i(xNb)                                             he knows a value that can be used as N_B
=>
w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],false,xc).   A is still in the same state
i(xka).i(xNa).i(xNb).
m(2,I,xb,xa,crypt(xka,c(xNa,xNb)),0)

w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],true,xc).
i(crypt(xka,c(xNa,xNb)))                           intruder knows a message of that form
=>
w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],false,xc).
i(crypt(xka,c(xNa,xNb))).
m(2,I,xb,xa,crypt(xka,c(xNa,xNb)),0)
```

In this example, one can see the application of the boolean flag in principal facts: the impersonate rules can only be applied to principal facts where the flag is true, and after the impersonate rule was applied, the boolean flag in the used term is set to false. This simple mechanism prevents sending several intruder-generated messages to the same principal in the same state before he reacted to the first one.

Note that for the typed version, all variables are augmented with type constraints, namely instead of `xNb` there will be the term `nonce(xNb)`.

The grammar for the impersonate rules is:

Impersonate	::=	`h(s(xTime))`.*Principal* (.*IntruderKnowledge*)$^+$ EOL
		`=>` EOL
		`h(xTime)`.*MessageFact*.*Principal* (.*IntruderKnowledge*)$^+$ EOL

Note that there are no other rules that describe how the intruder can compose items from his knowledge to form new messages; these rules are implicit in the impersonate rules.

The impersonate rules can be very efficient, since a tool may only consider those intruder-generated messages that can actually be received by the honest agents; however this model is not complete in the sense of Dolev-Yao. The problem can be seen in the example above: the rule (a) allows the intruder to take an arbitrary item from his knowledge as nonce xNb, excluding the possibilities that xNb might itself be a composed item that is not directly in the intruder knowledge while all of its components are. One might say that the impersonate rules give only an approximation of the Dolev-Yao model. In fact, attacks can be excluded by this approximation. To illustrate this, consider the protocol

$$\begin{array}{ll} 1. & A \to B : N_A \\ 2. & B \to C : \{N_A\}_K \\ 3. & C \to A : \{N_A, N_A\}_K \end{array}$$

A type-flaw attack upon this protocol involves sending the composed term N_A, N_A where B expects a nonce:

$$\begin{array}{ll}
1. & A \to B\times : N_A \\
1. & I_A \to B : N_A, N_A \\
2. & B \to C\times : \{N_A, N_A\}_K \\
3. & I_C \to A : \{N_A, N_A\}_K
\end{array}$$

Even though the impersonate rules are not complete in the sense of Dolev and Yao, the problem is not severe:

- The impersonate rules are only provided as a first approximation; the CL and the OFMC back-ends both started with these impersonate rules as a simple and effective basis, but later replaced them with the more efficient lazy intruder model, which is complete for Dolev-Yao.
- As a first approximation, the impersonate rules serve well, since all attacks excluded by this simplification are very 'artificial' attacks in the sense that they are unlikely to work in real implementations (e.g. none of these attacks works in practice if the implementation uses fixed bit-widths for all the items).
- The only back-end that currently relies on the impersonate rules is SATMC. However, in this case, the impersonate rules are no restriction, since SATMC is based on the typed model of messages, and for the typed model the impersonate rules are complete; this is because in the typed model an agent would never accept composed messages in the place of message parts that should contain atomic messages (of a certain type).

3.3.4 Goal model

As goals we have state-based safety properties, i.e. the goals determine which states are to be considered as a flaw — the attack the intruder is searching for. Recall that we have four kinds of goals that can be specified in HLPSL:

- *Correspondence.* Correspondence between two roles means that for every pair of agents instantiating these roles, if one of the agents has finished one run of the protocol, he can be sure the other agent has participated in that run.
- *Secrecy.* The intruder may never find out an item that was declared secret, unless it is generated in a session in which the intruder participates officially (i.e. under his real name).
- *Short-term secrecy.* Like secrecy, but additionally, when a run of the protocol is over, then any short-term secret generated during the run is no longer regarded as a secret and is given to the intruder. This is used to model that an intruder may find out and reuse short-term secrets after the respective session is over, e.g. by an off-line cryptographic attack.
- *Authenticate.* `A authenticate B on M` means that when an agent a in role A has finished the protocol, believing he executed a session with an agent b in role B who sent m as a value for M, then b actually sent m as M to a in the respective roles; moreover if a accepts the same value m several times, then b must have said it at least that often (i.e. the intruder didn't replay old messages to make a re-accept m).

$$\begin{array}{rcl}
\textit{GoalState} & ::= & \textit{Correspondence} \mid \textit{Secrecy} \mid \textit{STSecrecy} \mid \textit{Authenticate}\ \mathsf{EOL} \\
\textit{GoalFact} & ::= & \textit{Secret} \mid \textit{Give} \mid \textit{Witness} \mid \textit{Request} \\
\textit{SimplificationRule} & ::= & \textit{f\_simplif} \mid \textit{matching\_request} \mid \textit{no\_auth\_intruder}
\end{array}$$

Recall that *GoalFact* is a non-terminal symbol used above in the message rules for honest agents to specify goal relevant facts; for example, *Secret* is used to express which of the messages are considered to be a secret.

Correspondence. Failure of correspondence can be expressed as a state where one agent in a session `xc` has executed a whole run of the protocol, i.e. is now in the initial state of session `s(xc)`, while the other agent is still in the initial state of session `xc`.

Correspondence	::=	*Principal*.*Principal*

For instance, the HLPSL2IF translator maps the correspondence goal of the NSPK example into the following two IF goal states:

```
w(0,_,mr(a),_,_,_,s(xc)).w(1,mr(a),mr(b),_,_,_,xc)
w(0,_,mr(a),_,_,_,xc).w(1,mr(a),mr(b),_,_,_,s(xc))
```

The first goal describes a state in which `a` has completed one run of a session (and is now prepared to start a new session `s(xc)`), while `b` hasn't noticed anything (and is expecting the first message of session `xc`). The second goal describes the inverse situation, i.e. `b` has executed one session without any participation from `a`.

Secrecy. To keep track of which items in an actual protocol run are secret, we issue facts that declare secrecy, either in the initial state (if the secret is a fixed constant) or in the RHS of agent rules (if the secret is a fresh constant). Since the intruder may play as a normal agent in a session, he may legally find out the secrets associated with this session. Therefore secrets are always associated with a particular session number, i.e. secret facts are of the form

Secret	::=	`secret`(*Message*,`f`(*Session*))

The symbol `f` has a special meaning declared by the static IF rule

f_simplif	::=	`f(s(xc))` EOL `=>` EOL `f(xc)`

i.e. a secret is independent of the number of the runs that have been performed in a session.

A secret fact appears on the RHS of all message rules whenever, during this step, the agent creates a new item that is declared to be a secret in the goal specification of HLPSL. The goal is:

`secret(xsecret,f(`i`)).i(xsecret)`

for every session i the intruder is not an official participant of. Therefore the grammar is:

Secrecy	::=	`secret(xsecret,f(`*Session*`)).i(xsecret)`

Short-term secrecy. A short-term secret is given to the intruder once the run of the session, in which it was created, is over. Therefore, in the rule where an honest agent receives and accepts the final message of the protocol, he issues a *Give* fact that declares, intuitively, that the secret is no longer needed to be kept secret. *Give* facts have the form

```
Give ::= give(Message,f(Session))
```

When a secret is released by the give fact, the secret fact is removed from the state (so there can not be violations of the goals) and the secret message is added to the intruder knowledge.

```
STSecrecy ::= give(xsecret,f(xc)).secret(xsecret,f(xc)) EOL
              => EOL
              i(xsecret)
```

Authenticate. For authenticate goals, we use the following facts:

```
Witness ::= witness(Agent,Agent,Constant,Message)
Request ::= request(Agent,Agent,Constant,Message)
```

Witness facts express that an honest agent actually sent a particular item for a certain purpose to another honest agent. Request facts express that a corresponding witness fact is required.

For instance, `witness(mr(a),mr(b),Na,nonce(c(Na,7)))` means that `a` has created the nonce `c(Na,7)` as item `Na` of the protocol for communication with `b`.

The fact `request(mr(b),mr(a),Na,nonce(c(Na,7)))` requests the existence of this witness fact (note the inverse position of the agents).

If we have a goal `B authenticate A on Na`, then, in the message rule where `A` first sends `Na`, on the RHS a witness fact is issued, and in the step where `B` receives his final message (i.e. after which `B` accepts the run), the RHS is extended with an appropriate request fact.

Again, there are two static rules that handle the witness and request facts:

```
 matching_request ::= witness(xa,xb,xitem,xmsg).request(xb,xa,xitem,xmsg)=>
no_auth_intruder  ::= request(xb,mr(I),xitem,xmsg)=>
```

Note that both of these rules are not to be read as state descriptions, but as transition rules with an empty RHS (i.e. when the LHS matches, these facts are removed). The first rule describes that, if the state contains a matching pair of a witness and a request term, then they must be removed. The second rule means that the intruder never needs to be authenticated, i.e. that when he officially plays under his real name one role of the protocol, everything he says is really from him.

Note that these two rules are considered to be normalization rules, i.e. whenever they are applicable, they must be applied. To put it another way, the goal check may only be applied to a state that is normalized with respect to the two rules.

The goal, for authentication, is the existence of a request fact in the state (that has not been removed by the normalization rules), hence a request that wasn't satisfied:

Authenticate ::= `request(x1,x2,x3,x4)`

As an example, consider again the NSPK protocol, but this time with the goal

`A authenticate B on Nb.`

According to the protocol, the nonce `Nb` is created by `B` before sending the second message of the protocol. Hence the message rule that describes `B` sending the message is augmented with a witness fact:

```
...
=>
...m(2,xb,xb,xa,crypt(xka,c(xNa,nonce(c(Nb,xTime))))).
witness(xb,xa,Nb,nonce(c(Nb,xTime)))
```

Here, the value sent by `B` as `Nb` is `nonce(c(Nb,xTime))`.

As the counterpart, role `A` requests that the received nonce `Nb` is genuine, when she accepts the protocol run, i.e. after sending step 3 of the protocol.

```
...m(2,xr,xb,xa,crypt(xka,c(xNa,xNb)),xc2)
=>
...m(3,xa,xa,xb,crypt(xkb,xNb),xc2).request(xa,xb,Nb,xNb)
```

4. IF optimizations

The IF specification of a protocol is composed of rules describing the behavior of the honest agents and that of the intruder. The rules describing the behavior of the honest agents permit to run the protocol in the "classical" way, while the rules for the intruder describe his possibility to read messages (eavesdropping), intercept messages (divert), or send messages under any principal's name (impersonate).

In this section, we present two important optimizations of this model, which allow us to drastically increase the efficiency of our tool, namely the *lazy intruder* model and the *step-compression* method. The lazy intruder model is used by both the CL and the OFMC back-ends, while the step-compression methods is realized by all three back-ends.

4.1 The lazy intruder

4.1.1 The Dolev-Yao intruder model

In the classical intruder model of Dolev and Yao [21], the intruder can always analyze the messages in his knowledge, and from this knowledge compose messages using all the classical constructors (pairing, encrypting,...). The main problem with this general model is that it does not terminate. For instance, from a knowledge a, the intruder can compose a, a, then $(a, a), a$, and so on.

To handle this problem, the HLPSL2IF translator represents the impersonate behavior by a finite set of rules in the IF. These impersonate rules can be very efficient since all the messages they permit to compose are acceptable by honest principals, but we want to take this idea further.

4.1.2 Problems with the Dolev-Yao intruder model

The generated impersonate rules transform a possibly infinite model into a finite one. But they are not complete in the sense of Dolev-Yao: the composition of messages follows the structure of the awaited messages in all their details.

This property is too strong: the intruder cannot use whatever he wants for completing a message, even if this part of message will never be analyzed by principals receiving the message. In other words, a disadvantage of this way of composing messages is that it rules out some of the possible type flaws (as it was explained in Section 3.3.3).

The impersonate rules suffer from another serious problem: there may be many of them, since they have to describe all ways to compose the messages. In addition, for one rule, there may be many ways to apply it. For instance, if one nonce is needed and the intruder has 5 in his knowledge, all the possibilities will have to be considered. And since this is the same for each piece of knowledge that is used, the combination of all these choices may give a dramatically large number of possible applications.

4.1.3 The lazy intruder model

The drawback of using the impersonate rules produced by the HLPSL2IF translator is clear. One has to do a lot of computing to search the entire space, while most of the states reached are equivalent in terms of finding a flaw. This problem seems to be unavoidable, since a part of a message a principal does not know at step i may be used later at step i'. Thus, it cannot simply be discarded.

This is illustrated by the following example protocol:

$$\begin{array}{l} 1.\ A \rightarrow B : \{N_A, B\}_{K_B} \\ 2.\ B \rightarrow A : N_B \\ 3.\ A \rightarrow B : \{N_A, N_B\}_{K_B} \end{array}$$

Since B does not know N_A, he may accept any value for it when he receives the first message, and when he receives the third message, he will simply check that the same value appears again for N_A. This however implies that in the first message the intruder can send for N_A anything he can produce from his knowledge. This leads to a combinatorial explosion of the states.

This explosion, generated by our protocol-dependent intruder model given by the impersonate rules of the IF, can be avoided by replacing this model with a protocol-independent symbolic intruder model, which we call the "lazy intruder". This new model has two advantages over the impersonate rules: first, it reduces the branching of the search tree by using a symbolic representation of the messages the intruder can generate, and second, it is complete in the sense of Dolev-Yao.

The basic idea is the following: if an agent expects to receive a message, but does not expect a particular value in some parts of that message (e.g. when he wants to receive a new nonce from another agent), then for the receivability of that message it is irrelevant which concrete value the intruder chooses for this part when constructing the message.

Consider again the previous example. The receiver of the first message is able to perform the decryption, but then only expects a particular value in the position of the un-encrypted agent names; the value of N_A is completely irrelevant for the receiver, i.e. the intruder may send any message of the form

$$1.\ A \rightarrow B : \{x_M, B\}_{K_B}$$

where x_M is an arbitrary message term constructed from his knowledge.

The idea is that the search procedure postpones the decision which message term should be sent in the places of the variables. In other words, the search procedure leaves the variables uninstantiated and only keeps track of which items were known by the intruder when he generated this message. We achieve this using constraints of the form

$$\textit{intruder-generated-terms}\ \textit{from}\ \textit{intruder-knowledge}\ .$$

The intruder can now send a message totally independent from the protocol to any agent b that is expecting a message from another agent a:

$$I_a \rightarrow b : x_M \quad \text{with the new constraint } x_M\ \textit{from}\ IK$$

where IK is the list of items known by the intruder in the current state.

Note that here we start with a completely undetermined message x_M, i.e. we don't take into account yet, what the recipient expects to receive; the lazy intruder subsumes a standard heuristic used in the impersonate rules that we only have to consider receivable messages, since it will be checked later, if there is a match between the messages the intruder can generate and the messages that the honest agents can receive (in other words there is no need for the intruder to try sending a message that would not be acceptable by the receiver).

Generability check

To let an honest agent perform a regular protocol step, the left-hand side of the respective step-rule must be unified against a subset of the facts in the current state. Note that, since the facts in the state might now contain variables like the x_M above, a pattern match between state and rule wouldn't be sufficient. The unification results in a most general unifier, i.e. the least substitution for variables in the current state. For instance, assume agent b in the above example expects a message of the form $\{a, x_{Na}\}_{K_b}$ where x_{Na} is any message; then we have the substitution $x_M \mapsto \{a, x_{Na}\}_{K_b}$ which must also be applied to the *from*-constraints:

$$\{a, x_{Na}\}_{K_b} \ \textit{from} \ \textit{IK}$$

Now that the reception of the intruder-generated message has changed the *from* constraint, it is necessary to check if the intruder was able to generate such a message from the denoted *IK*. Before describing how this check is performed in general, let us illustrate it on the above example.

For public key encryption there are two possibilities how the intruder can have generated the message: either he has a message of this form that he recorded earlier but could not analyze further, or he knows the public key and constructed the message parts from his knowledge. Following the latter possibility, the *from* constraint is transformed into

$$K_b, a, x_{Na} \ \textit{from} \ \textit{IK} \ .$$

Assuming the intruder knows a and her public key ka (i.e. they are contained in *IK*), then only

$$x_{Na} \ \textit{from} \ \textit{IK}$$

remains, which simply means the nonce x_{Na} is again an arbitrary item the intruder generated from his knowledge.

The evaluation of the *from* constraint stops at the point where only variables are left on the left-hand side; this is why we call this intruder model "lazy": all constraints are only evaluated as far as this is demanded by the expectations of the agents.

Let us now formalize the backward analysis that checks whether there exists a message that can be unified with the intruder-generated one, i.e. that can be composed by the intruder from his knowledge. This analysis amounts to the partial resolution of the following constraints system, expressed in a very general way, where $T \cdot \textsc{compose}(t)$ *from* $\textsc{Know}(I) : \mathcal{E}$ represents a list of constraints where $\mathcal{E}$ is the tail of the list and $T \cdot \textsc{compose}(t)$ *from* $\textsc{Know}(I)$ is the head of the list. The expression $T \cdot \textsc{compose}(t)$ is a list of $\textsc{compose}(\tau)$ expressions meaning that the term τ has to be composed from the knowledge I of the intruder. The last element of this list is $\textsc{compose}(t)$.

For sake of conciseness, we denote by $\textsc{Apply}(t_1, t_2)$ the result of the creation of a new term from t_1 and t_2 by a binary operation (such as an encryption).

$$\begin{array}{ll}
(\mathcal{C}_{unif}) & T \cdot \textsc{compose}(t) \ \textit{from} \ \textsc{Know}(s \cup I) : \mathcal{E} \rightarrow \\
& T\sigma \ \textit{from} \ \textsc{Know}((s \cup I)\sigma) : \mathcal{E}\sigma \quad (\sigma = mgu(t, s)) \\
(\mathcal{C}_{dec}) & T \cdot \textsc{compose}(\textsc{Apply}(t_1, t_2)) \ \textit{from} \ \textsc{Know}(I) : \mathcal{E} \rightarrow \\
& T \cdot \textsc{compose}(t_1) \cdot \textsc{compose}(t_2) \ \textit{from} \ \textsc{Know}(I) : \mathcal{E} \\
(\mathcal{A}_{pub}) & T \ \textit{from} \ \textsc{Know}((\{t_1\}^{pub} t_2) \cup I) : \mathcal{E} \rightarrow \\
& \textsc{compose}(\textsc{inv}(t_2)) \ \textit{from} \ \textsc{Know}((\{t_1\}^{pub} t_2) \cup I) \\
& : T \ \textit{from} \ \textsc{Know}((\{t_1\}^{pub} t_2) \cup (t_1) \cup I) : \mathcal{E} \\
(\mathcal{A}_{sym}) & T \ \textit{from} \ \textsc{Know}((\{t_1\}^{sym} t_2) \cup I) : \mathcal{E} \rightarrow \\
& \textsc{compose}(t_2) \ \textit{from} \ \textsc{Know}((\{t_1\}^{sym} t_2) \cup I) \\
& : T \ \textit{from} \ \textsc{Know}((\{t_1\}^{sym} t_2) \cup (t_1) \cup I) : \mathcal{E} \\
(\mathcal{A}_{pair}) & T \ \textit{from} \ \textsc{Know}((\{t_1, t_2\}) \cup I) : \mathcal{E} \rightarrow \\
& T \ \textit{from} \ \textsc{Know}((t_1) \cup (t_2) \cup (\{t_1, t_2\}) \cup I) : \mathcal{E}
\end{array}$$

These rules have the following meaning:

- $\mathcal{C}_{unif}$: using this rule, the intruder unifies a term he knows with a term he has to compose;
- $\mathcal{C}_{dec}$: it is applied when the intruder, trying to compose a term $\text{APPLY}(t_1, t_2)$, tries first to compose t_1 and t_2;
- $\mathcal{A}_{pub}$, $\mathcal{A}_{sym}$: these rules are applied when the intruder tries to decompose an encrypted term. In order to decompose this cipher, the intruder has to show he can compose the corresponding key from his current knowledge;
- $\mathcal{A}_{pair}$: this rule is applied when the intruder tries to decompose a message built by the concatenation of two terms t_1 and t_2. No assumptions on his knowledge is needed.

The characteristic of this method is that it is not necessary to resolve constraints like

$$\text{COMPOSE}(x) \ \textit{from} \ \text{KNOW}(I)$$

where x is a variable, since this expresses that any value is accepted at this position in the expected message.

The terms may now contain variables. This means that the variables in the expected message may be instantiated later during the reception of a new message, or during the resolution of another constraints system. Therefore, we have to store all the constraints systems resolved so far, and to impose that all these systems must be simplified in order to have a possible sequence of messages.

Analysis of new knowledge

When the intruder overhears an honest agent's message, we must analyze what new items he can learn from that. As atomic analysis steps, the intruder can decompose concatenated messages and decrypt messages if he knows the appropriate keys. These analysis steps are performed until a fixed-point is reached, i.e. no further items can be learned.

This kind of analysis is standard, however things are slightly more complicated in our setting, since all messages might contain variables. One can simply ignore variables that result from an analysis step: since the variable was something the intruder constructed earlier, he already knows that item, whatever it might be.

However, if one wants to check if the intruder has the necessary key for a decryption, we perform again the generability check described above on that item. This is a solution not only for the problem of variables in the message terms but also for handling composed keys. For instance, consider the analysis of the message $\{M\}_{Na,x_{Nb}}$ where Na is a nonce generated by an honest agent, x_{Nb} is an item generated earlier by the intruder, and the pair Na, x_{Nb} is used as a symmetric key. The analysis procedure performs the generability check for the encryption key: Na, x_{Nb} *from* IK where IK is the current intruder knowledge. In this example, the generability check is successful if and only if Na is contained in the current intruder knowledge IK. If the key can be generated, then the deciphered message is added to the intruder knowledge and subjected to analysis, too. If the key cannot be generated, then the generability check must be repeated later whenever the intruder learns new items during the analysis.

Note that calling the generability check can result in new constraints and substitutions. This intuitively means, that the intruder has sent a message x_M in the past which is not yet instantiated, and we now find out that he would be able to analyze some message M now, if he sent a particular value in that past message x_M; if he sent a different message, however, he now would not be able to perform certain analysis steps. This latter possibility can not be excluded, since it might also be interesting to instantiate x_M later for a different purpose.

4.2 Step-compression

We now introduce *step-compression*, a second method that helps to reduce the analysis time of the tools without excluding any attacks, and which is employed by all three back-ends. The step-compression method is independent of the lazy intruder method, but it is well-suited to be combined with it.

The idea behind step-compression is to merge some of the transition rules into one atomic operation to exclude certain interleavings that don't necessarily need to be considered. This is a continuation of an idea that we already employed in the message rules of honest agents in the IF: as explained in Section 3.3.2, when an honest agent has received a message that is conform with the protocol, he can directly send the answer message. It is thus not necessary to consider an interleaving where, for instance, the agent first receives several messages of different protocol runs and then later replies them; he can reply to each of the messages immediately upon reception.

This optimization of letting the agents directly react to the messages they receive is implemented in the HLPSL2IF translator. We could similarly program the HLPSL2IF translator to perform the step-compression method as well and to directly generate the merged rules. Alternatively, each of the analysis tools can implement the step-compression method independently. We have chosen not to make step-compression mandatory for the tools, as this allows us to choose if we want to connect it with the lazy intruder or not.

The operations that we want to merge are the transition rules that involve sending or receiving of messages: first the message rules that describe the behavior of the honest agents, second the impersonate rules that describe the messages the intruder could send to the agents, and finally the eavesdrop and divert rules that allow the intruder to read and to intercept messages of honest agents.

We will now first describe the step-compression method using impersonate rules and later show how it can be integrated into the lazy intruder strategy. Note also that for the method to work we must assume that the intruder has the full intruder ability that can be specified in HLPSL, i.e. he can impersonate, eavesdrop, and divert.

First idea. If the intruder sends a message to an honest agent using impersonate, we can let this agent execute the appropriate message rule immediately. Intuitively, the impersonate rules are designed to let the intruder produce a message that can be received by an honest agent, and we want to consider only the cases where this agent actually receives the message and answers.

For example, let us consider the NSPK protocol again. One impersonate rule for the second message of the protocol, $B \rightarrow A : \{N_A, N_B\}_{K_A}$, lets the intruder generate the message from his knowledge.

```
w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],true,xc).   A is waiting for B
i(xka).i(xNa).                                    intruder can compose the message and
i(xNb)                                            he knows a value that can be used as N_B
=>
w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],false,1).   A is still in the same state
i(xka).i(xNa).i(xNb).
m(2,I,xb,xa,crypt(xka,c(xNa,xNb)),0)
```

The corresponding message rule lets A react to that message if it contains the right nonce N_A:

```
h(s(xTime)).w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],xbool,xc).
m(2,xr,xb,xa,crypt(xka,c(xNa,xNb)),xc2)
=>
h(xTime).m(3,xa,xa,xb,crypt(xkb,xNb),xc2).
w(0,xinit,xa,[],[xa,xka,xka',xb,xkb],true,xc)
```

The combination of the two rules works as follows: the RHS of the impersonate rule is regarded as producing an intermediate state. When a fact on the impersonate rule's RHS can be unified with a fact on the message rule's LHS, then both facts can be removed, propagating the substitution in the rest of the rule, which then contains the facts that have not been removed .

```
h(s(xTime)).w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],true,xc).i(xka).i(xNa).i(xNb)
=>
h(xTime).m(3,xa,xa,xb,crypt(xkb,xNb),0).
w(0,xinit,xa,[],[xa,xka,xka',xb,xkb],true,xc).i(xka).i(xNa).i(xNb).
```

Second idea. If the intruder has sent a message using impersonate and the corresponding honest agent has answered using his message rule, then the intruder could directly eavesdrop that message. That is not a restriction since it can only add new items to the intruder knowledge.

In the example above, this yields the additional fact `i(crypt(xkb,xNb))` on the RHS, since the intruder has read the answer message from A.

We have now three transitions merged into one atomic transition: impersonate, message step, and eavesdrop. Note that there are two exceptions: if an agent is in his initial state and wants to send the initial message of the protocol, the intruder doesn't need to send an impersonate message; also, for the final message, the honest agent won't send a reply message, hence the intruder has no new message to add to his intruder knowledge.

Third idea. The step-compression is sufficient to model all the communication in the network: we can see the intruder as a 'moderator' of all communications between agents, i.e. a never talks directly to b, but the intruder diverts the message from a, and after analysis, possibly sends it to b. Since in this way every communication can be simulated, we no longer need the original rules; that is, we can remove all message, impersonate, eaves-dropping, and divert rules.

As a consequence, we can also replace the eavesdropping in the step-compression rule with a divert step, i.e. the intruder takes any answer message from an honest agent directly off the network and into his knowledge; in the step-compression rule this means that the RHS no longer contains a message fact.

The step-compression rule for the above example is thus:

```
h(s(xTime)).w(2,xb,xa,[xNa],[xa,xka,xka',xb,xkb],true,xc).i(xka).i(xNa).i(xNb)
=>
h(xTime).i(crypt(xkb,xNb)).
w(0,xinit,xa,[],[xa,xka,xka',xb,xkb],true,xc).i(xka).i(xNa).i(xNb).
```

Fourth idea. Using step-compression, message facts appear neither in the fact list of any reachable state, nor in any rule: in the initial state there is no message term, and step-compression only changes the local states of honest agents and expands the intruder knowledge. This is helpful for the SAT-based model-checker, for instance, since the encoding of the state space into boolean variables is much smaller.

Integration with the lazy intruder. The combination of the step-compression and the lazy intruder is relatively straightforward. Since the lazy intruder is not expressed in terms of IF rules, we can not perform a merge of rules directly. We rather directly integrate the step-compression into the lazy intruder technique as follows:

1. *Impersonate.* From the current state the intruder selects a w-fact, describing an honest agent willing to communicate. If the corresponding step rule contains a message term in the LHS, the intruder provides such a message by sending some message from his knowledge. In the lazy intruder approach, however, we do not have to decide directly what this message is. Hence we choose to let the intruder send $I_A \to B : x_M$ whenever B is the agent who wants to communicate and is expecting a message from A, and add the constraint x_M *from* IK where IK is the current intruder knowledge.

2. *Step.* Then the step rule is performed by matching LHS and the current state, resulting in a generability check applied to the intruder-generated message. The LHS facts are replaced by the RHS facts of the rule.

3. *Divert.* If the agent sent a reply message (i.e. it wasn't the last message of the protocol), the intruder takes it from the net, adds it to his knowledge and analyses it.

Further details on how the lazy intruder and the step-compression optimization methods are actually realized in the different back-ends are given in the following sections.

5. Output format

Our three back-ends, the OFMC model-checker, the CL theorem-prover and the SAT model-checker, require a common output format, which provides us with an unambiguous and clear way to trace the attacks found by the inference engines. It also simplifies the comparison of results obtained with the different techniques. The back-ends, however, do not handle the same set of debugging information. Therefore, we specify here a flexible output grammar with several optional declarations. We are currently implementing this common syntax and we expect to obtain compatible back-end outputs very soon.

We use throughout this specification the same definitions given in Section 2 for the comment lines and the non-terminals: *ident*, *instance*, *int*, *goal_ description* and *msg*.

The grammar supports reports of multiple flaws.

outputDescription	::=	*descriptionSection**

The main parts of the description are structured as follows:

descriptionSection	::=	*protocolId*
		backEndId$^?$
		statistics$^?$
		goal
		attackTrace

We identify the input protocol by its HLPSL file name:

protocolId	::=	`protocol␣`*ident* ;

We can optionally mention the applied analysis technique:

backEndId	::=	`back_end␣`(`cl` | `ofmc` | `sat`) ;

As statistical data, the back-ends have in common the total execution time. Depending on the precision of back-end, the execution time can either be an integer or a float. Moreover, we allow the addition of data specific to each technique. These data are defined by a customized label and unit:

$$\begin{array}{rcl}
\textit{statistics} & ::= & \texttt{statistics}\textvisiblespace\textit{executionTime}^{?}\ (\textit{additionalStatistic})^{*} \\
\textit{executionTime} & ::= & \texttt{time}:\ \textit{int}(.\textit{int})^{?}\textvisiblespace\texttt{sec}\ ; \\
\textit{additionalStatistic} & ::= & \textit{label}:\ \textit{int}(.\textit{int})^{?}\ (\textvisiblespace\textit{unit})^{?}\ ; \\
\textit{label} & ::= & \textit{ident} \\
\textit{unit} & ::= & \textit{ident}
\end{array}$$

The tool should specify the violated goal from the set of HLPSL goals:

$$\textit{goal}\ ::=\ \texttt{violated\_goal}\textvisiblespace\textit{goal\_description}\,;$$

The following part of the grammar provides us with a trace of the protocol attack. We declare the session field as optional because it is handled by the back-ends in different ways. We mention, at the beginning of the *trace* rule, the receiver's session number. This is widely used in the reports of protocol flaws. Besides, it is possible to provide the session number of both sender and receiver. Obviously, the grammar supports the specification of an impersonated user (if any).

$$\begin{array}{rcl}
\textit{attackTrace} & ::= & \texttt{attack\_trace}\textvisiblespace\textit{trace}^{+} \\
\textit{trace} & ::= & (\textit{session}.)^{?}\ \textit{step}.\ (\textit{session}.)^{?}\ \textit{user}\ \texttt{->}\ (\textit{session}.)^{?}\ \textit{user}:\ \textit{msg} \\
\textit{user} & ::= & \textit{instance}\ |\ \texttt{I}\ |\ \texttt{I}(\textit{instance}) \\
\textit{session} & ::= & \textit{int} \\
\textit{step} & ::= & \textit{int}
\end{array}$$

Here is an example of a concrete output describing the flaw of the NSPK protocol, i.e. the *man-in-the-middle* attack that was first reported in [31]. (The HLPSL specification of this protocol is given in Section 2.)

`%`Attack report.
`protocol` NSPK;
`statistics`
`time` : 0.1 `sec`;
`violated_goal` `correspondence_between` A B;
`attack_trace`

$$\begin{array}{lcll}
1.1.\ a & \rightarrow & I & : \{Na(1), a\}ki \\
2.1.\ I(a) & \rightarrow & b & : \{Na(1), a\}kb \\
2.2.\ b & \rightarrow & I(a) & : \{Na(1), Nb(2)\}ka \\
1.2.\ I & \rightarrow & a & : \{Na(1), Nb(2)\}ka \\
1.3.\ a & \rightarrow & I & : \{Nb(2)\}ki \\
2.3.\ I(a) & \rightarrow & b & : \{Nb(2)\}kb
\end{array}$$

In this attack trace, which consists of two protocol sessions, the intruder I acts as a "man in the middle": he acts as responder in session (1) and as initiator in session (2). In message (1.1), the principal a initiates the session (1) with the intruder I, sending the nonce *Na(1)* encrypted with I's public key. (The problem here is that a thinks that I is an honest principal, and does not know that I will attack the protocol, masquerading as a to a third principal b.) The intruder then starts the session (2) with b by sending the initial message (2.1), using the same nonce *Na(1)*; b replies to this message by sending also his own nonce *Nb(2)*. The intruder is not able to decrypt b's message and cannot reply to b by sending back the nonce *Nb(2)*. Therefore, I sends b's response to a, who will be used as an "oracle". Indeed, a decrypts the message (1.2) and accepts it as response in the session (1) established with the intruder. Assuming that nonce *Nb(2)* was generated by I, a returns it in message (1.3) to the intruder, encrypting it with I's key. I now learns the nonce *Nb(2)* and is thus able to finish session (2) by sending message (2.3). As this message has the correct form, b (wrongly!) believes, at the end of session (2), that he is communicating with a, i.e. that a is the initiator of session (2). Hence, the intruder has succeeded in masquerading as a to b.

We are currently employing the AVISS tool to collect attack descriptions like this one, in order to create a library of protocol flaw descriptions, which will provide the basis for future post-processing, such as enhancing the tool with graphical report generation or attack animations.

6. The back-ends

6.1 The OFMC model-checker

6.1.1 Introduction

The HLPSL2IF-compiler translates the high-level protocol specification language HLPSL into the low-level Intermediate Format IF. The IF describes an infinite-state transition system in terms of an initial state and a transition relation. It further provides a predicate that describes flaw states, i.e. states that manifest a successful attack against the protocol.

In the on-the-fly model checking approach, we consider the infinitely deep tree of states that results from unrolling the transition system provided by the IF, i.e. we take the initial state as the root node and the transition relation as the successor function. This tree is examined using iterative deepening search to find a flaw state.

Note that this is a semi-decision procedure. If a flaw state is found, an appropriate attack is reported to the user according to the output grammar described in Section 5. However, if the tree contains no flaw states and the protocol is correct according to our model, then the search will never terminate.

We use Haskell to implement the search. Haskell allows us to define infinite data structures that are evaluated in a demand-driven, *lazy* fashion. Hence, we can declaratively describe the tree as an infinitely large data structure (of which only a finite portion is evaluated) and heuristics as transformers and filters on that data structure. Moreover, there exist efficient compilers for Haskell that can translate the search routine into a fast binary program.

With respect to the preliminary implementation of the on-the-fly model-checker OFMC described in deliverable D3.1 we have made two major changes:

1. *No separate compilation step.* In the preliminary version of the checker, there was a translator, IF2OFMC, which essentially mapped a given IF-file into Haskell syntax. Afterwards, the resulting Haskell description of the protocol, together with a static Haskell file (describing the protocol-independent part of the model, the search routine and heuristics) was compiled into a search binary by the Glasgow Haskell Compiler (ghc). This construction had a number of drawbacks. First, even if the search time itself often took below one second, our total performance had a ten second compilation time penalty. Second, this construction required us to distribute source code, and the user would have had to install ghc on his computer (which can be quite complicated).

   Combining the translator, the static part of the model, and the search routine allowed us to do without the separate compilation step. The front-end of the search routine is now an IF-parser that extracts the successor-relation for the search tree which is directly searched with no further compilation step. The parsing time itself is linear in the size of the IF and is actually irrelevant, at least for the protocols we have tested so far.

   Hence, we can now distribute an on-the-fly model-checking binary that directly accepts IF files as input, simplifying the original architecture displayed in Figure 1.1 to that of Figure 6.1.

2. *Lazy intruder.* In the preliminary version, we often had to cope with an enormous branching of the search tree induced by the large variety of (meaningful) messages an intruder could send to honest agents according to the impersonate rules. Despite this variety, the impersonate rules provide only a finite approximation of the Dolev-Yao intruder model [21], which involves infinitely many possible intruder messages as explained in Section 3.3.3. As a solution to both reduce the branching and realize the complete Dolev-Yao model, we now use the lazy intruder strategy (replacing the impersonate rules) combined with the step-compression method as described in Section 4.

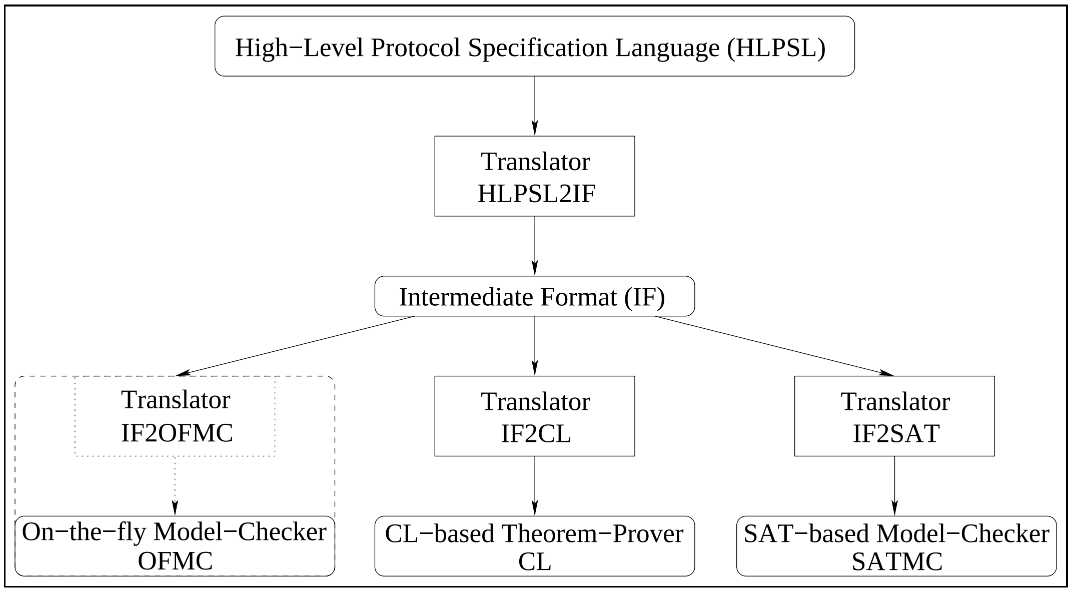

High−Level Protocol Specification Language (HLPSL)
Translator HLPSL2IF
Intermediate Format (IF)
Translator IF2OFMC
Translator IF2CL
Translator IF2SAT
On−the−fly Model−Checker OFMC
CL−based Theorem−Prover CL
SAT−based Model−Checker SATMC

Figure 6.1: Simplified architecture of the AVISS tool: the OFMC accepts IF files as input.

We proceed as follows: in Section 6.1.2 we describe in detail how the information provided in the IF files is stored in appropriate Haskell data types and how the concepts of the IF can be realized in Haskell. In Section 6.1.3 we discuss the integration of the lazy intruder model and the step-compression technique into the OFMC approach, and in Section 6.1.4 we consider the NSPK protocol to give a detailed example of the employed techniques. In Section 6.1.5 we report on our experiments with the on-the-fly model-checker and draw conclusions. A detailed comparison of the relative performance of the three back-ends of the AVISS project is subject of Section 7.

6.1.2 Translation of the IF to Haskell

In the following, we assume familiarity with the IF as defined in Section 3. The IF describes a protocol by an initial state, transition rules, and goal (or attack) states. The first part of the implementation is focussed on defining appropriate data types in Haskell to hold these informations and to write a parser that reads IF and stores the information in these data types. To a large extent, this construction is straightforward, i.e. n-ary function symbols of the IF are represented by n-ary data constructors in Haskell; one might see the Haskell data types we define as an abstract syntax tree of the IF. However there are some subtleties which need special consideration; they are discussed in this section.

Messages

In the IF, an atomic message like a key, a nonce, or a principal name is declared by a string of alphanumeric characters like `kab`. It can be thought of as representing a sequence of bits in real world protocols. A crucial question is then, if (and which) type information should be associated with this message. On the one hand, each message has a type, e.g. it is a public key or an agent name, since one must be able to keep track of the properties that hold for this message, e.g. a message that is encrypted with a public key can be decrypted with the corresponding private key. On the other hand, we want to be able to model type-flaws: the intruder might deliberately (mis)use, for example, a name in place of key. As a consequence, we cannot use the type system of Haskell to model the type of different kinds of messages since this would enforce correct types for all message parts, and hence exclude all type-flaw attacks from the model.

The IF handles this problem by using unary function symbols for declaring the type information, e.g. `sk(kab)` describes a symmetric key with name `kab`. This type information is only used to determine which properties hold for such an atomic message. Rules that describe the behavior of honest agents are *blind* with respect to this type information of items they don't know in advance, e.g. when an agent expects to receive a message of the form `crypt(sk(kab),xNa)`, he will accept any message `xNa` (not necessarily a nonce) encrypted with the key `sk(kab)`. (Recall that variable names start with the letter 'x' in the IF.)

The HLPSL2IF compiler by default produces transition rules in this *untyped* model, but it also has an option to produce a typed version of the rules, e.g. the above example would then be `scrypt(sk(kab),nonce(xNa))`, ensuring that `xNa` is a message of type nonce. The untyped version is more general (i.e. there are many attacks based on type confusion) but this comes at the price of a higher complexity (since more possible messages must be considered).

To encode messages in Haskell we define a recursive data type `Msg` that expresses the algebra of message terms of the IF: it has constructors for each composition method of the IF terms (e.g. concatenation, encryption, and the like) and for the functions that describe the type information (e.g. `Pk`):

```
data Msg = Atom Atomtype
         | Var Vartype
         | Mr Msg | Nonce Msg | Pk Msg | Pk' Msg | Sk Msg
         | Table Msg | Fu Msg
         | Crypt Msg Msg | Scrypt Msg Msg | C Msg Msg
         | Tb Msg Msg | Tb' Msg Msg | Funct Msg Msg
         | Session Int
         | Succsession Msg
```

A number of observations have to be made concerning this data type:

- Every constructor of this data type stands for a function symbol of the IF whose first character is lower-case, except for the constructors `Atom`, `Var`, `Session`, and `Succsession`.
- `Atomtype` and `Vartype` are some types used to represent atoms like `kab` and variables like `xKab`. Both types can be integer to make atomic comparisons faster, however currently we still use strings to represent atoms.
- We need object level variables, since we want to use this data type to represent rules (containing variables for some message parts) and later adopt the lazy intruder technique which introduces variables also in the state.
- `Session` and `Succsession` are used to represent session numbers (the base number and the `s` symbol of the IF). Though session numbers never appear inside messages, we included them in the messages definition, since session terms may contain variables and in this way we can handle them with the same functions (like substitution and unification) which we developed for message terms.
- Some terms are possible in the Haskell data type that may not make much sense, e.g. `pk(sk(ka))`; the OFMC trusts the HLPSL2IF compiler to produce only well-formed terms and rules, i.e. according to the definitions of Section 4.

The translation of IF message terms to Haskell terms of type message is now straightforward, e.g. the term

```
crypt(sk(ka),nonce(xNa))
```

becomes

```
Crypt (Sk (Atom "ka") (Nonce (Variable "xNa"))) .
```

Note that there is a special convention with variable names for public and private keys: if the variable names `xKa` and `xKa'` appear in a rule, then `xKa` must be a public key and `xKa'` the corresponding private key. We replace such terms by `Pk (Variable "xKa")` and `Pk' (Variable "xKa")` in Haskell.

Another complication we had to deal with is the associativity of concatenation: it should hold that `c(c(na,a),b)` is equal to `c(na,c(a,b))`. However, since the lazy intruder technique introduces variables for these terms, we must replace the matching of messages (e.g. between current state and the LHS of a transition rule) with *unification* of messages. This unification problem is very difficult if we assume `c` to be associative (but not commutative). Currently, we use an approximation by associating every concatenation to the right when translating from IF to Haskell data types, e.g. `c(c(a,b),na)` in the IF becomes

```
C (Atom "a") (C (Atom "b") (Atom "na")).
```

During the search, two terms are then unified by simply unifying their sub-terms. This does not solve the problem completely (e.g. `C (Variable "xKa") (Atom "na")` cannot be unified with `C (Atom "a") (C (Atom "b") (Atom "na"))` in this setting), and we are currently working on this.

States

A state is a set of facts. In the IF, there are three kinds of declarations that use states: the initial state, which is a set of ground facts (i.e. no variables appear in any term), the goal states which describe all situations that violate the security goals (if each fact of a goal state can be matched with some fact in the current state), and, finally, states appear as the LHS and RHS of transition rules.

Recall that in the IF there are 8 kinds of facts:

- *MessageFact*, describing a message that was sent but has not yet been received;
- *Principal*, expressing the local state of an honest agent;
- *IntruderKnowledge*, expressing that the intruder has learned a certain message;
- *TimeFact*, giving the current time point;
- *Secret*, declaring that a message is the secret of a certain session;
- *Witness*, expressing that a certain message was sent by an agent to another agent for a certain purpose,
- *Request*, requesting the presence of an appropriate witness fact in the current state.
- *Give*, stating that one run of the protocol is over and the intruder gets the corresponding short-term secret of that run.

Again, the states are translated literally into Haskell, with two exceptions:

- The boolean flag in principal facts is ignored.[1]

[1]The boolean flag in principal facts is only used to store the information whether the intruder has already impersonated a message for this principal. This optimization technique is subsumed by the step-compression.

- We drop the *Time* facts and handle the time statically in our model. (Hence we don't have to bother with matching variables in this kind of facts.)

We hence define appropriate data types `Fact` and `State`:

```
data Fact = M Int Msg Msg Msg Msg Msg
          | W Int Msg Msg Msg Msg Msg
          | I Msg
          | Secret Msg Msg
          | Witness Msg Msg Msg Msg
          | Request Msg Msg Msg Msg
          | Give Msg Msg
type State = [Fact]
```

The first `Int` in `W` facts and `M` facts is the step number. Note that the concepts of the goals employed by the IF need special consideration, as will be discussed later in this Section.

Transition rules

In the IF, a transition rule has the form *LHS*=>*RHS* where *LHS* and *RHS* are states, i.e. sets of facts (represented by lists). The meaning of such a rule is: if each fact on the *LHS* of a rule can be matched against a fact in the current state, then the transition to a successor state is possible where the *LHS* facts are replaced by the *RHS* facts under the substitution induced by the match.

In the preliminary version of OFMC described in deliverable D3.1, we directly translated such rules into Haskell functions, which take the current state as input and return a list of possible successor states. (This list is empty, if there is no match between the rule's LHS and the current state, and it has several elements, if there are several possible matches.) In this way it is possible to construct the whole search tree based on the message rules and impersonate rules.

However, we have now replaced this straightforward mechanism with the more efficient lazy intruder technique. This makes the match between current state and LHS of a rule more complicated; we expand on this below. Note that from the IF files the OFMC parser only needs to read the initial state, the message rules (the behavior of the honest agents), and the goal states; all the other rules, i.e. impersonate rules and static rules, are ignored since they are subsumed by the lazy intruder.

Modeling goals

Correspondence goals. This goal checks if the current state contains principal facts of a certain pattern, e.g. for the NSPK, one of the goals is that `a` has completed one run of a session (and is now prepared to start a new session `s(xc)`), while `b` hasn't noticed anything (and is expecting the first message of session `xc`):

```
w(0,_,mr(a),_,_,_,s(xc)).w(1,mr(a),mr(b),_,_,_,xc)
```

This can be directly translated into a Haskell goal predicate (i.e. a function from states to bool) that checks for a given state, if it contains all facts of the goal.

Secrecy goals. Goals of this kind always have the form `secret(xsecret,f(`*session*`)).i(xsecret)` to express that the intruder may never find out items that are secrets associated with *session*.[2] This mechanism is used to allow the intruder to know the session secrets of sessions where he officially (i.e. under his real name) plays one of the roles. In OFMC, for each secrecy goal we store only the number *session*; this gives us a list of secret sessions. During the search, the on-the-fly model-checker OFMC tests, for every fact `secret(`*Msg*, *Session*`)` in the current state, if both the *Session* is in the list of secret sessions and the message *Msg* is known by the intruder.

[2]The function symbol `f` is a rewrite context with the static rule `f(s(xc))=>f(xc)`, meaning that secrets are invariant over several runs of the same session. We don't need `f` in Haskell as we keep all session numbers for secrecy terms normalized according to this static rule.

Short-term secrets. Short-term secrets, like session keys or nonces, are generated during a session and we want to take into account that such secrets might be compromised after the session is over (for example, due to off-line cryptographic attacks), similar to Paulson's 'Oops-rule' [39]. In the IF, short-term secrets are a special form of a secrecy goal: a short-term secret must be kept secret only as long as the session isn't over; after the session is over, it is given to the intruder. In the IF this is handled as follows: when the short-term secret is generated, the RHS of that rule contains a secret-term declaring that it may not be found out by the intruder if the corresponding session is to be secured, just like a normal secrecy goal. In particular, if the intruder is able to find out the secret at this time, this is considered to be an attack. The rule that describes the last protocol step for an honest agent contains a give-term on the right-hand side; this give-term releases the secrecy fact and lets the intruder gather the secret as described by the static rule

```
give(xsecret,f(xc)).secret(xsecret,f(xc))=>i(xsecret).
```

In the on-the-fly model-checker OFMC, we handle the secrecy facts as in the case of a normal secrecy goal. However, when a give term appears on the RHS of an applied rule, we search for an appropriate secret fact in the current state; if one is found, it is removed and the secret message is added to the intruder knowledge.

Authenticate-goals. As explained in Section 3, the IF issues witness and request facts in the RHS of step rules. A witness fact expresses the fact that an agent created a fresh item, like a nonce or a session key, for a specific purpose. A request fact requires the existence of the corresponding witness fact in the current state. Hence witness facts are handled as normal facts, i.e. they are added to the state as they appear on the RHS of an executed rule. If a request fact appears on the RHS of an applied rule, however, then we check if a matching witness fact is in the present state. If yes, the request is satisfied and the witness fact is removed, otherwise an authentication flaw is found, i.e. an agent believes to have received an authentic item that has been in fact generated, replayed, or otherwise misused by the intruder.

6.1.3 Implementation of the lazy intruder and step-compression

In the preliminary implementation we have used a protocol-dependent intruder model given by the impersonate rules of the IF. We have now replaced this model with a protocol-independent symbolic intruder model, i.e. the lazy intruder described in Section 4. This model has two advantages over the impersonate rules: first, it reduces the branching of the search tree by using a symbolic representation of the messages the intruder can generate, and second, it is complete in the sense of Dolev-Yao [21].

To realize the lazy intruder in the OFMC tool, we have to address the following key issues:

1. *Variables.* Variables can now occur in message facts and principal facts of a state; as noted above, the `Msg` data type defined in Haskell already holds a constructor for variables to handle pattern variables in rules. The preliminary version of OFMC, however, relied on the fact that all states are ground, i.e. contain no variables; this was the case without lazy intruder, since the initial state is always ground and for all rules holds, that the variables on the RHS are a subset of the variables on the LHS (i.e. no rule introduces new variables).

2. *from constraints.* We want to express the fact that a certain message (or list of messages) was created by the intruder using the knowledge at creation time. We call them *from* constraints, since they express that the intruder generated a certain message *from* his current knowledge. They are equivalent to the constraints used in Section 4.1, but for simplicity we omit the unnecessary functions COMPOSE and KNOW. Hence we define the following data type to store these constraints:

```
type Constraint = ([Msg],[Msg])
type State = ([Fact],[Constraint],[Fact])
```

A state now consists of a list of facts (as before), but now additionally contains an (initially empty) list of constraints and a history, which contains the message facts of all messages that have been exchanged so far. The history will later be used for the attack output.

3. *Unification vs. matching.* In the preliminary version of OFMC, we had to match, for a rule application, the LHS facts against the current state. This is no longer possible, since the state may now contain variables, hence one must perform a unification of the LHS and the state. The unification results in a most general unifier, i.e. the least substitution for variables in the current state. For instance, to perform the unification of the intruder-generated message `m(1,I,a,b,xM,0)` in the current state with the LHS of the rule for the first step of the NSPK protocol, containing the term `m(1,_,a,b,crypt(kb,c(a,xNa)),_)`, leads to the unifier `xM` $\mapsto$ `crypt(kb,c(a,xNa))`. The obtained unifier must be applied to all components of the state, including the *from* constraints and the history of messages; hence, if variables get instantiated during the search, this is also performed on the history. When an attack is found and the history still contains variables, the intruder could send any message in the place of these variables. For the OFMC tool, we have implemented first-order unification and substitution as it is standard; note that unification does not assume the binary function symbol `c` to be associative as already discussed above.

4. *Composition and decomposition of intruder knowledge.* Finally, we need means for the generability check and analysis of the intruder knowledge according to the rules given in Section 4; these rules are actually the core of the lazy intruder. However we will use a little variant, splitting composition and decomposition into separate tasks.

Splitting composition and decomposition. In the rule set given in Section 4, there are rules for both composition and decomposition. The rules are applied, whenever the *from* constraints have changed (after a substitution) and it must be checked if the intruder was able to generate the required messages. The analysis is hence a part of the generability check, for instance consider the constraint

$$Na \textit{ from } \{Na\}_{ki}\,, ki' \,.$$

Note that the intruder knowledge is not closed under the analysis rules, namely he can find out Na, but this only checked as soon as the intruder has to generate this message. One could say that, in this setting, even the knowledge analysis is lazy.

However, for the OFMC we wanted to implement the lazy intruder in a different way: the intruder should analyze the new messages he received after each step (this is required anyway if secrecy goals are given) and keep all analyzed items (like Na in the above example) in his knowledge as well. We do this to avoid to perform the same analysis steps again for each state that is reached; only the new items (i.e. the newly learned messages) are analyzed to see if they yield any new information. Hence we split the generability check and the analysis of the intruder knowledge.

Generability check

In the rule set for the lazy intruder in Section 4, there are two rules that describe how the intruder can generate items:

- $\mathcal{C}_{unif}$: He can use an item from his knowledge that can be unified with the item he has to generate.
- $\mathcal{C}_{dec}$: He can try to compose every part of the message from his knowledge, e.g. he can construct $\{M\}_K$, if he can construct both M and K. Note that this holds for all kinds of composed items, including, e.g., concatenation, encryption, hash functions, and key-tables.

In the OFMC tool, we have implemented a function `gencheck: State -> [State]` that takes a state with unevaluated constraints and evaluates them until it is determined whether the intruder was able to construct the required messages or not. For every message on the LHS of a *from* constraint, we check:

- if it is present on the RHS, i.e. the intruder directly knows the message he has to construct, then it can be removed from the LHS;
- if it is a variable, then it currently needs no further evaluation since it is any message the intruder can construct; this is the point why we call the intruder model *lazy*: all constraints are only evaluated as demanded;
- otherwise, i.e. if it is not present on the RHS and is not a variable, then it must be a composed message and we perform the following case split: we unify the message with one of the RHS items ($\mathcal{C}_{unif}$), and we replace the message with its components ($\mathcal{C}_{dec}$). Each case is represented by a new state and on this state we recursively apply the `gencheck` function. Note that the case split is always finite, since the intruder knowledge is finite and messages are only composed from a finite set of sub-messages. Further, the `gencheck` can only finitely recur, since each recursion solves one message the intruder must generate.

When the `gencheck` function terminates, it returns either an empty list, meaning for at least one message on the LHS neither unification nor generation worked and hence the intruder can not construct it, or it returns all possibilities the intruder could have followed to generate the items, and the constraints have only variables on the LHSs.

Analysis of new knowledge

In the rule set in Section 4, there are three rules that describe how the intruder can decompose messages: he can always decompose concatenated messages ($\mathcal{A}_{pair}$), and he can decompose messages for which he knows the decryption key ($\mathcal{A}_{sym}, \mathcal{A}_{pair}$).[3] For other operators like hash functions one might define analysis rules in a similar way. Naturally, atomic messages can not be analyzed further; if a message is a variable, then this means the intruder has analyzed a message and it contained a part that he himself created earlier; hence whatever this message part is, he already knows it, and the variable can therefore be dropped from the intruder knowledge.

In the OFMC implementation, the analysis uses the generability check, i.e. the function `gencheck` discussed above, to see if the intruder can find out the key to decrypt a message. This is necessary to handle composed keys like in $\{M\}_{(Na,Nb)}$ (where one would check if the intruder can find out the key (Na, Nb)). Since the analysis calls `gencheck` and `gencheck` returns a list of possible states, the analysis itself can lead to several possible successor states, hence `analz: State -> [State]`.

The analysis function checks what new messages can be learned by the intruder by trying each of the analysis rules. The analysis is finished, if all possible applications of the rules don't lead to further knowledge, i.e. a fixed-point of the analysis is reached. The analysis will always reach such a fixed-point, since there is only a finite set of messages that can be learned by the intruder.

Still, the analysis is not easy to implement efficiently (i.e. without naively analyzing the same messages again and again) and correct (i.e. ensuring that no possible decomposition is excluded by a search strategy). To achieve this, we use three lists during the analysis:

- *Unanalyzed.* The list of new messages that so far have not been analyzed; at the beginning of the analysis, all messages are in this list, while the other two are empty.
- *Encrypted.* A list of messages that are encrypted and can so far not be decrypted by the intruder.

[3]Note that, since we perform the decomposition separated from the composition, for the check if the intruder can generate necessary keys, we need to consider the *current* intruder knowledge (while in the merged variant, the intruder knowledge is always the knowledge at the *creation time* of the message).

- *Analyzed.* This list contains all messages that are successfully analyzed and can not reveal any further information anymore. After the analysis, this list contains the new intruder knowledge.

The analysis procedure works on these lists as follows. First all messages in the list *Unanalyzed* are considered:

- If there is an atomic message in *Unanalyzed*, then it is transfered to the list *Analyzed*, since atomic messages can not be further analyzed.
- If there is a concatenation of two messages, then it is replaced with its components, according to rule $\mathcal{A}_{pair}$.
- If there is an encrypted message, it is moved into the *Encrypted* list (to check later if the respective key can be generated) and a copy to the *Analyzed* list (since all encrypted messages are kept in the intruder knowledge, even if the intruder can decompose them), according to rule $\mathcal{A}_{sym}$ and $\mathcal{A}_{pub}$.

Once the *Unanalyzed* list is empty, the intruder starts to check what messages in the *Encrypted* list he can decrypt to find out new messages, using the knowledge currently held in *Analyzed.* For every encrypted message, a generate check for the respective key is performed, according to rule $\mathcal{A}_{sym}$ and $\mathcal{A}_{pub}$. If this is successful, then the intruder can decrypt the message, hence it is removed from the list (there is still a copy in *Analyzed*), and its content is added to the list *Unanalyzed.* If the generate check was not successful, then the intruder can not (yet) decrypt the message and hence it is kept in the *Encrypted* list, and the analysis carries on with the next encrypted message.

The analysis stops, if *Unanalyzed* is empty and none of the *Encrypted* messages can be decrypted, since the analysis has reached a fixed point.

Step-compression

In Section 4 we have described a technique to reduce many interleavings by merging three individual transition rules into one atomic step, the *step-compression.* We can integrate this technique directly into the on-the-fly search easily, if we perform the merge of transition rules described in that Section. However this merge is based on the impersonate rules and we want to employ the lazy intruder technique instead of the impersonate rules. It was already noted, that the lazy intruder can be combined with the step-compression method, and we describe in this paragraph, how this can be done in the OFMC approach.

We define a successor function that takes a state as input, and outputs all states that can be reached using the three atomic operations that the step-compression technique merges: the intruder sending a message to an honest agent, the honest agent sending a reply message, and the intruder analyzing this message.

We can see the operations integrated in this successor function as a sequence of atomic operations producing intermediate states. However some of these operations may return several possible intermediate states, like the `gencheck` operation. Hence every piece of the successor function has type `State -> [State]`. All subsequent operations are applied to each of the states that result from an operation. The final result is the union of all possibilities.

Here we summarize the sequence of operations that are performed in the successor function:

- From the present state, the intruder selects one principal term, representing an honest agent willing to communicate, and looks up the message rule that describes how the agent can do his next transition. This rule can be identified by checking for every message rule, if it contains a principal term on the LHS that can be matched with the selected principal term in the current state. The substitution induced by this match is performed on both the LHS and the RHS of the rule.

- If the message rule contains a message fact on the LHS, i.e. the agent expects to receive a certain message before he can do the next step, then the intruder must provide such a message. Let `mr(b)` be the agent waiting for a message of step `i`, and let the expected sender be `mr(a)`. Then the intruder sends the message `m(i,mr(I),mr(a),mr(b),xM,0)` where `xM` is a fresh variable and we add the constraint `xM` ***from*** `IK` where `IK` is the current intruder knowledge. Note that we do not need to consider at this point, what `mr(b)` expects `xM` to be; this is handled by the lazy intruder later.

  If the message rule contains no message fact on the LHS, then the agent is the initiator of the protocol, hence the intruder doesn't need to send a message.

- If an intruder generated message `xM` was sent, we must unify it with the message in the message fact of the LHS of the rule. This leads to a substitution of the current state and possibly to new *from* constraints.

- Hence we must now perform a generability check to see if the intruder can satisfy the expectations of the agent he talks to.

- The message rule can now be applied by removing the select principal term and the message fact generated by the intruder (if present) and replacing it with the RHS of the rule.

- If the RHS of the rule contains a message fact, i.e. the honest agent sent a reply message, this fact is removed from the state and the message is added to the intruder knowledge.

- If the intruder knowledge has grown during the last operation, a new analysis is performed.

6.1.4 A larger example

We would now like to demonstrate the techniques we have just introduced, the lazy intruder and the step-compression, at hand of a larger example. We choose the NSPK protocol with Lowe's fix, also known as the NSL protocol. In [31], Lowe described for the first time the man-in-the-middle attack on the NPSK protocol (cf. Section 5) and suggested to add the name B into the second message to prevent the attack. The NSL protocol thus is:

$$\begin{array}{l} 1.\ A \to B : \{A, Na\}_{Kb} \\ 2.\ B \to A : \{Na, Nb, B\}_{Ka} \\ 3.\ A \to B : \{Nb\}_{Kb} \end{array}$$

This protocol is correct in a typed model, as is shown in [31,39]. However, there is a type-flaw attack on the protocol, which we are now going to illustrate.

We assume that there is an agent b who wants to play as B with a as A, and that there is an agent a who wants to play as B with the intruder I as A; there may be other principals, too, but these are the ones required to perform the attack. The initial intruder knowledge $IK_0 = \{a, ka, b, kb, I, ki, ki'\}$ contains the agent names, their public keys and the intruder's key-pair.

First step. The intruder chooses to first send a message to b, posing to be a; note we do not determine yet what that message precisely is, but just use a variable x_1 and the information that it is a message constructed from the current intruder knowledge. The message and the *from* constraint are:

$$\begin{array}{l} 1.\ I(a) \to b : x_1 \\ x_1 \textit{ from } IK_0 \end{array}$$

Since we use step-compression, we must now let b perform his step. That implies that the message sent by the intruder is matched against b's expectations; he expects $\{a, xNa\}_{kb}$, changing both the sent message term and the *from* constraint:

$$\begin{array}{l} 1.\ I(a) \to b : \{a, xNa\}_{kb} \\ \{a, xNa\}_{kb} \textit{ from } IK_0 \end{array}$$

Before b can send the answer message, it must be checked if that constraint is still satisfiable, i.e. if the intruder could generate the message. Since the intruder has no composed messages in his knowledge, he can only have composed the items:

$$a, xNa, kb \text{ \textit{from} } IK_0$$

Since a and kb are directly in his knowledge, only xNa remains. Recall that the evaluation stops whenever only variables remain on the LHS of a *from* constraint; in this case it simply means that xNa is some message constructed from the knowledge IK_0. b will now answer by generating a new nonce, say nb:

$$\begin{array}{l} 1.\ I(a) \to b : \{a, xNa\}_{kb} \\ 2.\ b \to I(a) : \{xNa, nb, b\}_{ka} \\ xNa \text{ \textit{from} } IK_0 \end{array}$$

The intruder diverts this message (hence $I(a)$ as receiver) and adds it to his knowledge, unable to analyze it further:

$$IK_1 = IK_0 \cup \{\ \ \{xNa, nb, b\}_{ka}\ \ \}$$

Note that all the events that happened so far are part of only one step according to step-compression technique.

Second Step. The intruder now chooses to talk to a, this time under his real name:

$$\begin{array}{l} 1.\ I(a) \to b : \{a, xNa\}_{kb} \\ 2.\ b \to I(a) : \{xNa, nb, b\}_{ka} \\ 1.\text{'}\ I \to a : x_2 \\ xNa \text{ \textit{from} } IK_0 \\ x_2 \text{ \textit{from} } IK_1 \end{array}$$

The message that a expects in the place of x_2 is $\{I, xNi\}_{ka}$:

$$\begin{array}{l} 1.\ I(a) \to b : \{a, xNa\}_{kb} \\ 2.\ b \to I(a) : \{xNa, nb, b\}_{ka} \\ 1.\text{'}\ I \to a : \{I, xNi\}_{ka} \\ xNa \text{ \textit{from} } IK_0 \\ \{I, xNi\}_{ka} \text{ \textit{from} } IK_1 \end{array}$$

There are two ways to satisfy the new *from* constraint: either the intruder again composes this message from his knowledge, or he replays the message from b he just recorded. The latter possibility is the most interesting one, as it leads to the unification

$$\{xNa, nb, b\}_{ka} = \{I, xNi\}_{ka}$$

with the solution $xNa \mapsto I$ and $xNi \mapsto (nb, b)$.[4]

This does not only solve the current constraint, but also changes the older constraint on xNa and the messages in which xNa was sent. In particular, we now know that the intruder must have sent his own name I as the nonce xNa in the first message in order to obtain a message from b that he can use with a in the desired way:

$$\begin{array}{l} 1.\ I(a) \to b : \{a, I\}_{kb} \\ 2.\ b \to I(a) : \{I, nb, b\}_{ka} \\ 1.\text{'}\ I \to a : \{I, nb, b\}_{ka} \\ I \text{ \textit{from} } IK_0 \end{array}$$

[4]Recall that the messages are internally represented as `c(xNa,c(nb,b))` and `c(I,xNi)`, and that `c` was not assumed to be associative.

The remaining *from* constraint is trivial. Now a sends the reply message to the intruder, containing his "nonce" (nb, b) and a new nonce created by a, which we call na:

1. $I(a) \to b : \{a, I\}_{kb}$
2. $b \to I(a) : \{I, nb, b\}_{ka}$

1.' $I \to a : \{I, nb, b\}_{ka}$
2.' $a \to I : \{nb, b, na, a\}_{ki}$

Since the message is encrypted for the intruder, he can find out all items in the message, in particular nb. If the goal is the secrecy of the nonces, then this is already an attack (since nb was a nonce created by b for communication with a, so the intruder may not find it out). If the goal is either correspondence between A and B, or authentication on the nonces, then a further step must be performed to attack the protocol: the intruder sends a message to b, who now expects his nonce nb back. Since the intruder has found it out, he can easily satisfy this constraint:

1. $I(a) \to b : \{a, I\}_{kb}$
2. $b \to I(a) : \{I, nb, b\}_{ka}$
1. $I \to a : \{I, nb, b\}_{ka}$
2. $a \to I : \{nb, b, na, a\}_{ki}$
3. $I(a) \to b : \{nb\}_{kb}$

6.1.5 Experiments with the on-the-fly model-checker

The Clark/Jacob library. The OFMC tool performs extremely well on the protocols of the Clark/Jacob library: it can find a flaw in all protocols that are known to be flawed (except for the ones that can not be modeled in HLPSL) in a total analysis time of less than one minute. Here, we briefly discuss the experimental results related to the OFMC tool, and point the reader to Section 7 for a more detailed comparison and discussion of the running times of the three back-ends of the AVISS tool.

Impact of the new techniques. The integration of the two optimizations discussed above, the lazy intruder and the step-compression technique, have improved the OFMC tool significantly. As compared to the first implementation of the OFMC (described in Deliverable D3.1), these techniques have drastically reduced the search times: for a number protocols including the Kao-Chow protocols, the search for a flaw took a few minutes in the first version of the tool, while in the current version all these examples require less than one second of search time.

During the experimentation phase, we made the following observation. As to be expected, the verification time rapidly grows with the depth of the attack, i.e. in the smallest number of messages that have to be exchanged to give rise to an attack. This depth is the ply in the tree up to which we have to search. The step-compression method described in Section 6.1.3 has lead to a major improvement here: since we regard a sequence of applications of an impersonate rule, a message rule, and a divert rule as an atomic operation, the depth of an attack in the search tree is often drastically reduced as compared with the preliminary implementation of the OFMC tool. For instance Lowe's attack on NSPK consists of 10 rule applications, but only of 4 applications of the merged rules according to the step-compression method; hence in the preliminary version the search tree had to be explored up to ply 10, while in the new approach the search covers only 4 plies.

To summarize, the lazy intruder technique has reduced the branching of the tree, while, intuitively, we could say that the step-compression "reduces the depth" by merging layers of the tree. Hence, the lazy intruder might be considered as a horizontal compression of the search tree and the step-compression as a vertical compression.

The typed version. We have studied protocols both using the default untyped model and using the typed model. Some protocols turned out to suffer both from a type-flaw attack and from an attack that doesn't rely on type confusion, for instance the Andrew Secure RPC protocol displayed in the Table 7.1. In all considered cases, the type-flaw attack is simpler, i.e. it requires less steps. Hence the OFMC tool finds the type-flaw attack first when using the untyped model. Using the restricted typed model, however, excludes the type-flaw attack, so the search procedure will, typically after a considerably longer analysis time, reach the other attack that works without type confusion.

Since the typed model is a restriction of the default untyped model, we wanted to see how much this restriction improves the running times. Therefore we compared both variants for protocols which have a flaw that doesn't rely on type confusion, and hence can be found in both models. To our surprise, the analysis times were almost identical (note that in the first preliminary version of OFMC, the typed model had a significant impact on the performance). We have analyzed some examples in detail to find the reason why there is no substantial difference. The result of this analysis is that protocols that don't suffer from type flaws are either not vulnerable for any type confusions at all (e.g. by ensuring that all messages have a unique form and can not be misinterpreted as a different message of the protocol), or the type confusion can not be exploited for an attack. Recall that the lazy intruder technique uses variables to represent the parts of messages that are not determined by the expectations of the receiver. During the search, a variable can only be instantiated with a message term if some agent can actually receive the resulting message in which the variable appeared. Hence the search explores the different possibilities of the intruder to design a submessage "only as far as this makes sense". Since the protocols considered either didn't allow type flaw attacks or didn't allow to exploit type confusion for attacks, the search tree for the untyped model contains inessentially more nodes than the restricted tree of the typed model.

Since the untyped model is more general and this generality comes at no extra cost, we suggest to run the OFMC tool in the untyped model by default. However, in some cases the typed model can still be helpful, e.g. if the user wants to analyze a protocol for which a real implementation prevents the type flaws while the abstract protocol model in HLPSL/IF does not; in this case one might be interested in the question if the protocol has any flaws *besides* type-flaws.

Outlook. The application of the optimization techniques lead to such an enormous compression of the search tree, that it is possible for all flawed protocols in the Clark/Jacob library to let the OFMC tool perform a complete search of the corresponding compressed tree (up to the ply where the attack is found); it was not necessary to apply any further search heuristics.

We hence see a great potential for improving the OFMC tool even further by applying heuristics. For example, a simple evaluation function could be: 'has the intruder learned anything new through this step, and how interesting is what he learned?' Note also that the partial-order reduction obtained by applying the step-compression method is only light-weight. We plan to develop a formal concept for 'equivalent' states and further reduce the state space that way.

We expect that such further improvements will allow us not only to avoid specifying session instances in the HLPSL description of a protocol and simply let the OFMC tool explore all the different scenarios, but also to analyze large-scale, industrial e-commerce protocols.

6.2 The CL theorem-prover

We begin our discussion of the CL back-end developed by the Nancy group by introducing, in Section 6.2.1, the theorem-prover daTac the back-end is based on. In Section 6.2.2, we then describe how the rewrite rules obtained from HLPSL2IF are transformed into rules suitable for daTac. We conclude this short presentation by giving in Section 6.2.3 a selection of the experimental results obtained using the CL approach. The theorem-prover daTac allowed us to analyze protocols that could not be handled before, to find a flaw on a protocol previously reported as correct, and to detect more than 90% of the flaws previously reported.

6.2.1 The theorem-prover daTac

The theorem-prover daTac, for *Déduction Automatique dans des Théories Associatives et Commutatives*, has been developed by the Nancy group. This software is written in Objective Caml (22000 lines), a language of the ML family.

daTac can be used for refutational theorem proving or directly for deriving new consequences from axioms. It manipulates formulas of first order logic with equality expressed in a clausal form $A_1 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee \ldots \vee B_m$ (written $A_1, \ldots, A_n \Rightarrow B_1, \ldots, B_m$ in the daTac syntax). The deduction techniques implemented are detailed in [45]. We give here only a brief overview.

Deduction techniques

The prover is based on the first order logic with equality, hence the clauses may use the equality predicate. But, we do not need to include the equality axioms. The symmetry, transitivity and functional reflexivity of the equality are built-in by a deduction rule called *Paramodulation*. Its principle is to apply replacements of equals by equals in clauses. A paramodulation step in a positive literal is defined by

$$\frac{L_1 \Rightarrow l = r \vee R_1 \qquad L_2 \Rightarrow A[l'] \vee R_2}{(\ L_1 \wedge L_2 \Rightarrow A[r] \vee R_1 \vee R_2\ )\sigma}$$

where σ is a substitution (a mapping replacing variables by terms) unifying the term l and the subterm l' of A, i.e. $l\sigma$ is equal to $l'\sigma$. This last subterm is replaced by the right-hand side r of the equality $l = r$, and the substitution is applied on the whole deduced clause.

A similar paramodulation rule is defined for replacements in negative literals, i.e. literals on the left-hand side of the $\Rightarrow$ sign.

The reflexivity property of the equality predicate built-in by a rule called *Reflexion*, defined by:

$$\frac{l = r \wedge L \Rightarrow R}{(\ L \Rightarrow R\ )\sigma}$$

where the substitution σ unifies the terms l and r.

The *Resolution* rule allows us to deal with predicate symbols other than the equality:

$$\frac{A_1 \wedge L_1 \Rightarrow R_1 \qquad L_2 \Rightarrow A_2 \vee R_2}{(\ L_1 \wedge L_2 \Rightarrow R_1 \vee R_2\ )\sigma}$$

where σ unifies A_1 and A_2.

Strategies

The prover uses several deduction strategies in order to avoid useless or redundant deductions. These strategies allow us to limit the overall number of deductions explored in the proof search.

The first strategy is an *ordering strategy.* Some specified ordering is applied for comparing terms and for orienting equations. Hence, when a paramodulation step is applied from an equational literal $l = r$, it is checked that a term encompassing l is never replaced by a bigger one.

The ordering is also used for the selection of literals where a replacement is allowed to take place. For instance, it is possible to force each deduction step to apply to the maximal literal of a clause.

Another essential strategy is the *simplification* one. The role of simplification rules is to replace a clause by a simpler one, using term rewriting. Simplification rules are expressed as equalities between two terms, $\Rightarrow u = v$. Such equalities are oriented as rewrite rules using an ordering on terms.

We have also defined *deletion rules.* First, clauses which contain a positive equation $l = l$, or the same atom on the left-hand side and on the right-hand side of the implication sign $\Rightarrow$, are deleted. Another deletion technique is the *subsumption*, which can be described as follows: if a clause $L \Rightarrow R$ belongs to a set of clauses S, then any clause of the form $L\sigma \wedge L' \Rightarrow R\sigma \vee R'$ can be removed from S.

Deduction modulo E

The most specific feature of daTac is the deduction modulo a set of equations E. The motivation for such deduction is the following.

The *commutativity property* of an operator f, i.e. $f(a,b) = f(b,a)$, cannot be oriented as a rewrite rule. This is a major problem when applying paramodulation steps.

Also, the *associativity property* of an operator f, i.e. $f(f(a,b),c) = f(a,f(b,c))$, has the disadvantage to generate infinite sequences of paramodulation steps. For example, from an equation $f(d_1,d_2) = d_3$, we can derive

$$\begin{array}{rcl} f(d_1,f(d_2,e_1)) & = & f(d_3,e_1) \\ f(d_1,f(f(d_2,e_1),e_2)) & = & f(f(d_3,e_1),e_2) \\ & \vdots & \end{array}$$

Moreover, when these two properties (commutativity and associativity) are combined, it becomes very difficult to deduce useful clauses. For example, there are 1680 ways to write the term $f(a_1,f(a_2,f(a_3,f(a_4,a_5))))$, where f is associative and commutative. These 1680 terms are all semantically equivalent but none of them can be omitted, for the completeness of deduction.

So, daTac is designed to run modulo a set of equations E, including commutativity, or associativity and commutativity properties. These equations do not appear explicitly in the set of initial clauses. They are built-in by specific algorithms for equality checking, pattern matching and unification, in conjunction with some specially adapted paramodulation rules.

Advantages of daTac

daTac is an entirely automatic tool which, given a set of clauses, deduces new properties, consequences of these clauses. daTac is refutationally complete: if a set of clauses is incoherent, it will find a contradiction. Its only source of failure in searching for an existing contradiction is the memory limit of the computer.

The implemented techniques rely on ordering and simplification strategies combined with a deduction system based on paramodulation and resolution, as mentioned above, but other important strategies are also available, such as the superposition and basic or constraint strategies [50].

At the end of an execution, some important information is provided by the system. Especially the prover can display a proof of a derived property, or of the contradiction found. Statistics are also available, such as the number of deduction steps, the number of simplification steps, and the number of deletions.

Its deduction scheme makes daTac a very natural — though not the most efficient — choice for analyzing security protocols since these protocols can be interpreted directly as sets of rewrite rules system.

Deduction strategy for protocol verification

We recall that during the translation process, the goals of the intruder are expressed in the IF as states where the goal is violated. For example, consider the secrecy goal. It is translated, in the IF, into a state

$$\mathtt{i}(x_{secret}) \cdot \mathtt{secret}(x_{secret},\ldots)$$

that is, a state where a term x_{secret} is known by the intruder whereas it should have remained secret. Such a critical state is then *negated*, in the daTac input file.

Once transformed into a daTac format, the rules of the Intermediate Format can be listed as follows:

(0) rewrite rules for simplification;

(1) clauses representing transitions rules, including intruder's rules introduced to model the lazy intruder;

(2) the clause representing the initial state, $P\big(t_{init}(\mathcal{P})\big)$;

(3) the critical states $(\neg P\big(t_{goal}(\mathcal{P})\big))$;

In other words, HLPSL2IF translates a goal in HLPSL into a state in IF where this goal is violated. The deduction method used is therefore refutationally complete for finding flaws in security protocols.

The deduction strategy used by daTac for protocol verification is the following, and basically amounts to a breadth first search strategy:

```
Repeat:
  Select a clause $C$ in (2), $C$ contains only a positive literal
  Repeat:
    Select a clause $D$ in (1), $D$ is a clause $L \Rightarrow R$
    Apply Resolution between $D$ and $C$:
      Compute all the most general ac-unifiers
      For each solution $\sigma$,
        Generate the resulting clause $C'\sigma$
    Simplify the generated clauses:
      For each generated clause $C'\sigma$,
        Select a rewrite rule $l \rightarrow r$ in (0)
        For each subterm $s$ in $C'\sigma$,
          If $s$ is an instance $l\phi$ of $l$
          Then Replace $s$ by $r\phi$ in $C'\sigma$
    Add the simplified generated clauses into (2)
    Try Contradiction between the critical state and each new clause:
      If it applies, Exit with message "contradiction found".
  Until no more clause to select in (1)
Until no more clause to select in (2)
```

Note that any derivation of a contradiction with this strategy is a linear derivation from the initial state to the goal and it can be directly interpreted as a scenario for a flaw or an attack.

6.2.2 Translation mechanism

We now describe in more details how rules from the Intermediate Format are translated into a daTac file. The translator we use is a PERL script of about 1500 lines, split into three modules. Each module corresponds to a subset of the rules of the Intermediate Format. This translator directly calls HLPSL2IF. It is therefore possible to use a `-typed` options whenever we want to obtain the typed version of the Intermediate Format. In addition, a small bash script is also available. It allows us to run on all the protocols specifications in some directory in one shot.

The preamble

The preamble of the daTac file must contain the definition of the associative and commutative operators, as well as the definition of an order on the constructors subsequently used. This order does not depend on the protocol specification and is therefore simply included from a file `Order.dat`. The currently declared associative and commutative operators are the dot $\cdot$, which is used to represent multisets of terms, and the `rcrypt` operator, which represents the XOR encryption. We also add, at the end of the generated file, execution parameters used by daTac, which are of no real relevance here.

Literal translation

Although the IF and daTac formalism are very close, there is still some translation to perform before getting rules conforming to the daTac theorem-prover. We began to use the daTac theorem-prover without making any transformation but some syntactic ones. At that point, an IF rule:

$$\begin{array}{l}\mathtt{h}(\mathtt{s}(x_{Time}))\cdot\mathtt{m}(\ldots)\ \cdot\ \mathtt{w}(\ldots)\\ \Rightarrow\\ \mathtt{h}(x_{Time})\cdot\mathtt{m}(\ldots)\ \cdot\ \mathtt{w}(\ldots)\end{array}$$

is transformed into a daTac rule:

$$\begin{array}{l}\mathtt{R}(\mathtt{s}(x_{Time}),\mathtt{m}(\ldots)\ \cdot\ \mathtt{w}(\ldots)\ \cdot\ x_{state})\\ \Rightarrow\\ \mathtt{R}(x_{Time},\mathtt{m}(\ldots)\ \cdot\ \mathtt{w}(\ldots)\ \cdot\ x_{state})\end{array}$$

We use a new x_{state} variable to represent the *context* of rule application. The state is then encapsulated into an atom using the R constructor.

This first translation allows us to use directly the rules of the intermediate format without any interpretation. Finding a flaw in the resulting system is semi-decidable, but this is mostly a theoretical result, since the state space explosion, in this case, prevents us from finding any flaw but on the simplest protocols.

In order to simplify the unification problem, we perform a first transformation. We split the state into sub-states, each containing terms with the same head constructor:

$$\begin{array}{l}\mathtt{R}(\mathtt{s}(x_{Time}),\mathtt{m}(\ldots)\ \cdot\ x_m,\mathtt{w}\ \cdot\ x_w,x_i)\\ \Rightarrow\\ \mathtt{R}(x_{Time},\mathtt{m}(\ldots)\ \cdot\ x_m,\mathtt{w}(\ldots)\ \cdot\ x_w,x_i)\end{array}$$

The last field of R(...) contains the i(...) (intruder knowledge) terms. This transformation results in a small gain in the time needed to find a flaw, but does not address the state space explosion problem.

Using these transformation together with the intruder rules given in the output of HLPSL2IF, it is possible to find flaws only on simple protocols, or on protocols where the first flawed state is rapidly obtained.

Intruder dependent optimization

We have noted, as already described in Section 4.2, that it is possible to reduce the interleaving of rules in the case where the intruder has both the divert and impersonate abilities.

In this case, we can assume that all messages sent by the principals are received by the intruder, and that all messages received by the principals are first sent by the intruder. Under this assumption, it is unnecessary to model the case where a state contains multiple m(...) terms. We can therefore change the principals transition rules by stating that a message can be received if and only if it is the only one already sent and not already received. This remark leads to the step-compression optimization of Section 4.2.

This optimization helps to model a few more protocols. This interleaving reduction is in no way decisive for analyzing security protocols generally speaking. The major explanation for these early failures is that the impersonate rules lead to a state space explosion, and daTac has no specific optimization to handle such state space.

Lazy intruder

We have introduced the lazy intruder constraint system in order to better use the capabilities of daTac theorem-prover (see Section 4.1 for a description of this technique). The constraint system is now the fifth parameter of the R(...) term.

This strategy relies on a control on terms over which rules are applied. The major implementation difficulty is to detect whether a term t is a variable. This is mandatory, since the lazy intruder is essentially a strategy in which a resolution rule is not applied whenever some term is a variable. We now briefly presents how this control over variables was implemented.

There is no construct in daTac which allows us to check whether a given term t is a variable. Therefore, we have adopted the following technique:

First we consider that the argument of `i(_)` is either variable or non-variable terms. Then, we introduce a new symbol, $\mathtt{i}_{cons}$, which also contain terms known by the intruder. We also introduce new simplification rules for simplifying $\mathtt{i}(t)$ into $\mathtt{i}_{cons}(t)$

- for each ground terms t, such as $\mathtt{mr}(a)$ or $\mathtt{pk}(ka)$, that occur in the protocol transition rules
- for each nonce or composed term. For example, the cases of a couple and of fresh symmetric key are handled using the following rules:

$$\begin{aligned} => \mathtt{i}(\mathtt{c}(x_1,x_2)) &= \mathtt{i}_{cons}(\mathtt{c}(x_1,x_2)) \\ => \mathtt{i}(\mathtt{sk}(\mathtt{c}(x_1,x_2))) &= \mathtt{i}_{cons}(\mathtt{sk}(\mathtt{c}(x_1,x_2))) \end{aligned}$$

Every time the intruder's knowledge is augmented by some t, for instance when t is a diverted message, a new term $\mathtt{i}(t)$ is added. Then, when t is decomposed, its two parts generate two $\mathtt{i}(\ldots)$ terms. For example, the following simplification rule handles the decomposition of a pair:

$$=> \mathtt{i}_{cons}(\mathtt{c}(x_1,x_2)) = \mathtt{i}(x_1)\cdot\mathtt{i}(x_2)$$

This simplification system allows us to ensure that after all simplifications have been applied, if a term t is a variable known by the intruder, it appears in its knowledge as a term $\mathtt{i}(t)$, and if t is not a variable, it appears in intruder's knowledge as $\mathtt{i}_{cons}(t)$.

6.2.3 Experimental results

Analysis of an attack

We now consider the Otway-Rees protocol, in the case of which a type flaw leads to an authentication flaw. The HLPSL2IF specification of this well-known protocol is given in Figure 6.2.

This specification can be automatically compiled into a daTac file, on which daTac is applied. Note that we have added an extra message at the end, in which a secret X is sent. This modification was to show that not only does B accept a key that was not sent by A, but moreover the accepted key is known by the intruder.

The trace of an execution is quite hard to analyze if one is not familiar with the deduction techniques implemented in daTac, but luckily, daTac also outputs the sequence of derivations leading to the discovery of the flaw *(in 1.5s)*. In the case of the Otway-Rees protocol, this sequence reads as follows:

```
> Inference steps to generate the empty clause:
    60 = Resol (1,56)       60 = Simpl (11,60)       ...
    60 = Simpl (34,60)      63 = Resol (5,60)        63 = Simpl (11,63)
     ...                    63 = Simpl (30,63)       66 = Resol (44,63)
    66 = Simpl (14,66)      ...                      66 = Simpl (52,66)
        66 = Clausal Simpl({45},66)
```

Now, looking at the given trace reveals the scenario leading to the secrecy flaw. The displayed clauses have been fully simplified.

The first one is fairly simple to interpret, since it corresponds to the first principal sending its first message. All the simplifications following correspond to the decomposition of this message to intruder's knowledge. We thus have:

PROTOCOL Otway_Rees;
IDENTIFIERS
A, B, S : `user`;
Kas, Kbs, Kab : `symmetric_key`;
M, Na, Nb, X : `number`;
KNOWLEDGE
$A : B, S, Kas$;
$B : S, Kbs$;
$S : A, B, Kas, Kbs$;
MESSAGES
1. $A \to B : M, A, B, \{Na, M, A, B\}Kas$
2. $B \to S : M, A, B, \{Na, M, A, B\}Kas, \{Nb, M, A, B\}Kbs$
3. $S \to B : M, \{Na, Kab\}Kas, \{Nb, Kab\}Kbs$
4. $B \to A : M, \{Na, Kab\}Kas$
5. $A \to B : \{X\}Kab$

SESSION_INSTANCES
$[A : a; B : b; S : se; Kas : kas; Kbs : kbs]$;
INTRUDER Divert, Impersonate;
INTRUDER_KNOWLEDGE a;
GOAL Secrecy_Of X ;

Figure 6.2: HLPSL specification of the Otway-Rees protocol

$$a \to \_ : M, a, b, \{Na, M, a, b\}kas$$

The second resolution ($63 = Resol(5, 60)$) is rather unexpected, since it is the reception of the message labelled 4 in the protocol by principal a. Using the protocol specification, it is first read as:

$$\begin{array}{lcll} a & \to & \_ & : M, a, b, \{Na, M, a, b\}kas \\ \_ & \to & a & : M, \{Na, x5\}kas \\ a & \to & \_ & : \{X\}x5 \end{array}$$

At this point, we can only conclude that the intruder has tried to send to the principal a a message *matching* $M, \{Na, x5\}Kas$. He has no choice but to unify ($66 = Resol(44, 63)$) the term yielded after the first message with the pattern required for second step. Now, the sequence of messages becomes:

$$\begin{array}{lcll} a & \to & \_ & : M, a, b, \{Na, M, a, b\}kas \\ \_ & \to & a & : M, \{Na, M, a, b\}kas \\ a & \to & \_ & : \{X\}(M, a, b) \end{array}$$

Thus, the intruder has proved that it can send a term matching the pattern of awaited message, so we can go on to the next step ($66 = Simpl(52, 66)$). But after that, decomposing what it knows, the intruder obtains X, that should have remained secret. The last move (Clausal Simplification) exposes this contradiction, thus ending the analysis of this protocol.

Experiments using XOR encryption

It is also possible to specify XOR encryption in HLPSL. Using this feature, we were able to specify the Gong authentication protocol. Note that, for instance, Lowe was unable to analyze this protocol using the FDR2 model-checker because of state space explosion [22,32]. (See Section 8 for a detailed discussion of the Casper/FDR2 approach.) Using daTac, it was possible to analyze two sequential sessions of the protocol. However, we had to use a simplified model for the XOR encryption.

In the protocol specification given in [18], block-cipher properties were implicit in the third messages, since two vectors are XOR-ed. We thus have chosen to specify this protocol using thrice the XOR encryption operator, once for every couple of elements of the vectors. The resulting HLPSL file is given in Figure 6.3.

PROTOCOL Gong;
IDENTIFIERS
  A, B, S : **user**;
  Kas, Kbs : **symmetric_key**;
  Na, Nb, Ns : **number**;
  $F1, F2, F3, G$: **function**;
KNOWLEDGE
  $A : B, S, Kas, F1, F2, F3, G$;
  $B : A, S, Kbs, F1, F2, F3, G$;
  $S : A, B, Kas, Kbs, F1, F2, F3, G$;
MESSAGES
  1. $A \rightarrow B$: A, B, Na
  2. $B \rightarrow S$: A, B, Na, Nb
  3. $S \rightarrow B$: $Ns, F1(Ns, Nb, A, Kbs) XOR F1(Ns, Na, B, Kas)$,
     $F2(Ns, Nb, A, Kbs) XOR F2(Ns, Na, B, Kas)$,
     $F3(Ns, Nb, A, Kbs) XOR F3(Ns, Na, B, Kas)$,
     $G(F1(Ns, Na, B, Kas), F2(Ns, Na, B, Kas), F3(Ns, Na, B, Kas), Kbs)$
  4. $B \rightarrow A$: $Ns, F3(Ns, Na, B, Kas)$
  5. $A \rightarrow B$: $F2(Ns, Na, B, Kas)$
SESSION_INSTANCES
  $[A : a; B : b; S : se; Kas : Pa; Kbs : Pb; F1 : fk; F2 : fha; F3 : fhb; G : g]$;
INTRUDER Divert, Impersonate;
INTRUDER_KNOWLEDGE $a, b, se, fk, fha, fhb, g$;
GOAL Correspondence_Between AB ;

Figure 6.3: HLPSL specification of the Gong protocol

HLPSL2IF compiles this protocol into a set of rewrite rules. The translation to a daTac input file is done automatically. We have modeled the XOR properties by stating that the `rcrypt` operator, which represents XOR encryption in the intermediate format is an associative and commutative operator, and is nilpotent. We were able to express:

$$x \oplus 0 = 0 \oplus x = x \qquad \text{and} \qquad x \oplus x = 0$$

but our model does not capture the property that knowing $x_1 \oplus x_2$ and $x_1 \oplus x_3$, the intruder can also know $x_2 \oplus x_3$.

We were not able to find any flaws after two sequential executions of a session. In spite of this negative result, this analysis is interesting as it shows that, using a general purpose theorem-prover, it is possible to perform better than highly optimized model-checkers.

A novel flaw on the *Denning-Sacco symmetric protocol*

In all the papers surveyed so far, the *Denning-Sacco symmetric key protocol* was considered to be secure. It was conceived as a correction of the *Needham Schroeder symmetric key protocol*. The sequence of messages, in this protocol, is:

$$\begin{array}{lcll} A & \rightarrow & S & : A, B \\ S & \rightarrow & A & : \{B, Kab, T, \{A, Kab, T\}Kbs\}Kas \\ A & \rightarrow & B & : \{A, Kab, T\}Kbs \end{array}$$

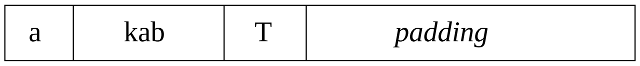
where T is a timestamp. In this protocol, A forwards to B a part of the message sent by the server S. The problem in this protocol is that messages 2 and 3 are of the same global shape. This fact can be exploited in the following sequence of messages:

$$\begin{array}{lcll} I(b) & \rightarrow & s & : b, a \\ s & \rightarrow & I(b) & : \{a, kab, T, \{b, kab, T\}Kas\}kbs \\ I(a) & \rightarrow & b & : \{a, kab, T, \{b, kab, T\}kas\}kbs \ (\equiv \{a, kab, T\}kbs) \end{array}$$

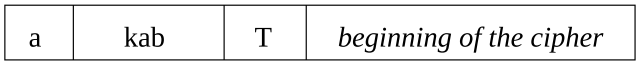
where $I(b)$ means the intruder impersonating the principal b, and a can be any principal.

After this message exchange b accepts a new value for the symmetric key he shares with a, whereas a is not aware that a protocol session took place. This attack relies on a type flaw. In this case, one shall note that it is unlikely that when receiving the whole message b considers $T, \{b, kab, T\}Kas$ to be a timestamp. But the implementation of this protocol may lead to a real flaw in two cases:

1. first, in case there is some padding in the encryption, it is possible that the principal playing the role B just does not check the bits following the part of the message he is waiting for. Since the two ciphers begin with the same data type, there is a real possibility for this principal to accept the message, thus leading to the flaw. This case is illustrated by the following figure:

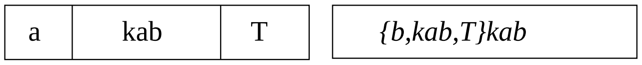


message awaited by b

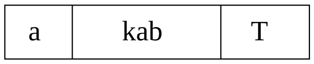


message sent by the intruder

2. second, it may happen that, if a block-cipher algorithm is used (such as DES, for example), the second message is split into smaller parts, from which the intruder will be able to construct a message acceptable by B, no matter how cautious B is. This will be the case, for example, if a block finishes right after T, as is shown in the next figure:



message sent by the server



message awaited by b

We thus believe that this type flaw should not be regarded as an artifact, and that it should be taken into consideration when carrying out an actual implementation of this protocol.

Experiments on the Clark/Jacob library

The main objective of the AVISS project was to automatically detect 70% of the flaws in the protocols that were already reported as flawed in the Clark/Jacob library. As is illustrated in

Section 7, using the CL tool, based on the general purpose theorem-prover daTac, we were able to detect more than 90% of the flaws that were already reported. We were also able to analyze the Gong protocol which Lowe can not handle in his Casper/FDR2 approach. Finally, we have also detected a flaw on a protocol, the *Denning-Sacco symmetric key protocol*, that was previously reported as correct.

6.2.4 Conclusion and perspectives

This part of the AVISS project has demonstrated that a general purpose theorem prover could be used to address problems considered as typical model-checking problems. We have succeeded in turning what was first a drawback of our tool, its complex unification algorithm, into an advantage by developing a new verification strategy for protocols.

The resulting tool permits to limit the state space explosion, thus enabling the analysis of protocols that were not be studied before. In the case of the *Gong protocol*, we were able to limit the space explosion by using a symbolic handling of the properties of the xor operator. We have also begun to study more complex protocols, such as the first part of the SET protocol. In this case, the large number of constants in the adopted simplified specification was unproblematic because of the symbolic handling, using constraints, of the knowledge of the intruder and of messages sent.

We were also able, thanks to the capacity of finding type flaws, to discover a previously unreported flaw on the *Denning-Sacco symmetric key protocol*. Note that very few tools are able to handle correctly type flaws. Generally speaking, model-checkers address well-typed protocols in order to consider only a finite search space. Some other tools, using BAN logic for example [14,17], are also able to discover type flaws. The drawback, in their case, is that they sometime find artifact flaws on correct protocols. This is the case, for example, of the *Yahalom protocol*.

The initial objective of the AVISS project was to develop tools able to automatically discover 70% of the flaws in protocols already known as unsecure. The results of the daTac theorem prover outperform this goal, since we were able to automatically discover around 90% of the flaws already known.

Many issues should be addressed in the future in order to strengthen the approach by making it more efficient and by widening the spectrum of protocols and properties that can be handled.

- *Cryptographic operators.* The CL approach will be extended with a unification algorithm, and possibly new built-in inference rules in order to take the properties of $\oplus$ into account; this would be particularly useful since many protocols rely on this operator.
- *Concatenation operators.* Similarly, the associativity of the concatenation operator c should be addressed in order to conform better to the implementation with strings. Currently, we can force either left-associativity or right associativity by adding specific rules, hence it is not fully automated.
- *Timestamps.* We currently model timestamps as nonces. This approximate model prevents us from finding flaws on some protocols or leads to trivial flaws. Hence the model should be refined and may require the introduction of interval constraints.
- *Constraint solving optimizations.* A better management of set constraints inspired from the OFMC approach should definitely lead to better performance.
- *Infinite sessions.* We have successfully experimented an extension of the CL method that allows for some principals to conduct any number of sessions. This procedure terminates and might be useful to prove the absence of flaws.

6.3 The SAT-based model-checker

The SAT-based Model Checker, SATMC, takes an IF specification and generates a propositional formula whose satisfiability guarantees the reachability of a goal state from an initial state. By feeding the propositional formula to a SAT solver (currently Chaff [37], SIM [25], and SATO [52] are interfaced to SATMC), it is possible to determine its satisfiability. Since the SAT encoding computed by SATMC is constructive, from any model of the propositional formula it is possible to extract a sequence of rule applications leading from the initial state to a goal state. Therefore whenever a SAT solver finds a model for the formula, the SATMC extracts and displays the corresponding trace.

In the following we illustrate our ideas by referring to the following simple authentication protocol:

$$\begin{array}{ll} 1. & A \rightarrow B : \{Na\}Kab \\ 2. & B \rightarrow A : \{g(Na)\}Kab \end{array}$$

where Na is a nonce generated by A, Kab is a symmetric key, and g is a function known to A and B. Successful execution of the protocol should convince A that she has been talking with B, since only B could have formed the appropriate response to the message issued in step 1. In fact, I can deceit A into believing that she is talking with B whereas she is talking with her. This is achieved by executing concurrently two sessions of the protocol and using messages from one session to form messages in the other as illustrated by the following protocol trace:

$$\begin{array}{ll} 1.1. & A \rightarrow I(B) : \{Na\}Kab \\ 2.1. & I(B) \rightarrow A : \{Na\}Kab \\ 2.2. & A \rightarrow I(B) : \{g(Na)\}Kab \\ 1.2. & I(B) \rightarrow A : \{g(Na)\}Kab \end{array}$$

Principal A starts the protocol with step 1.1. Agent I intercepts the message and (pretending to be B) starts a second session with A by replaying the received message in step 2.1. A replies to this message with step 2.2. But this is exactly the message A is waiting to receive in the first protocol session. This allows A to finish the first session by using it in step 1.2. At the end of the above steps, A believes she has been talking with B, but this is obviously not the case.

6.3.1 Security problems

We regard IF specifications as specifications of security problems. A *security problem* is a tuple $\Xi = \langle \mathcal{S}, \mathcal{L}, \mathcal{R}, \mathcal{I}, \mathcal{B} \rangle$ where $\mathcal{S}$ is a set of atomic formulae of a sorted first-order language called *facts*, $\mathcal{L}$ is a set of function symbols called *rule labels*, and $\mathcal{R}$ is a set of rewrite rules of the form $L \xrightarrow{\ell} R$, where L and R are finite subsets of $\mathcal{S}$ such that the variables occurring in R occur also in L, and ℓ is an expression of the form $l(\vec{x})$ where $l \in \mathcal{L}$ and $\vec{x}$ is the vector of variables obtained by ordering lexicographically the variables occurring in L. Let S be a state and $(L \xrightarrow{\ell} R) \in \mathcal{R}$, if σ is a substitution such that $L\sigma \subseteq S$, then one possible next state of S is $S' = (S \setminus L\sigma) \cup R\sigma$ and we indicate this with $S \xrightarrow{\ell\sigma} S'$. The components $\mathcal{I}$ and $\mathcal{B}$ of a security problem are sets of states whose elements represent the initial and the bad states of the protocol respectively. A *solution to a security problem* (i.e. an attack to the protocol) is a sequence of instantiated rewrite rules whose execution leads from an initial to a bad state and the precondition of each action appears in the state to which it applies.

We now consider the security problem associated to the IF specification of the simple protocol given above. Facts are built out of a first-order sorted signature with sorts `user`, `time`, `number`, `nonceid`, `key`, `text` (super-sort of all the previous sorts), `int`, `bool`, `session`, and `list_of text`. The constants 0, 1, and 2 (of sort `int`) denote protocols steps, $\underline{1}$ and $\underline{2}$ (of sort `session`) denote session instances, a and b (of sort `user`) denote the honest participants, `intruder` (of sort `iuser`, super-sort of `user`) denotes the intruder, na (of sort `nonceid`) is a nonce identifier, `T` and `F` (of sort `bool`) denote truth and falsity respectively, and kab (of sort `key`) denotes a symmetric key. The function symbol $\{\_\}\_$: `text` $\times$ `key` $\rightarrow$ `text` denotes the encryption function, g : `number` $\rightarrow$ `number` denotes a function known to the honest participants, s : `time` $\rightarrow$ `time`,

nc : `nonceid` $\times$ `time` $\to$ `number` and sc : `session` $\to$ `session` are the time, nonce, and session constructors respectively. The predicate symbols are i of arity `text`, h of arity **`time`**, m of arity `int` $\times$ `iuser` $\times$ `iuser` $\times$ `iuser` $\times$ `text` $\times$ `session`, $witness$ of arity `iuser` $\times$ `iuser` $\times$ `nonceid` $\times$ `text`, $request$ of arity `iuser`$\times$`iuser`$\times$`nonceid`$\times$`text`, and w of arity `int`$\times$`iuser`$\times$`user`$\times$**`list_of`** `text`$\times$ **`list_of`** `text` $\times$ `bool` $\times$ `session`. The initial states are all the ground instances of:

$$\begin{aligned} & w(0,a,a,[\,],[a,b,kab],\mathrm{T},\underline{1})\,.\,w(1,a,b,[\,],[b,a,kab],\mathrm{T},\underline{1}) \\ & .\,w(0,b,b,[\,],[b,a,kab],\mathrm{T},\underline{2})\,.\,w(1,b,a,[\,],[a,b,kab],\mathrm{T},\underline{2}) \\ & .\,i(a)\,.\,i(b)\,.\,h(X) \end{aligned}$$

where here and in the following we write variables using capital letters.

The protocol rules are as follows. The following rewrite rule models the activity of sending the first message:

$$\begin{aligned} & h(s(X))\,.\,w(0,A,A,[\,],[A,B,Kab],\mathrm{T},S) \xrightarrow{step_1(A,B,Kab,S,X)} \\ & \quad h(X)\,.\,m(1,A,A,B,\{nc(na,X)\}Kab,S)\,.\,w(2,B,A,[nc(na,X)],[A,B,Kab],\mathrm{T},S). \\ & \qquad .\,witness(A,B,na,nc(na,X)) \end{aligned} \tag{6.1}$$

The receipt of the message and the reply of the responder is modeled by:

$$\begin{aligned} & h(s(X))\,.\,m(1,R,A,B,\{Na\}Kab,S)\,.\,w(1,A,B,[\,],[B,A,Kab],Xb,S) \xrightarrow{step2(A,B,C,Kab,R,S,Xb)} \\ & \quad h(X)\,.\,m(2,B,B,A,\{g(Na))\}Kab,S). \\ & \quad w(1,A,B,[\,],[B,A,Kab],Xb,sc(S))\,.\,request(B,A,na,Na) \end{aligned} \tag{6.2}$$

The final step of the protocol is modeled by:

$$\begin{aligned} & h(s(X))\,.\,m(2,R,B,A,\{g(Na)\}Kab,S)\,.\,w(2,B,A,[Na],[A,B,Kab],Xb,S) \\ & \quad \xrightarrow{step_3(A,B,C,Kab,Xb,S,Na)} h(X)\,.\,w(0,A,A,[\,],[A,B,Kab],Xb,sc(S)) \end{aligned} \tag{6.3}$$

The following rule models the ability of the intruder of overhearing the information exchanged by the honest participants:

$$m(P,Q,R,S,T,U) \xrightarrow{overhear(P,Q,R,S,T,U)} m(P,Q,R,S,T,U)\,.\,i(R)\,.\,i(S)\,.\,i(T) \tag{6.4}$$

The ability of encrypting and decrypting messages is modeled by:

$$i(T)\,.\,i(K) \xrightarrow{encrypt(K,T)} i(T)\,.\,i(K)\,.\,i(\{T\}K) \tag{6.5}$$

$$i(\{T\}K)\,.\,i(K) \xrightarrow{decrypt(K,T)} i(K)\,.\,i(T) \tag{6.6}$$

The ability of the intruder to send arbitrary messages possibly faking somebody else's identity in doing so is modeled by:

$$i(T)\,.\,i(S)\,.\,i(R) \xrightarrow{fake_{j,c}(R,S,T)} i(T)\,.\,i(S)\,.\,i(R)\,.\,m(j,intruder,S,R,T,c) \tag{6.7}$$

for all $j \in \|\texttt{int}\|$ and for all $c \in \|\texttt{session}\|$.

The following rule models the matching of a witness with a request term:

$$request(B,A,ID,T)\,.\,witness(A,B,ID,T) \xrightarrow{wmr(A,B,ID,T)} \tag{6.8}$$

Finally, any state which contains $request(B,A,na,T)$ and does not contain $witness(A,B,na,T)$ is a bad state.

6.3.2 Planning problems and their encoding into SAT

Our reduction of security problems to propositional logic is carried out in two steps. Security problems are first translated into planning problems which are then encoded into propositional formulae.

A *planning problem* is a tuple $\Pi = \langle \mathcal{F}, \mathcal{A}, Ops, I, G\rangle$, where $\mathcal{F}$ and $\mathcal{A}$ are disjoint sets of variable-free atomic formulae of a sorted first-order language called *fluents* and *actions* respectively; Ops is a set of expressions of the form

$$op(Act, Pre, Add, Del)$$

where $Act \in \mathcal{A}$ and Pre, Add, and Del are finite sets of fluents such that $Add \cap Del = \emptyset$; I and G are boolean combinations of fluents representing the initial and the final states respectively. A state is represented by a set of fluents. An action is applicable in a state S iff the action preconditions occur in S and the application of the action leads to a new state obtained from S by removing the fluents in Del and adding those in Add. A *solution to a planning problem* is a sequence of actions whose execution leads from an initial to a final state and the precondition of each action appears in the state to which it applies.

Encoding planning problems into SAT

Let $\Pi = \langle \mathcal{F}, \mathcal{A}, Ops, I, G\rangle$ be a planning problem with finite $\mathcal{F}$ and $\mathcal{A}$ and let n be a positive integer, then it is possible to build a set of propositional formulae Φ^n_Π such that any model of Φ^n_Π corresponds to a partial order plan for Π of length n and vice versa. The encoding of a planning problem into a set of SAT formulae can be done in a variety of ways (see [23, 29] for a survey). The basic idea is to add an additional time-index to the actions and fluents to indicate the state at which the action begins or the fluent holds. Fluents are thus indexed by 0 through n and actions by 0 through $n-1$. If p is a fluent or an action and i is an index in the appropriate range, then $i\!:\!p$ is the corresponding time-indexed propositional formula.

The set of formulae Φ^n_Π is the smallest set (intended conjunctively) such that:

- **Initial State Axioms:** $0\!:\!I \in \Phi^n_\Pi$;
- **Goal State Axioms:** $n\!:\!G \in \Phi^n_\Pi$;
- **Universal Axioms:** for each $op(\alpha, Pre, Add, Del) \in Ops$ and $i = 0, \ldots, n-1$:

$$\begin{gathered}(i\!:\!\alpha \supset \bigwedge\{i\!:\!p \mid p \in Pre\}) \in \Phi^n_\Pi\\ (i\!:\!\alpha \supset \bigwedge\{(i+1)\!:\!p \mid p \in Add\}) \in \Phi^n_\Pi\\ (i\!:\!\alpha \supset \bigwedge\{\neg(i+1)\!:\!p \mid p \in Del\}) \in \Phi^n_\Pi\end{gathered}$$

- **Explanatory Frame Axioms:** for all fluents f and $i = 0, \ldots, n-1$:

$$\begin{gathered}(i\!:\!f \wedge \neg(i+1)\!:\!f) \supset \bigvee\{i\!:\!\alpha \mid op(\alpha, Pre, Add, Del) \in Ops, f \in Del\} \in \Phi^n_\Pi\\ (\neg i\!:\!f \wedge (i+1)\!:\!f) \supset \bigvee\{i\!:\!\alpha \mid op(\alpha, Pre, Add, Del) \in Ops, f \in Add\} \in \Phi^n_\Pi\end{gathered}$$

- **Conflict Exclusion Axioms:** for $i = 0, \ldots, n-1$:

$$\neg(i\!:\!\alpha_1 \wedge i\!:\!\alpha_2) \in \Phi^n_\Pi$$

  for all $\alpha_1 \neq \alpha_2$ such that $op(\alpha_1, Pre_1, Add_1, Del_1) \in Ops$, $op(\alpha_2, Pre_2, Add_2, Del_2) \in Ops$, $Pre_1 \cap Del_2 \neq \emptyset$, and $Pre_2 \cap Del_1 \neq \emptyset$. To avoid generation of redundant formulae it is convenient to include the additional requirement $\alpha_1 \prec \alpha_2$, where $\prec$ is an ordering over actions. Conflict Exclusion Axioms ensure the linearizability of plans by excluding the simultaneous execution of any pair of actions such that the effect of one action prevents the applicability of the other.

It is immediate to see that the number of literals in Φ^n_Π is in $O(n|\mathcal{F}| + n|\mathcal{A}|)$ where $|\mathcal{F}|$ and $|\mathcal{A}|$ are the cardinalities of the sets $\mathcal{F}$ and $\mathcal{A}$, respectively. Moreover the number of Universal Axioms is in $O(nP_0|\mathcal{A}|)$ where P_0 is the maximal number of fluents mentioned in an operator (usually a small number); the number of Explanatory Frame Axioms is in $O(n|\mathcal{F}|)$; finally, the number of Conflict Exclusion Axioms is in $O(n|\mathcal{A}|^2)$.

6.3.3 Security problems as planning problems

Given a security problem $\Xi = \langle \mathcal{S}, \mathcal{L}, \mathcal{R}, \mathcal{I}, \mathcal{B}\rangle$, it is possible to build a planning problem $\Pi_\Xi = \langle \mathcal{F}_\Xi, \mathcal{A}_\Xi, Ops_\Xi, I_\Xi, G_\Xi\rangle$ such that each solution to Π_Ξ can be translated back to a solution to Ξ: $\mathcal{F}_\Xi$ is the set of facts $\mathcal{S}$; $\mathcal{A}_\Xi$ and Ops_Ξ are the smallest sets such that $\ell\sigma \in \mathcal{A}_\Xi$ and $op(\ell\sigma, L\sigma, R\sigma, L\sigma \setminus R\sigma) \in Ops$ for all $(L \xrightarrow{\ell} R) \in \mathcal{R}$ and all ground substitutions σ; finally $I_\Xi = \bigvee_{S_\mathcal{I} \in \mathcal{I}} \bigwedge S_\mathcal{I}$ and $G_\Xi = \bigvee_{S_\mathcal{B} \in \mathcal{B}} \bigwedge S_\mathcal{B}$.

For instance, the actions associated to (6.1) are of the form:

$$\begin{array}{l} op(step_1(A, B, Kab, S, X), \\ \quad [h(s(X)), \\ \quad\ w(0, A, A, [\,], [A, B, Kab], \mathrm{T}, S)], \\ \quad [h(X), \\ \quad\ m(1, A, A, B, \{nc(na, X)\}Kab, S), \\ \quad\ w(2, B, A, [nc(na, X)], [A, B, Kab], \mathrm{T}, S), \\ \quad\ witness(A, B, na, nc(na, X))], \\ \quad [w(0, A, A, A, [\,], [A, B, Kab], \mathrm{T}, S), \\ \quad\ h(s(X))]) \end{array} \tag{6.9}$$

The reduction of security problems to planning problems paves the way to an automatic SAT-compilation of security problems. However a direct application of the approach (namely the reduction of a security problem Ξ to a planning problem Π_Ξ followed by a SAT-compilation of Π_Ξ) is not immediately applicable. We therefore devised a set of optimizations and heuristics whose combined effects often succeed in drastically reducing the size of the SAT instances generated by the SATMC.

Optimizations

Language specialization. We recall that for the reduction to propositional logic described in Section 6.3.2 to be applicable the set of fluents and actions must be finite. Unfortunately, the security problems introduced in Section 6.3.3 have an infinite number of facts and rule instances, and therefore the corresponding planning problems have an infinite number of fluents and actions. However the language can be restricted to a finite one since the set of states reachable in n steps is obviously finite (as long as the initial states comprise a finite number of facts). To determine a finite language capable to express the reachable states, it suffices to carry out a static analysis of the security problem.

To illustrate, let us consider again the simple security problem presented above and let $n = 7$, then $\|\texttt{int}\| = \{0, 1, 2\}$, $\|\texttt{user}\| = \{a, b\}$, $\|\texttt{iuser}\| = \{a, b, \texttt{intruder}\}$, $\|\texttt{key}\| = \{kab\}$, $\|\texttt{nonceid}\| = \{na\}$, $\|\texttt{session}\| = \bigcup_{i=0}^{\lfloor n/(k+1)\rfloor} sc^i(\underline{1}) \bigcup_{i=0}^{\lfloor n/(k+1)\rfloor} sc^i(\underline{2})$, where k is the number of protocol steps in a session run (in this case $k = 2$), $\|\texttt{time}\| = \{s^i(0) : i = 0, \ldots, 6\}$, $\|\texttt{number}\| = \bigcup_{i=0}^{6} g^i(nc(\texttt{nonceid}, \texttt{time}))$, $\|\texttt{text}\| = \|\texttt{iuser}\| \cup \|\texttt{key}\| \cup \|\texttt{number}\| \cup \bigcup_{i=0}^{6}\{g^i(nc(\texttt{nonceid}, \texttt{time}))\}kab$.[5] Moreover, we can safely replace `list_of text` with $[\texttt{text}, \texttt{text}, \texttt{text}]$. The set of facts is then equal to $i(\texttt{text}) \cup h(\texttt{time}) \cup m(\texttt{int}, \texttt{iuser}, \texttt{iuser}, \texttt{iuser}, \texttt{text}, \texttt{session}) \cup witness(\texttt{iuser}, \texttt{iuser}, \texttt{nonceid}, \texttt{text}) \cup request(\texttt{iuser}, \texttt{iuser}, \texttt{nonceid}, \texttt{text}) \cup w(\texttt{int}, \texttt{iuser}, \texttt{user}, \texttt{list\_of text}, \texttt{list\_of text}, \texttt{bool}, \texttt{session})$ which comprises 10^{14} facts. This language is finite, but still definitely too big for the practical applicability of the SAT encoding.

[5] If $S_1, \ldots, S_m$ and S' are sorts and f is a function symbol of arity $S_1, \ldots, S_m \to S'$, then $\|S_i\|$ is the set of terms of sort S_i and $f(S_1, \ldots, S_m)$ denotes $\{f(t_1, \ldots, t_m) : t_i \in \|S_i\|, i = 1, \ldots, m\}$.

A first important optimization is obtained by changing the way fresh constants are generated and handled. This is achieved by removing the sort `time`, the constant 0, the function symbol s : `time` $\to$ `time`, by replacing the predicate symbol h with a new predicate *fresh* of sort `number`, and by changing the arity of nc to `nonceid` $\times$ `session` $\to$ `number`. In the new language $\|\texttt{number}\| = \bigcup_{i=0}^{6} g^i(nc(\texttt{nonceid}, \texttt{session}))$ and $\|\texttt{text}\| = \|\texttt{iuser}\| \cup \|\texttt{key}\| \cup \|\texttt{number}\| \cup \bigcup_{i=0}^{6}\{g^i(nc(\texttt{nonceid}, \texttt{session}))\}kab$. This gives a language with 10^{13} facts.

A simple optimization is obtained by removing redundant fields from the facts of the form $w(\ldots)$ and $m(\ldots)$. For instance, the boolean field can be eliminated from the terms of form $w(\ldots)$ thereby changing the arity of w to `int`$\times$`iuser`$\times$`user`$\times$`list_of text`$\times$`list_of text`$\times$`session`.[6] A similar simplification can be carried out on the terms of the form $m(\ldots)$ by removing the fields devoted to the real sender and to the session thereby changing the arity of m to `int` $\times$ `iuser` $\times$ `iuser` $\times$ `text`. The application of these two optimizations halves the number of facts.

A closer look to the protocol reveals that the above language still contains many spurious facts. In particular the $m(\ldots)$, $w(\ldots)$, and $i(\ldots)$ can be specialized (e.g. by using specialized sorts to restrict the message terms to those messages which are allowed by the protocol). By analyzing carefully the facts of the form $m(\ldots)$ and $w(\ldots)$ occurring in the protocol rules of our example we can restrict the sort `text` in such a way that $\|\texttt{text}\| = \|\texttt{iuser}\| \cup \|\texttt{key}\| \cup \|\texttt{number}\| \cup \{nc(\texttt{nonceid}, \texttt{session})\}kab \cup \{g(nc(\texttt{nonceid}, \texttt{session}))\}kab$ and `list_of text` to $[\texttt{iuser}, \texttt{iuser}, \texttt{key}] \cup [\texttt{number}]$. Thanks to this optimization, the number of facts drops to 10^5.

Fluent splitting. The second family of optimizations is based on the observation that in $w(j, s, r, ak, ik, c)$, the union of the first three arguments with the sixth form a key (in the data base theory sense) for the relation. This allows us to modify the language by replacing $w(j, s, r, ak, ik, c)$ with the conjunction of two new predicates, namely $wk(j, s, r, ak, c)$ and $inknw(j, s, r, ik, c)$. Similar considerations (based on the observation that the initial knowledge of a principal r does not depend on the protocol step j nor on the s) allow us to simplify $inknw(j, s, r, ik, c)$ to $inknw(r, ik, c)$. Another effective improvement stems from the observation that ak and ik are lists. By using the set of new facts $wk(j, s, r, ak_1, 1, c), \ldots, wk(j, s, r, ak_l, l, c)$ in place of $wk(j, s, r, [ak_1, \ldots, ak_l], c)$ the number of wk terms is no longer linear in $|\texttt{list\_of text}|^{|\texttt{list\_of text}|}$ but simply linear in $|\texttt{text}| * l$.[7] The application of fluent splitting to the aforementioned language with 10^5 facts reduces the number of facts to 10^4.

Exploiting static fluents. The previous optimization enables a new one. Since the initial knowledge of the honest principal does not change as the protocol execution makes progress, facts of the form $inknw(r, ik, c)$ occurring in the initial state are preserved in all the reachable states and those not occurring in the initial state will not be introduced. In the corresponding planning problem, this means that all the atoms $i : inknw(r, ik, c)$ can be replaced by $inknw(r, ik, c)$ for $i = 0, \ldots, n-1$ thereby reducing the number of propositional letters in the encoding. Moreover, when the initial state is unique, this transformation enables an off-line simplification of the propositional formula.

Invariants. Invariants are an effective way to decrease the number of fluents. Invariants of the form $t = t'$or $p \leftrightarrow p'$ can be used to normalize fluents with the hope of turning two or more fluents into a single fluent which acts as representative. Invariants of the form p or $\neg p$ can instead be used to simplify fluents to the boolean constants *true* and *false* respectively. For instance, we can

[6]It is worth emphasizing that the role of the boolean field in the principal term is operational and it is mainly used to avoid useless repetitions of the impersonate rules. The boolean field is thus used by the OFMC and the CL theorem-prover to reduce the branching factor and avoid the exploration of already visited portions of the search space. However, the use of the boolean field has the nasty effect of doubling the cardinality of principal terms. Since the number of fluents is critical in our approach, we found it convenient to drop the boolean field from the principal terms. Notice that in doing this we get smaller SAT instances which in principle can be harder to solve. However computer experiments indicate that the performance of the SAT solvers is unaffected.

[7]If S is a sort, then $|S|$ is the cardinality of $\|S\|$.

state that the official sender and the receiver in a message term must be different by declaring an invariant of form $\neg m(\_, x, x, \_)$.

Reducing the number of Conflict Exclusion Axioms. A critical issue in the propositional encoding technique described in Section 6.3.2 is the quadratic growth of the number of Conflict Exclusion Axioms in the number of actions. This fact often confines the applicability of the method to problems with a small number of actions. A way to lessen this difficulty is to reduce the number of conflicting axioms by considering the intruder knowledge as *monotonic*. Let s be a fact, then we say that s is monotonic iff for all S if $s \in S$ and $S \rightarrow S'$, then $s \in S'$. Since a monotonic fluent never appears in the delete list of some action, then it cannot be a cause of a conflict. The idea here is to transform the rules so to make the facts of the form $i(\ldots)$ monotonic. The transformation on the rules is very simple as it amounts to adding the monotonic facts occurring in the left hand side of the rule to its right hand side. A consequence is that a monotonic fact simplifies the Explanatory Frame Axioms relative to it. The nice effect of this transformation is that the number of Conflict Exclusion Axioms generated by the associated planning problems drops dramatically. For instance, application of this optimization to the analysis of the NSPK protocol and of the Encrypted Key Exchange (EKE) protocol reduces the number of Conflict Exclusion Axioms from 740 to 20 and from 6,678 to 324, respectively.

Note that another possible solution to the problem of the number of Conflict Exclusion Axioms is to avoid their generation altogether. Of course, by doing this, we are no longer guaranteed that the partial plans found are linearizable and a check must be performed to ensure this. This approach is further discussed in Section 6.3.4.

Merging intruder and protocol rules. Other important optimizations are based on the idea of merging the intruder rules with protocol rules. As first instance consider the following. By merging with (6.4), a generic protocol rule:

$$\ldots \xrightarrow{step_i(\ldots)} \ldots \centerdot m(j, s, r, t) \centerdot \ldots$$

is transformed into

$$\ldots \xrightarrow{step_i(\ldots)} \ldots \centerdot m(j, s, r, t) \centerdot i(s) \centerdot i(r) \centerdot i(t) \centerdot \ldots$$

By systematically repeating this step for all protocol rules and then removing (6.4) we get an optimized set of rules. In our example, this optimization step allows us to eliminate all the 32 instances of (6.4), thereby leaving us with a total of 84 rule instances. Intuitively speaking, this transformation amounts to incorporating the intruder's activity of memorizing the content of the messages into the protocol rules. An evolution of the previous optimization is to apply the step-compression optimization which is described in details in Section 4.2.

Heuristics

In some cases in order to get encodings of reasonable size, we must supplement the above attack-preserving optimizations with the following heuristics. By heuristics here we mean a transformation that reduces the size of the encoding and is not attack preserving. Our experience with the following heuristics indicates that they are successful in most cases.

Bounding the number of session runs. Let s and j be the bounds in the number of operation applications and in the number of protocol steps characterizing a protocol session respectively. Then the maximum number of times a session can be repeated is $max\_ses\_rep = \lfloor s/(j + 1) \rfloor$. The results published in literature as well as our experience show that the attacks require a number of session repetitions (ses_rep) that is less than max_ses_rep. For example, the NSPK protocol is characterized by 3 protocol steps (i.e. $j = 3$), the attack is found with $s = 10$ and $ses\_rep = 1$, while $max\_ses\_rep = 2$. Hence, it is convenient to bound the number of session runs to this number. By using this optimization we reduce the cardinality of the sort `session` (in the case of the NSPK protocol, we reduce it by a factor 1.5) and therefore the number of facts that depend on it.

Multiplicity of fresh terms. The number of fresh terms needed to find an attack is usually less than the number of fresh terms available. As a consequence, a lot of fresh terms allowed by the language associated with the protocol are not used, and many facts depending on them are allowed, but also not used. Often, one single fresh term for each fresh term identifier is sufficient for finding the attack. Therefore, the basic idea of this heuristics is to restrict the number of fresh terms available.

Constraining the rule variables. This heuristics is best illustrated by considering the Kao-Chow protocol:

1. $A \rightarrow S : A, B, Na$
2. $S \rightarrow B : \{A, B, Na, Kab\}Kas, \{A, B, Na, Kab\}Kbs$
3. $B \rightarrow A : \{A, B, Na, Kab\}Kas, \{Na\}Kab, Nb$
4. $A \rightarrow B : \{Nb\}Kab$

In the second step S sends B a pair of messages of which only the second component is accessible to B. Since B does not know Kab, then B cannot check that the occurrence of A in the first component is equal to that inside the second. As a matter of fact, we might have different terms at those positions. The heuristics amounts to imposing that the occurrences of A (as well as of B, Na, and Kab) in the first and in the second part of the message must coincide. Thus, messages of the form $m(2, a, a, b, c(\{a, b, nc(na, s(1))\}kas, \{a, b, nc(nb, s(1))\}kbs), \_)$ would be ruled out by the heuristics.

6.3.4 Implementation

Architecture

The architecture of the SATMC is depicted in Figure 6.4. The IF2SATE compiler translates the IF problem given as input into a planning problem specified in the SATE specification language using the optimizing transformations described in Section 6.3.3. The planning problem is transformed into a SAT formula by the SATE compiler. The SAT formula is then fed to a SAT solver. If the formula is found unsatisfiable by the solver then a failure is reported, otherwise the module Model2Plan extracts a plan from the satisfying assignment. If the plan is *feasible* (i.e. if it is linearizable) then it is reported to the user, otherwise a set of clauses excluding the simultaneous execution of the conflicting actions is added to the propositional formula and the resulting SAT instance is fed back to the SAT solver.

The SATE specification language

To express planning problems we have designed a STRIPS-like specification language for planning problems called SATE. The syntax of the SATE language is given by the BNF grammar of Figure 6.5. Here is a description of the main constructs of the language:

- **Sort declaration** (*sort_ decl*). For example, the declarations `sort(fluent)`, `sort(action)`, and `sort(time)` declare `fluent`, `action`, and `time` to be sorts.
- **Super-Sort declaration** (*super_ sort_ decl*). For example, the declarations `super_sort(atom,mr)` and `super_sort(atom,pk)` declare `atom` to be super-sort of `mr` and `pk`.

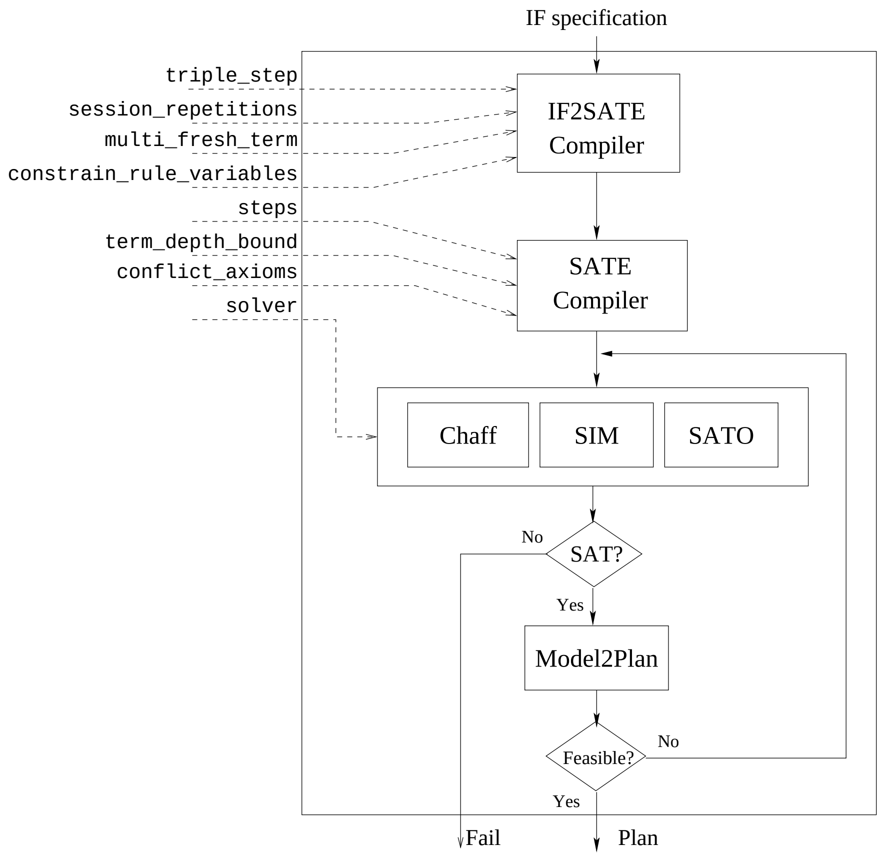


Figure 6.4: Architecture of the SAT-based model-checker

- **Constant declaration** (*constant_ decl*). For example, the constant declarations `constant(0,time)` and `constant(s(time),time)` declare `0` to be an individual constant of sort `time` and `s` to be a function symbol of arity `time` $\rightarrow$ `time`.

- **Invariant** (*invariant*). Invariant declarations are used to state facts that hold in all time instants. For example, the invariant `invariant(~(m(_,A,A,_)))` states that all ground instances of `m(_,A,A,_)` are false in all time instances.

- **Static** (*static*). A declaration of the form `static(`P`)` states that all ground instances of P are static fluents i.e. their initial truth value cannot possibly change. Formally, if p is a ground fluent instance of P, then $0{:}p \equiv i{:}p$ for $i = 1, \ldots, n$.

- **Monotone** (*monotone*). A declaration of the form `monotone(`P`)` states that all ground instances of P can change their truth value from false to true, but not vice versa. Formally, if p is a ground fluent instance of P, then $i{:}p \supset i+1{:}p$ for $i = 0, \ldots, n-1$.

- **Initial State** (*initial_ state*). This statement specifies the initial states. For example, the declaration

```
facts([h(s(s(s(0)))),
       wk(0,mr(intruder),mr(a),etc,1,1),
       wk(1,mr(a),mr(b),etc,1,1)]).
```

  asserts that the listed fluents hold at time 0.

- **Goal** (*goal_ state*). For example, the assertion

sate_spec	::=	(assertion .)*
assertion	::=	sort_decl | super_sort_decl | constant_decl | invariant | monotone | static initial_state | goal_state | operator
sort_decl	::=	`sort(`sort_id`)`
super_sort_decl	::=	`super_sort(`sort_id,sort_id`)`
constant_decl	::=	`constant(`arity,sort_id`)`
invariant	::=	`invariant(`invariant_composite_fluent`)`
monotone	::=	`monotone(`fluent`)`
static	::=	`static(`fluent`)`
initial_state	::=	`facts(`init`)`
goal_state	::=	`goal(`condition, goal`)`
operator	::=	`action(`action, condition, pre, add, del`)`
arity	::=	const_id | const_id`(`sort_id⋆`)`
fluent, action	::=	const_id | const_id`(`term⋆`)`
term	::=	var | const_id | const_id`(`term⋆`)`
pre, add, del	::=	`[`fluent⋆`]`
init, goal	::=	`[`wff⋆`]`
invariant_composite_fluent	::=	term `=` term | fluent | `~` fluent | fluent `<=>` fluent
wff	::=	fluent | `~` wff | wff `&` wff | wff `v` wff
sort_id, const_id	::=	[a-z,A-Z][_ ,a-z,A-Z,0-9]*
var	::=	[_ ,A-Z][_ ,a-z,A-Z,0-9]*

Proviso: *In the production rule for operator:* $Vars(cond), Vars(prec), Vars(add), Vars(del) \subseteq Vars(actions)$.

Legenda: $X^\star$ *abbreviates* $(X(,X)^*)$, *and* $X_1, \ldots, X_n ::= E$ *abbreviates* $X_1 ::= E \ldots X_n ::= E$.

Figure 6.5: Grammar for the SATE specification language

```
goal(true,[~(witness(mr(a),mr(b),X1,X2)),request(mr(b),mr(a),X1,X2)]).
```

states that every state S, such that `request(mr(b),mr(a),X1`σ`,X2`σ`)`$\in S$ and `witness(mr(a),mr(b),X1`σ`,X2`σ`)`$\notin S$ for some ground substitution σ, is a goal state.

- **Operator** (*operator*). For example, the assertion

```
action(step_1(XA,XB,XKab,XS,X),
       true,
       [h(s(X)),
        wk(0,mr(intruder),XA,etc,1,XS)],
       [h(X),
        m(1,XA,XB,crypt(XKab,nonce(c(na,X)))),
        wk(2,XB,XA,nonce(c(na,X)),1,XS)],
       [h(s(X)),
        wk(0,mr(intruder),XA,etc,1,XS)]).
```

states that if S is a state and σ a substitution such that `h(s(X))`σ and `wk(0,mr(intruder), XA,etc,1,XS)`σ are in S, then the state obtained from S by removing `h(s(X))` and `wk(0, mr(intruder),XA,etc,1,XS)` and adding `h(X)`σ, `wk(2,XB,XA,nonce(c(na,X)),1,XS)`σ, and `m(1,XA,XB,crypt(XKab,nonce(c(na,X))))`σ is a successor state of S.

The IF2SATE compiler

The IF2SATE compiler takes as input an IF specification and generates a SATE specification by applying the optimizations presented in Section 6.3.3. The result of the compilation depends on the following parameters:

- `session_repetitions`, a positive integer specifying the maximal number of session iterations allowed for the protocol (this parameter is used by the heuristics "bounding the number of session runs");
- `triple_step`, a boolean parameter stating whether the step-compression optimization is enabled or not;
- `multi_fresh_term`, a boolean parameter stating whether the terms of sort `fresh` must depend on the session iteration or not (this parameter is used by the heuristics "multiplicity of fresh terms"); and
- `constrain_rule_variables`, a boolean parameter stating whether the heuristics based on constraining the rule variables (see Section 6.3.3) is enabled or not.

Note that by setting the parameter `session_repetitions` to a value smaller than $\lfloor s/(j+1)\rfloor$ (s and j are the bounds in the number of operation applications and in the number of protocol steps characterizing a protocol session, respectively), or by setting the parameter `multi_fresh_term` to `false` or `constrain_rule_variables` to `true`, we can obtain dramatic savings in the size of the language, but in doing this we may loose some attacks to the protocol.

The SATE compiler

The SATE compiler takes as input a SATE specification and generates a propositional formula by applying the encoding technique presented in Section 6.3.2. The result of the compilation depends on the following parameters:

- `steps`, the maximal length of the partial-order plan;
- `term_depth_bound`, a bound on the depth of the expressions of the language. Setting this parameter is necessary if the language specified by the SATE specification given as input has infinite size. Notice that this is never the case for the SATE specifications generated by the IF2SATE compiler.
- `conflict_axioms`, a boolean parameter stating whether Conflict Exclusion Axioms must be generated or not. As pointed out in Section 6.3.3, setting this parameter to `false` may reduce significantly the number of clauses but the assignment returned by the SAT solver is no longer guaranteed to correspond to a feasible execution trace.

The SATE compiler generates a propositional formula whose satisfiability guarantees the existence of a partial-order plan π such that:

1. the length of π is no greater than `steps`;
2. the depth of the action expressions occurring in π is no greater than `term_depth_bound`. The same bound holds for all the fluents occurring in any state generated by any possible execution of π starting from an initial state;
3. (if `conflict_axioms` is set to `true`) the plan π is linearizable.

Figure 6.6 displays the output returned by SATMC when given as input the NSPK protocol and the following settings of the parameters:

```
set(steps,10).
set(triple_step,false).
set(session_repetitions,2).
set(multi_fresh_term,true).
set(constrain_rule_variables,true).
set(conflict_axioms,true).
set(term_depth_bound,10).
```

The first row indicates the time required by the IF2SATE module to translate the IF specification into SATE. The output then contains the statistics of the pre-processing phase that instantiates and stores the fluents and actions, and the statistics concerning the generation of the SAT formula. Then, the numbers of atoms and of clauses characterizing the propositional formula are given. Finally, SATMC outputs the time spent by the SAT solver and the trace of the attack.

6.3.5 Experimental results

In our experience, the SAT encodings generated from untyped IF specification are usually very big. Therefore we focused on the typed IF. The implication of this fact is that type flaws are not found by SATMC.

By running SATMC against the testsuite with different values of the parameters we found that the setting that leads to the best performance (both in the number of attacks found and in the time spent to find them) is the following:

```
set(triple_step,true).
set(steps,10).
set(session_repetitions,2).
set(multi_fresh_term,true).
set(constrain_rule_variables,true).
set(conflict_axioms,false).
```

We thus fixed the parameters of SATMC to these values and run our tool against all the protocols of the Clark/Jacob library that are known to suffer from an attack of kind different from a type flaw. The results are reported in Table 6.1.

For each protocol the table gives the kind of attack (Attack), the number of actions (A) and the number of fluents (F) of the planning problem, the time spent by the SATE compiler to generate the propositional encoding (E), as well as the time spent by Chaff to solve the corresponding SAT instance (S).[8]

The labels TO and MO indicate a failure to analyze the protocol due to time-out and memory-out, respectively.[9] The label FF indicates a finite failure (this happens only on the *Needham-Schroeder Signature protocol*).

The experiments show that both the translation carried out by IF2SATE and the SAT solving activity are carried out very quickly and that the overall time is dominated by the SAT encoding.

6.3.6 Conclusion and perspectives

We have implemented a tool that reduces security problems to propositional logic. Application of the tool to a set of well-known authentication protocols shows that the propositional formulae generated by the tools are solved in a fraction of a second by state-of-the-art SAT solvers and hence attacks to the protocols are found very quickly by our tool.

The only problem with the approach is the size of and, consequently, the time spent generating the propositional encoding. A promising approach to tackle this problem amounts to treating properties of cryptographic operations as invariants. Currently these properties are modeled as

[8]Times have been obtained on a PC with a 1.4 GHz Processor and 512 MB of RAM.

[9]We set a time limit of 1 hour for each attempt. Due to a limitation of SICStus Prolog the SAT-based model-checker is bound to use 128Mb during the encoding generation.

```
SATE file generated in 0.41 sec...


STATISTICS (PREPROCESS PHASE)   RUNTIME(sec)
Fluents Preprocessing:          0.16
Actions Preprocessing:          0.36
                                ------
Total:                          0.52


Fluents: 505
Actions: 820


STATISTICS (CLAUSES GENERATION) CLAUSES RUNTIME(sec)
Initial Facts:                  490     0.02
Goals:                          16      0.0
Ape Axioms:                     50320   1.25
Explanatory Frame Axioms:       9800    0.36
Conflict Exclusion Axioms:      340     0.02
                                        ------
Total:                                  1.65

Number of Atoms:        13590
Number of Clauses:      60966

Plan found in 0.04 sec and in 10 steps:


Step 0:          [2.1. a -> I : {na(2),a}ki]

Step 1:          [decrypt_public_key_1(pk(ki),c(nonce(c(na,2)),mr(a)))]

Step 2:          [decompose_1(nonce(c(na,2)),mr(a))]

Step 3:          [1.1. I(a) -> b :{na(2),a}kb]

Step 4:          [1.2. b -> a : {na(2),nb(1)}ka]

Step 5:          [2.2. I -> a : {na(2),nb(1)}ka]

Step 6:          [2.3. a -> I : {nb(1)}ki]

Step 7:          [2.1. a -> I : {na(s(2)),a}ki]

Step 8:          [1.3. I(a) -> b :{nb(1)}kb]

Step 9:          [decompose_1(nonce(c(na,s(2))),mr(a)),step_3_1_1_1(c(nb,1),c(na,2))]


Total Time:      2.1
```

Figure 6.6: Output generated by SATE

Table 6.1: Experimental results

Protocol	**Attack**	**A**	**F**	**E**	**S**
ISO symmetric key 1-pass unilateral authentication	Replay	25	47	0.18	0.00
ISO symmetric key 2-pass mutual authentication	Replay	87	110	0.43	0.01
Andrew Secure RPC Protocol	Replay	15650	473	80.57	2.65
ISO CCF 1-pass unilateral authentication	Replay	22	47	0.46	0.01
Needham-Schroeder Conventional Key	Replay STS	5220	6771	29.25	0.39
Woo-Lam Π	Parallel session	252	290	3.31	0.04
Woo-Lam Mutual Authentication	Parallel session	27051	45602	1024.08	7.95
Needham-Schroeder Signature protocol	Man-in-the-middle	268	443	FF	
Neuman Stubblebine repeated part	Replay STS	2408	1425	15.17	0.21
Kehne Langendorfer Schoenwalder (repeated part)	Parallel session	-	-	MO	-
Kao Chow Repeated Authentication, 1	Replay STS	2042	2766	16.34	0.17
Kao Chow Repeated Authentication, 2	Replay STS	22344	32976	339.7	2.11
Kao Chow Repeated Authentication, 3	Replay STS	55727	49391	1288	MO
ISO public key 1-pass unilateral authentication	Replay	49	72	0.32	0.00
ISO public key 2-pass mutual authentication	Replay	249	140	1.18	0.01
Needham-Schroeder Public Key	Man-in-the-middle	604	313	1.77	0.05
Needham-Schroeder Public Key with key server	Man-in-the-middle	662	460	4.29	0.04
SPLICE/AS Authentication Protocol	Replay	658	841	5.48	0.05
Encrypted Key Exchange	Parallel session	10924	1160	75.39	1.78
Davis Swick Private Key Certificates, protocol 1	Replay	258	518	1.37	0.02
Davis Swick Private Key Certificates, protocol 2	Replay	450	711	2.68	0.03
Davis Swick Private Key Certificates, protocol 3	Replay	482	534	1.5	0.02
Davis Swick Private Key Certificates, protocol 4	Replay	1283	1370	8.18	0.13

Legenda: TO: Time Out
MO: Memory Out
FF: Finite Failure
Replay STS: Replay attack based on a Short-Term Secret

rewrite rules in the IF. This has a bad impact on the number of actions in the corresponding planning problem and hence in the size of the final encoding. A more natural way to look at the properties of cryptographic operations is as invariants of the input security problem. However the incorporation of invariants in our encoding mechanism requires, among other things, a modification of the explanatory frame axioms and therefore more work (both theoretical and implementational) is needed to exploit this very promising transformation.

Moreover, we would like to experiment SATMC against security problems with multiple initial states. Problems of this kind occur when the initial knowledge of the principal in not completely defined or when the session instances are partially defined. We conjecture that neither the size of the SAT encoding nor the time spent by the SAT solver to check the SAT instances will be significantly affected by this generalization. But this requires some changes in the current implementation of SATMC and a thorough experimental analysis.

We plan to focus on these questions as part of future work in the remaining month of the AVISS project, as well as in the follow-up RTD project with industry involvement that we are currently planning.

7. Experimental results

In this section we illustrate the experimental testsuite we have set up to demonstrate that the AVISS tool achieves the project objectives, according to the success criteria adopted as a measure of the assessment in the project proposal, i.e.

Coverage: 46 of 51 of the protocols in the Clark/Jacob library [18] are specifiable in the tool's input specification language HLPSL, which amounts to a coverage of 90% (80% was requested).

Effectiveness: the tool detects attacks in 91% of the protocols in the library known to be insecure (70% was requested). More specifically, 35 out of the 51 protocols are flawed, and we can find a flaw in 32 of them, including a previously unknown flaw.

Performance: the tool can find the attack for 91% of the flawed protocols in the Clark/Jacob library (i.e. the 32 flawed protocols that can be modeled), in less than one minute of CPU time on a 1.4 GHz processor (the criterion required that at least 70% of the (flawed) protocols were processed in less than 1 hour of CPU time).

7.1 The testsuite

By following [22] and counting different versions of the protocols as different protocols, the Clark/Jacob library contains 51 protocols, 34 of which were already known to be flawed. Our tool finds a previously unknown flaw of the Denning-Sacco symmetric protocol. (This flaw in the Denning-Sacco symmetric protocol was first discovered by the Nancy group and is described in Section 6.2.3.) Thus, the Clark/Jacob library contains 35 protocols that are flawed. The remaining 16 protocols are to our knowledge flawless: in all the session instances we have considered so far, our tool has never revealed a flaw in a reasonable amount of time (1 hour). Hence, here we focus on the 35 protocols that are known to be flawed.

We were able to formalize all known flawed protocols from the Clark/Jacob library and their security goals in HLPSL and automatically translate them into the IF with 3 exceptions:

- the Kerberos protocol and the Wide Mouthed Frog protocol could not be modeled, because they rely on timeliness properties of timestamps that cannot be specified in HLPSL yet;
- for the CCITT X.509 protocol it was not possible to specify an appropriate security goal, namely that an agent only signs messages he knows.

Of the 16 protocols that are to our knowledge flawless, we can specify 14: we cannot specify the Diffie-Hellman exchange protocol, because it uses key exponentiation, and we cannot specify the Gong mutual authentication protocol because it uses XOR encryption. (Note, however, that the Nancy group has carried out some preliminary experiments on XOR encryption extending the current HLPSL syntax to specify the Gong protocol. See Section 6.2.3 for a discussion on this.)

In our AVISS tool, we can thus specify 46 of the 51 protocols of the library, which yields a coverage of 90%.

It is worth pointing out that, to our surprise, our tool initially failed to find an attack on the Yahalom protocol. A closer look to the protocol soon revealed that the attack described in the Clark/Jacob survey is quite questionable. The original protocol is:

1. $A \rightarrow B : A, Na$
2. $B \rightarrow S : B, \{A, Na, Nb\}_{Kbs}$
3. $S \rightarrow A : \{B, Kab, Na, Nb\}_{Kas}, \{A, Kab\}_{Kbs}$
4. $A \rightarrow B : \{A, Kab\}_{Kbs}, \{Nb\}_{Kab}$

The attack described is:

1. $I(a) \rightarrow b : a, na$
2. $b \rightarrow I(s) : b, \{a, na, nb\}_{kas}$
4. $I(a) \rightarrow b : \{a, na, nb\}_{kas}, \{nb\}_{(na,nb)}$

Here the intruder tricks b into accepting na, nb as a new key. However, to produce the final message of the above attack trace, he must know nb. Since nb is a fresh nonce created by b, the intruder can not know it, and hence in our model this protocol is flawless. Note that Lowe comes to the same conclusion in [22]. However, if the intruder can find out nb (perhaps, because it is guessable), then the attack does exist. Similarly to Lowe, we model guessability by a transmission of the nonce in cleartext in step 2:

2. $B \rightarrow S : B, \{A, Na, Nb\}_{Kbs}, Nb$

This attack is particularly interesting since it requires a composed key, while the protocol itself does not handle them. As a consequence, this flaw can not be found using impersonate rules, whereas with the lazy intruder this is an almost trivial task. Hence, we count (this slightly modified version of) the Yahalom protocol as flawed. Note that, as we discuss in Section 8, this also demonstrates an advantage of our tool with respect to other approaches that cannot model non-atomic keys.

The list of flawed protocols in our testsuite is given in the leftmost column of Table 7.1. Note that the lists includes two variants of the Needham-Schroeder public key protocol that are not taken into account in [22]: without key-server (*Needham-Schroeder Public Key*) and with Lowe's fix (*Needham-Schroeder with Lowe's fix*; cf. Section 6.1.4). Moreover for the Neuman-Stubblebine protocol (*Neuman Stubblebine (complete)*) we consider as single protocols the initial part (*Neuman Stubblebine initial part*) and the repeated part (*Neuman Stubblebine repeated part*) since the repeated part contains a flaw that results from a type flaw in the initial part. It is worth remarking that the four protocols marked with a "*" in Table 7.1 are additional case studies we have carried out with our back-ends, but that these protocols are not taken into account when computing the figures of the project results.

7.2 Computer experiments

In order to assess the effectiveness of our tool in general and, more in particular, of the individual back-ends, we run our tool against the protocols in the testsuite. The results are given in Table 7.1. Preliminary to the execution of the back-ends we generated both the untyped and the typed version of the IF specifications by means of the HLPSL2IF translator. The OFMC back-end and the CL back-end are run against the untyped and the typed IF specifications of each protocol. The kind of the attack found (if any) and the time spent by the back-end are given in the corresponding columns.[1] Note that the analysis of the untyped and typed IF specifications may lead to the detection of different kinds of attacks. When this is the case, in Table 7.1 we report the two attacks found. Since, as we discussed in Section 6.3.5, the SATMC is not suited to analyze the untyped IF specifications, we applied it to the typed IF specifications only. As a consequence, we did not apply SATMC on protocols suffering from type flaw attacks.

[1]Times are obtained by running our back-ends on a PC with a 1.4 GHz Pentium III Processor and 512 Mb of RAM.

Table 7.1: Performance of the back-ends over the testsuite

Protocol	**Attack**	**OFMC**	**CL**	**SATMC**
ISO symmetric key 1-pass unilateral authentication	Replay	0.01	1.98	0.18/0.00
ISO symmetric key 2-pass mutual authentication	Replay	0.01	3.86	0.43/0.01
Andrew Secure RPC Protocol	Type flaw	0.03	4.26	NA
	Replay	0.05	32.74	80.57/2.65
ISO CCF 1-pass unilateral authentication	Replay	0.00	2.23	0.17/0.00
ISO CCF 2-pass mutual authentication	Replay	0.01	4.55	0.46/0.01
Needham-Schroeder Conventional Key	Replay STS	0.31	63.43	29.25/0.39
Denning-Sacco (symmetric)	Type flaw	0.02	15.98	NA
Otway-Rees	Type flaw	0.02	10.71	NA
Yahalom with Lowe's alteration	Type flaw	0.02	44.08	NA
Woo-Lam Π_1	Type flaw	0.01	0.81	NA
Woo-Lam Π_2	Type flaw	0.01	0.80	NA
Woo-Lam Π_3	Type flaw	0.00	0.82	NA
Woo-Lam Π	Parallel session	0.24	1074.95	3.31/0.04
Woo-Lam Mutual Authentication	Parallel session	0.27	245.56	1024.08/7.95
Needham-Schroeder Signature protocol	Man-in-the-middle	0.13	53.88	FF
**Neuman Stubblebine initial part*	Type flaw	0.03	6.19	NA
**Neuman Stubblebine repeated part*	Replay STS	0.03	3.54	15.17/0.21
Neuman Stubblebine (complete)	Type flaw	0.04	46.78	NA
Kehne Langendorfer Schoenwalder (repeated part)	Parallel session	0.20	199.43	MO/-
Kao Chow Repeated Authentication, 1	Replay STS	0.45	76.82	16.34/0.17
Kao Chow Repeated Authentication, 2	Replay STS	0.48	45.25	339.70/2.11
Kao Chow Repeated Authentication, 3	Replay STS	0.49	50.09	1288/MO
ISO public key 1-pass unilateral authentication	Replay	0.02	4.23	0.32/0.00
ISO public key 2-pass mutual authentication	Replay	0.01	11.06	1.18/0.01
**Needham-Schroeder Public Key*	Man-in-the-middle	0.04	12.91	1.77/0.05
Needham-Schroeder Public Key with key server	Man-in-the-middle	1.12	TO	4.29/0.04
**Needham-Schroeder with Lowe's fix*	Type flaw	0.01	31.12	NA
SPLICE/AS Authentication Protocol	Replay	4.02	352.42	5.48/0.05
Hwang and Chen's modified SPLICE	Man-in-the-middle	0.02	13.10	NS
Denning Sacco Key Distribution with Public Key	Man-in-the-middle	0.52	936.90	NS
Shamir Rivest Adelman Three Pass Protocol	Type flaw	0.03	0.70	NA
Encrypted Key Exchange	Parallel session	0.10	240.77	75.39/1.78
Davis Swick Private Key Certificates, protocol 1	Type flaw	0.05	106.15	NA
	Replay	1.19	TO	1.37/0.02
Davis Swick Private Key Certificates, protocol 2	Type flaw	0.16	348.49	NA
	Replay	0.86	TO	2.68/0.03
Davis Swick Private Key Certificates, protocol 3	Replay	0.03	2.68	1.50/0.02
Davis Swick Private Key Certificates, protocol 4	Replay	0.04	35.97	8.18/0.13

Legenda:		
	TO: Time Out	MO: Memory Out
	NA: Not Attempted	FF: Finite Failure
	NS: Not Supported	Replay STS: Replay attack based on a Short-Term Secret

The parameter settings of SATMC used in the experiments are those described in Section 6.3.5 for all protocols in the testsuite. For SATMC we give a pair of values T_e/T_s, where T_e is the *encoding time*, i.e. the time spent to generate the propositional formula, and T_s is the *search time*, i.e. the time spent by the SAT solver to check the formula (the SAT timings are obtained using the Chaff solver [37].) The labels TO and MO indicate a failure to analyze the protocol due to time-out and memory-out, respectively.[2] The label FF indicates a finite failure (this happens only to the SATMC on the *Needham-Schroeder Signature protocol*), whereas the label NS indicates that the back-end does not support some of the features occurring in the corresponding IF specification (this happens to the SATMC on the *Hwang and Chen's modified SPLICE* protocol and the *Denning Sacco Key Distribution with Public Key* protocol since key-tables are not yet supported by this back-end). Finally the label NA indicates the problem has not been attempted.

The experimental results clearly indicate that also the effectiveness and the performance criteria are met:

- The AVISS tool detects attacks on all 32 flawed protocols that can be modeled, i.e. it detects attacks on 91% of the 35 protocols in the Clark/Jacob library known to be flawed.
- The AVISS tool detects these attacks in a few seconds.

Table 7.1 allows us also to analyze the performance of the individual back-ends:

OFMC: The OFMC model-checker performs uniformly well on all the protocols: most of the attacks are found in a fraction of a second, and detecting all the attacks requires a total time of less than one minute. It is immediate to see that the OFMC tool achieves alone both the effectiveness and the performance criteria set in the proposal for the whole tool.

CL: The poorer timings of the CL model-checker are balanced by the fact that it is based on an off-the-shelf prover (daTac) and it offers other advantages such as the simple integration of algebraic relations on message constructors (e.g. commutativity of encryptions in RSA). In any case, it must be noticed that also this back-end achieves alone both the effectiveness and the performance criteria set in the proposal for the whole tool.

SATMC: The experiments show that the time spent to generate the SAT formula largely dominates the time spent to check the satisfiability of the SAT instance. Nevertheless, in many cases the overall timing is not too far from that of the OFMC back-end and it is better than that of the CL back-end. It is also interesting to observe that in many cases the time spent by the SAT solver is smaller than the time spent by the OFMC back-end for the same protocol.

[2] We set a time limit of 1 hour for each attempt. Due to a limitation of SICStus Prolog the SAT-based model-checker is bound to use 128Mb during the encoding generation.

8. Related work

The current generation of formal method tools for security protocols are mostly based on either interactive verification or automatic finite-state model-checking [5,7,8,10–12,14,17,19,20,22,27,28, 30,32,34–36,39,43]. The interactive tools require a considerable investment of time by expert users and provide no support for error detection when the protocols are flawed. The automatic tools generally require strong assumptions that bound the information analyzed, so that an infinite-state system can be approximated by a finite-state one. Moreover, the current level of automated support scales poorly and is insufficient for the validation of realistic protocols. Finally, tuning the specification (such as building relevant approximations) and tuning the tool's parameters often requires significant expertise too.

There are a number of tools that provide features similar to ours, and we discuss the most relevant ones below. To our knowledge, though, only the CAPSL [15, 20] environment, developed at SRI International (Menlo Park, CA) in the context of a DARPA-project, is designed to support multiple analysis techniques.

Specifications of authentication protocols written in CAPSL (an acronym standing for "Common Authentication Protocol Specification Language") are compiled to an intermediate formalism CIL, which may be converted to an input for analysis tools. In [20] methods are given for integration of CAPSL/CIL with SRI's own PVS [42] and Maude [33], with the special-purpose NRL protocol analyzer [13], and with the automatic verifier Athena [48].

The CAPSL environment and the AVISS tool share the idea of using both a high-level specification language close to the language used in text-books and by engineers (in fact, our HLPSL was inspired by CAPSL), and a low-level language that provides an interface to different inference engines. There are, however, several important differences. First, the CAPSL translator does not generate attacker rules, whereas we are able to produce IF rules from the specification of the intruder behavior. Second, we test a more general notion of executability (useful for e-commerce protocols such as non-repudiation protocols). CAPSL is unable to handle protocols where a principal receives a message that he cannot decrypt, say $\{Na\}_K$, and later receives the symmetric key K with which he can decrypt $\{Na\}_K$ and thus use Na in some later message. In our case, the principal will store $\{Na\}_K$ and will decrypt it when he later receives the key. Finally, based on the available experiments (cf. our experiments in Table 7.1 and the results in [15], as well as the comparison [8] on the NSPK protocol of CAPSL-Maude and of a predecessor of the OFMC), the AVISS tool and its back-ends are considerably more effective on the Clark/Jacob library than CAPSL and its current connectors.

We have presented the AVISS tool to the main developers of the CAPSL environment, Grit Denker and Jonathan Millen from SRI International, who visited the Freiburg group in September 2001 before attending the Dagstuhl seminar "Specification and Analysis of Secure Cryptographic Protocols" (which was co-organized by David Basin and attended by several members of the AVISS project; see Section 9). They have expressed their interest in our project and have suggested possible collaborations.

Note also that ease of tool integration was an important design consideration for the AVISS tool as well; the IF already provides a specification language that allows for the integration of other existing tools as further back-ends into the AVISS architecture. We have begun to investigate in this direction, and the first results are very promising. This will not, however, be a trivial task: we expect that several of the other tools, especially the ones that have a method-specific model (e.g., as we discuss below, a model such as the one of Casper/FDR2 and other finite-state model-checkers, or a model based on different goals as in the case of TAPS), will require us to extend the features of the IF and possibly change some of its mechanisms. Further investigations of the required general extensions and changes of the IF will be subject of future work.

Gavin Lowe and his group at the University of Leicester (UK) have analyzed the Clark/Jacob library using Casper [22, 32], a compiler that maps protocol specifications, written in a high-level language similar to HLPSL, into protocol descriptions in the process algebra CSP [1], where CSP processes are associated to each honest agent and to the intruder, and a finite number of roles are combined in parallel to form a system. The approach is oriented towards finite-state verification by model-checking with FDR2 [2], and thus requires a number of strong assumptions for bounding information (necessary to get a finite number of states). Lowe's approach using Casper/FDR2 has been very successfull for discovering new flaws in classical protocols: among the protocols of the Clark/Jacob library it has found attacks on 20 of the 25 previously known to be insecure, and it has also found attacks on 10 other protocols originally reported as secure. First experiments on the search time indicate, however, that the AVISS tool is considerably more effective than Casper/FDR2. Moreover, our tool has a number of other advantages. First, Casper/FDR2 inherently limits the size of messages that are sent in the network. This restriction is not needed for finite session cases since the existence of an attack is decidable without this restriction, and our experiments have shown that relaxing this condition does not limit efficiency. Second, as pointed out by Lowe himself, the Casper/FDR2 is rather weak for finding type flaws, as it can only find flaws that are "simple, in the sense that a single atom of one type is interpreted as being an atom of a different type, but cannot find compound type flaws where the concatenation of several atoms is interpreted as being a single atom, or vice versa" [32, p. 27]. Finally, it does not support non-atomic keys, which is a real drawback for analyzing e-commerce protocols.

Another tool that has been successfully applied to protocols in the Clark/Jacob library is the automatic verifier TAPS [19], developed by Ernie Cohen at Telcordia Technologies. TAPS is built on top of SPASS, an automated theorem prover for first-order logic with equality, which had already been applied by Weidenbach [51] to reason about security protocols. The goals of TAPS and the AVISS tool are opposite ones: while the AVISS tool tries to prove the incorrectness of protocols searching for flaws, TAPS tries to prove the correctness of protocols. In the case that a protocol is flawed, TAPS can only provide a hint of what the flaw could be, while in the case that a protocol is correct, the AVISS tool can only ensure that no flaw occurs for the finite number of sessions that is analyzed. In other words, the AVISS tool can constitute a decision procedure for the safety of protocols in case of finite sessions. The methods employed for achieving these goals are also quite complementary: while TAPS tries to construct first-order invariants that are not violated by any sequence of events, the deduction approaches of the AVISS project explore, with different strategies, the infinite state-space introduced by the protocol, trying to find a flaw-state. Experiments on the protocols in the Clark/Jacob library show that both tools are very successfull in their domain: TAPS can verify most of the protocols in the library for which no flaw is known, AVISS can falsify most of the protocols in the library for which a flaw is known. The combination of TAPS and AVISS, hence, yields an almost total coverage of the library. The connection of the two tools is therefore an interesting research line which we plan to investigate in the future. However it is worth noticing that the connection of TAPS with the IF is not trivial, as TAPS is based on different kinds of goals and on a different model of the problem.

The safety properties proved by TAPS are roughly equivalent to those considered by Paulson's interactive verifications of security protocols using Isabelle/HOL, an instantiation of the generic theorem prover Isabelle to higher-order logic [38, 39]. Paulson's pioneering verification approach is based on inductive definitions of sets of traces modeling all the possible communications between the agents as well as the attacker's interference. It has been applied to reason interactively about a number of protocols in the Clark/Jacob library such as the NSPK and the Yahalom protocol [41], as well as on some commercially deployed security protocols such as SET [9] and TLS [40]. Paulson's interleaving trace-based semantics method inspired the model underlying the predecessor [7] of the on-the-fly model-checking technique.

Besides for the already mentioned mechanization of rewriting logic Maude and for FDR2 (which, in conjunction with CSP, has also been employed in other approaches, e.g. [43, 44]), a number of other model-checking systems have been employed for the analysis of security protocols. For example, in [30], Leduc and Germau illustrate how to employ the ISO-standardized formal language LOTOS to specify security protocols and cryptographic operations, and how a model-based verification method can then be used to verify the robustness of a protocol against attacks. The LOTOS language comprises CSP as one of its parts and this approach is closely related to Lowe's, and like that and other model-checking approaches, is limited to finite sets of messages.

We conclude this overview of relevant related work by discussing the recently proposed Athena approach [48], which combines model checking and theorem proving techniques with the strand space model (used also in [16, 24]) to reduce the search space and automatically prove the correctness of security protocols with arbitrary number of concurrent runs. The method used in Athena has the main advantage of not requiring the declaration of instances of principals, and can handle an unbounded number of principals, which, however, implies the search procedure may not terminate. Like for Casper/FDR2, this approach is not well-suited for analyzing e-commerce protocols such as SET, as it based on atomic keys. Moreover, Athena is not able to handle type flaws. It is also worth pointing out that Athena is part of AGVI, a prototype toolkit [49] for the generation, verification, and implementation of security protocols. As we discuss in the next, concluding section, we also plan to investigate how to exploit our specification languages and analysis techniques to automatically generate and test code implementing the security protocols considered.

9. Conclusions and outlook

The previous sections document the success of our assessment project: our AVISS tool more than satisfies the project objectives, according to the success criteria adopted as a measure of the assessment. This is summarized in Table 1.1, which we repeat here.

Success criteria		**Objectives**	**Results**
Coverage	(number of protocols specified)	80%	90%
Effectiveness	(number of flaws detected)	70%	91 %
Performance	(CPU time)	$<$ 1 hour on 70%	$<$ 1 min on 91%

The main reason for this great success has been, we believe, the intensive collaboration between the groups, which resulted in a cross-fertilization of the different approaches and a combination of different techniques and ideas. This fact is witnessed by the optimizations we jointly employed, i.e.

(i) the lazy intruder approach, which provides an elegant and efficient model of the intruder behavior, and

(ii) the step-compression method, in which atomic events are merged to rule out many interleavings while preserving all attacks.

The first technique stems from the constraint-logic approach (it was developed in Nancy for the CL tool, but is successfully applied also in the on-the-fly model-checker OFMC). The second technique can be seen as a simple form of partial-order reduction which has been successfully applied in all three back-ends,

These techniques led to a remarkable improvement of the efficiency of our back-ends. They helped us reduce the search time by many orders of magnitude in some cases, or even allowed for the analysis of a larger number of protocols in the first place, as was the case for the adoption of the step-compression method in the SAT-based model-checker.

There is, however, still considerable room for improvement. For example, as we remarked above, step-compression is a "light-weight" form of partial-order reduction, and as future work we plan to develop a formal concept for "equivalent" states and in this way further reduce the state space, and hence increase the efficiency of our tool. In parallel to the tuning of our tools, we shall also investigate our model with formal means in order to obtain a tighter semantics for the IF, as discussed in Section 3. This will allow us to improve the efficiency of our inference engines, and it will also strengthen the formal basis for proving the correctness (and, possibly, completeness) of the new strategies we will implement in the back-ends. Moreover, it will allow for a deeper understanding of the differences with respect to other models and tools, such as the ones surveyed in Section 8, and will thereby provide a basis for the integration of other existing tools as further back-ends into the AVISS architecture.

We have already mentioned several other directions for future research throughout the report. We here summarize the main ideas, focusing in particular on how we plan to implement them in the full RTD project with industry involvement that we envision as a natural follow-up to this assessment project.

The main goal of this assessment was to investigate the potential of our proposed protocol analysis techniques and their combination into a single prototype tool based on a common specification language. The successful experimentation on the Clark/Jacob library [18] shows that our prototype is indeed ready to be turned into a mature technology for automatic error detection for e-commerce and related security protocols. In other words, further work on the AVISS prototype will lead to the development of a tool that will allow us to analyze large-scale, industrial protocols such as the SET [47] protocol, protocols in the GSM and UMTS families of telecommunication protocols, e.g. [3], as well as group protocols, multicast protocols, and protocols proposed by the Internet Engineering Task Force IETF [4]. We have already begun experimenting in this direction, and have, for example, been able to compile the card-holder registration phase of the SET protocol.

To this end, we have also initiated communication and collaboration with a number of academic and industrial partners — this will allow us not only to improve our tool but also to identify a set of representative case studies coming from the industrial practice on which to apply the results of the project. In particular, as reported in the project deliverable D1.2, representatives of all three project partners were among the academic and industrial participants at the Dagstuhl seminar "Specification and Analysis of Secure Cryptographic Protocols", co-organized by David Basin together with other leading researchers in the field of security protocol verification.[1] There, David Basin and Michaël Rusinowitch gave seminar talks presenting the work of the different partners and the prototype tool we have developed. Most importantly, during the seminar week, we had the opportunity of giving more in-depth demonstrations of the tool to a number of interested participants. In particular, Grit Denker and Jonathan Millen from SRI International have expressed their interest in our project and have suggested possible collaborations. So has Jorge Cuellar from the Research and Development Division of Siemens AG, who also collaborates with the IETF. Indeed, as a result of the seminar and of a subsequent visit to Freiburg by Dr. Cuellar, Siemens has expressed their interest in partaking in the follow-up RTD project.

In order to develop our techniques and prototype tool into a mature technology capable of analyzing industrial protocols such as the ones mentioned above, there are a number of interesting questions to consider, both for theoretical research and from a practical point of view.

For example, we would like to have a more flexible mechanism where the user can add further cryptographic operators and specify their desired properties, e.g. in order to handle efficiently protocols that employ Diffie-Helmann techniques. Hence the analysis method should account for algebraic laws that the operators satisfy in this setting. More work is also needed to handle group protocols with unbounded number of participants and key-hierarchies.

Usability is also an important criterion for the technology transfer of our tool. The graphical web-based interface we have designed integrates the different back-ends in a seamless way. It allows non-expert users of our prototype to specify protocols in a language that is close to the languages used in text-books and by engineers, and then apply the inference engines in a simple, push-button way. We plan to expand our current interface into a full-fledged graphical user interface that will allow engineers to apply our tool in the industrial setting. To this end, we would like, for example, to implement some form of graphical report generation such as animated message exchanges.

All the above directions for future research involve a "horizontal" extension of our prototype tool, widening its horizon of application to the industrial setting. Another challenging direction for future work is a "vertical" extension of our tool: we plan to investigate how to exploit our specification languages and analysis techniques to automatically generate code implementing the security protocols considered. Java seems to be an obvious choice for the implementation language, as it will allow us to exploit the standard Java packages for cryptography. The verification and formal testing of the code generated that way will then also be an interesting task.

[1] The seminar, which took place 23–28.9.01, was co-organized with Grit Denker and Jonathan Millen from SRI International Menlo Park and Gavin Lowe from Oxford University. It was attended by several members of the AVISS group, namely David Basin, Sebastian Mödersheim and Luca Viganò from Freiburg, Alessandro Armando from Genova, and Michaël Rusinowitch from Nancy. More information on the seminar can be found at `http://www.dagstuhl.de`.

Bibliography

[1] CSP — Communicating Sequential Processes. `http://www.formal.demon.co.uk/CSP.html`.

[2] FDR2 System — Failures-Divergence Refinement. `http://www.formal.demon.co.uk/CSP.html`.

[3] GSM World - the world wide web site of the GSM Association. `http://www.gsmworld.com`.

[4] IETF: The Internet Engineering Task Force. `http://www.ietf.org`.

[5] R. Accorsi, D. Basin, and L. Viganò. Towards an awareness-based semantics for security protocol analysis. In J. Goubault-Larrecq, editor, *Proceedings of LACPV'01*, ENTCS 55(1). Elsevier Science Publishers, Amsterdam, 2001.

[6] R. J. Anderson and R. Needham. Programming Satan's computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, LNCS 1000, pages 426–440. Springer-Verlag, 1995.

[7] D. Basin. Lazy infinite-state analysis of security protocols. In R. Baumgart, editor, *Secure Networking — CQRE'99*, LNCS 1740, pages 30–42. Springer-Verlag, 1999.

[8] D. Basin and G. Denker. Maude versus Haskell: an Experimental Comparison in Security Protocol Analysis. In K. Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Science Publishers, 2001.

[9] G. Bella, F. Massacci, L. C. Paulson, and P. Tramontano. Formal verification of cardholder registration in SET. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, editors, *Proceedings of the 6th European Symposium on Research in Computer Security: ESORICS 2000*, LNCS 1895, pages 159–174. Springer-Verlag, 2000.

[10] D. Bolignano. Towards the Formal Verification of Electronic Commerce Protocols. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 133–146. IEEE Computer Society Press, 1997.

[11] M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings of the 28th International Conference on Automata, Language and Programming: ICALP'01*, LNCS 2076, pages 667–681. Springer-Verlag, Berlin, 2001.

[12] S. Brackin. Evaluating and Improving Protocol Analysis by Automatic Proof. In *Proceedings of the 11th IEEE Computer Security Foundation Workshop*. IEEE Computer Society Press, 1998.

[13] S. Brackin, C. Meadows, and J. Millen. CAPSL Interface for the NRL Protocol Analyzer. In *Proceedings of ASSET'99, IEEE Symposium on Application-Specific Systems and Software Engineering Technology*. IEEE Computer Society Press, 1999.

[14] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.

[15] Common Authentication Protocol Specification Language. URL: `http://www.csl.sri.com/~millen/capsl/`.

[16] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *Proceedings of the 13th IEEE Computer Security Foundations Workshop: CSFW'00*, pages 35–51. IEEE Computer Society Press, 2000.

[17] I. Cervesato and P. F. Syverson. The logic of authentication protocols. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2171, pages 63–136. Springer-Verlag, 2001.

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

[19] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

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

[21] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 2(29), 1983.

[22] B. Donovan, P. Norris, and G. Lowe. Analyzing a Library of Security Protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, 1999.

[23] M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT-compilation of Planning Problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1169–1177. Morgan Kaufmann Publishers, 1997.

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

[25] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating Search Heuristics and Optimization Techniques in Propositional Satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of IJCAR'2001*, LNAI 2083, pages 347–363. Springer-Verlag, 2001.

[26] D. Gollmann. *Computer security.* John Wiley & Sons, 1999.

[27] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Narrowing Cryptographic Protocols. Technical Report 99-R-303, LORIA, Vandoeuvre les Nancy, Dec. 1999. URL: `www.loria.fr/equipes/protheo/SOFTWARES/CASRUL`.

[28] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and Verifying Security Protocols. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR 2000*, LNCS 1955, pages 131–160. Springer-Verlag, 2000.

[29] H. Kautz, H. McAllester, and B. Selman. Encoding Plans in Propositional Logic. In L. C. Aiello, j. Doyle, and S. Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 374–384. Kaufmann, M., 1996.

[30] G. Leduc and F. Germeau. Verification of Security Protocols using LOTOS – Method and Application. *Computer Communications, special issue on "Formal Description Techniques in Practice*, 23(12):1089–1103, 2000.

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

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

[33] The Maude System. URL: `http://maude.csl.sri.com`.

[34] C. Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26(2):113–131, 1996. See also `http://chacs.nrl.navy.mil/projects/crypto.html`.

[35] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *Proceedings of the DARPA Information and Survivability Conference and Exposition: DISCEX 2000*, pages 237–250. IEEE Computer Society Press, January 2000.

[36] J. C. Mitchell, M. Mitchell, and U. Stern. Automated Analysis of Cryptographic Protocols Using Murphi. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 141–153, 1997.

[37] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[38] L. C. Paulson. *Isabelle: a Generic Theorem Prover.* LNCS 828. Springer-Verlag, 1994.

[39] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[40] L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Computer and System Security*, 2(3):332–351, 1999.

[41] L. C. Paulson. Relations between secrets: The Yahalom protocol. In B. Christianson, B. Crispo, J. A. Malcolm, and M. Roe, editors, *Proceedings of the 7th Cambridge International Workshop on Security Protocols*, LNCS 1796, pages 73–77. Springer-Verlag, 1999.

[42] The PVS Specification and Verification System. URL: `http://pvs.csl.sri.com`.

[43] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop CSFW'95*, pages 98–107. IEEE Computer Society Press, 1995.

[44] A. W. Roscoe and M. Goldsmith. The perfect "spy" for model-checking cryptoprotocols. In *Proceeding of DIMACS Workshop on Design and Formal Verification of Crypto Protocols*, 1997.

[45] M. Rusinowitch and L. Vigneron. Automated Deduction with Associative-Commutative Operators. *Applicable Algebra in Engineering, Communication and Computation*, 6(1):23–56, Jan. 1995.

[46] B. Schneier. *Applied Cryptography.* John Wiley & Sons, New York, 1996.

[47] SET – Secure Electronic Transaction LLC. `http://www.setco.org/`.

[48] D. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9:47–74, 2001.

[49] D. Song, A. Perrig, and D. Phan. AGVI — Automatic Generation, Verification, and Implementation of Security Protocols. In *Proceedings of the 13th Conference on Computer Aided Verification CAV'01*, LNCS 2102, pages 241–245. Springer-Verlag, 2001.

[50] L. Vigneron. Associative-Commutative Deduction with Constraints. In A. Bundy, editor, *Proceedings of CADE'94*, LNCS 814, pages 530–544. Springer-Verlag, 1994.

[51] C. Weidenbach. Towards an Automatic Analysis of Security Protocols. In H. Ganzinger, editor, *Proceedings of CADE'99*, LNCS 1632, pages 378–382. Springer-Verlag, 1999.

[52] H. Zhang. SATO: An Efficient Propositional Prover. In W. McCune, editor, *Proceedings of CADE 14*, LNAI 1249, pages 272–275. Springer-Verlag, 1997.