www.avispa-project.org

IST-2001-39252

Automated Validation of Internet Security Protocols and Applications

Deliverable 4.5: AVISPA Tool v.2

Abstract

We describe version 2 of the AVISPA Tool for security protocol analysis, focussing in particular on the modifications and improvements with respect to version 1 of the tool. Besides the three original back-ends of the tool, namely OFMC, CL-AtSe, and SATMC, we describe a new back-end, TA4SP, which has been developed by the CASSIS group at INRIA and integrated into the current version of the AVISPA Tool.

Deliverable details

Deliverable version: *v1.0*
Date of delivery: *31.07.2004*
Classification: *public*

Person-months required: *10*
Due on: *31.07.2004*
Total pages: *25*

Project details

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

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

Contents

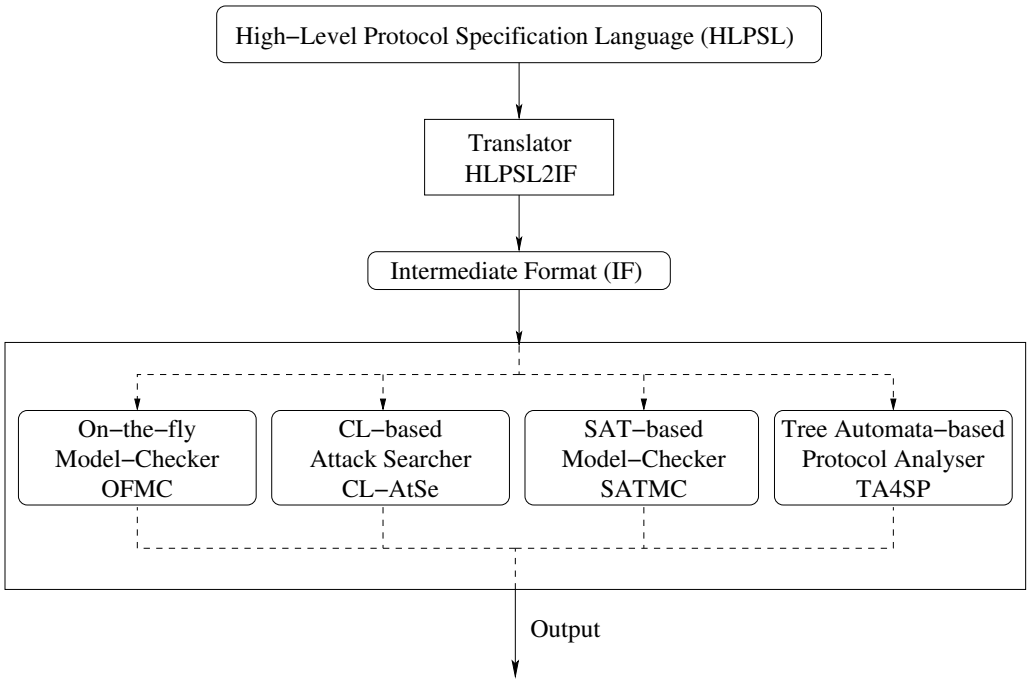

Figure 1: Architecture of the AVISPA Tool v.2

1 Introduction

The architecture of version 2 of the AVISPA Tool for security protocol analysis is depicted in Figure 1. Specifications of security protocols and properties written in the High-Level Protocol Specification Language (HLPSL [5]) are automatically translated (by the translator HLPSL2IF) into IF [7] specifications, which are then given as input to the different back-ends of the AVISPA Tool: OFMC, CL-AtSe, SATMC, and the new back-end TA4SP, developed by the LIFC group at INRIA. Whenever it terminates, each back-end of the AVISPA Tool outputs the result of its analysis using a common and precisely defined format stating whether the input problem was solved (positively or negatively), some of the system resources were exhausted, or the problem was not tackled by the required back-end for some reason.

This report is organised in the following way. In Section 2 we consider the HLPSL2IF translator. Section 3 is devoted to the back-ends of the tool. We give a detailed description of the new back-end TA4SP, while for the three original back-ends OFMC, CL-AtSe, and SATMC, as well as for the HLPSL2IF translator, here we only describe the main modifications and extensions that have been carried out with respect to version 1 of the AVISPA Tool. It is worth pointing out that the AVISPA Tool v.2 will be thor-

oughly assessed in a subsequent deliverable [15] by testing the tool against the AVISPA library of Deliverable 6.1 [12], a library of security problems drawn from the protocols developed by the IETF. In this deliverable [15], we will also report on the new version of the Output Format OF that we are working on. Here, we summarise the main features of the current version of the OF, pointing to Deliverable 4.4 [10] for formal definitions and examples.

2 The HLPSL2IF Translator

The rôle of the HPLSL2IF translator is to automatically translate high-level protocol specifications (written in HLPSL) to low-level specifications (in the Intermediate Format, IF). The HLPSL is a role-based language, designed to be easily readable and writable, that permits one to describe protocols in a precise and formal way; the semantics of this language is based on a fragment of Lamport's Temporal Logic of Actions, TLA [26].

HLPSL supports security protocol specification at a high level of abstraction and has many features that are common to most protocol specifications built-in, such as intruder models and encryption primitives. In contrast, the IF is a lower-level language at an accordingly lower abstraction level; that is, the IF is a tool-independent protocol specification language suitable for automated deduction (so that specifications in the IF can be given as input to the analysis back-ends of the AVISPA tool).

Further information on HLPSL and IF, including their syntax and semantics, can be found in the Deliverable 2.1 [5] and the paper [19], and in the Deliverable 2.3 [7].

We have upgraded our translator from HLPSL to IF according to the recent language extensions. The major improvements concern:

Compound types: variables in HLPSL can be declared to have a compound type, e.g. `agent.symmetric_key` (see Section 2 in Deliverable 3.2 [13]). Such types are translated into IF compound types (e.g. `pair(agent, symmetric_key)`) and, when required, variables are initialised appropriately (e.g. `pair(dummy_agent,dummy_sk)`).

Conditional transitions: transitions in HLPSL need sometimes to be converted into conditional transitions in the intermediate format. This was not handled by the previous version of the translator. Now, IF specifications may contain equational conditions, inequalities, and negations of facts.

In addition to those improvements related to language extensions, some new features have been included in the translator. For example, by default,

the generated IF specification is written in a file instead of to the standard output. However, the user can also ask for printing the IF on the standard output. Moreover, as for a given protocol there are often several security problems to analyse, the user can ask for generating one IF file per problem.

The translator is invoked by typing:

```
hlpsl2if [options] [inputFile.hlpsl]
```

where `options` is a combination of:

`--types`	for printing identifiers and their types,
`--init`	for printing the initial state,
`--rules`	for printing the protocol rules,
`--goals`	for printing the goals,
`--all`	for printing everything (default),
`--stdout`	for printing on the standard output,
`--split`	for generating one IF file per problem,
`-help`	for printing this list of options, and
`--help`	for printing this list of options.

The translator is freely available on the AVISPA web page (`http://www.avispa-project.org`), in the *software* section.

3 The Back-Ends of the AVISPA Tool v.2

In this section, we describe the back-ends of the AVISPA Tool v.2. We describe the three original back-ends, presenting, for each of them, the approach on which the back-end is based, the implementation and the advancements with respect to AVISPA Tool v.1, and we describe the new back-end TA4SP.

Note that in the following we will refer to security protocols *without loops* to indicate security protocols where a single execution of the protocol cannot involve an unbounded number of steps of any role. Similarly, a security problem without loops is a security problem whose associated security protocol is without loops (recall that a *security problem* is given by both a protocol and a security property that the protocol should satisfy).

3.1 OFMC

The On-the-Fly Model-Checker OFMC builds the infinite tree defined by the protocol analysis problem in a demand-driven way, i.e. on-the-fly, hence the name of the back-end.

As shown in Figure 2, OFMC consists of two modules, OFMC-core and OFMC-FP: OFMC-core searches reachable states (in the model corresponding to the input problem) for flaws, while OFMC-FP tries to verify the protocol by computing the fixed-point of an abstract model (see Deliverables 5.1 [11] and 5.2 [14] for more details). Both modules share common basic algorithms and data structures, such as unification of terms modulo algebraic properties (see Deliverable 2.2 [6]).

The IF2OFMC translator reads an IF file as input and returns the initial state, rules and goals in appropriate Haskell data structures (Haskell is the implementation language of OFMC). The session compilation heuristics (SessCo) then takes the initial state and rules from the translator and augments the initial state with additional messages the intruder knows. We are currently migrating the session compilation heuristics so that it can be applied by all the back-ends (see Deliverable 4.3 [9] for more details on this heuristics).

As discussed in more detail in [16, 17], OFMC-core transforms the rules given by IF2OFMC into a successor relation of symbolic states. OFMC-core has two layers. The lower layer deals with the constraints of the symbolic lazy intruder technique, analysing intruder knowledge or checking that the intruder can generate certain terms (shown on the right of the OFMC-core module in Figure 2). The higher layer uses the lower layer to build the symbolic (and "step-compressed", cf. [8, 16, 17]) transition relation that consists of the intruder forging a message that can be generated from the current intruder knowledge and intercepting the answer that can then be analysed (shown on the left of the OFMC-core module in Figure 2).

The successor relation computed by OFMC-core is used to build a tree from the extended initial state provided by the session compilation heuristics and to check for violations of the goals. Since OFMC-core and SessCo are correct, i.e. they never introduce false attacks, every attack found by this search is really an attack and can therefore be directly reported to the user (this is in contrast with attacks found by OFMC-FP, which, due to over-abstraction, may be false attacks). Note that, for a bounded number of sessions, OFMC-core is also complete: that is, the symbolic search space is finite and OFMC-core can give the answer to the user that the protocol is secure in the given scenario.

The OFMC-FP module also works on the initial state and rules provided by IF2OFMC, and it has a preprocessing phase, IF2FP, which performs the abstraction of fresh messages described in detail in Deliverables 5.1 [11] and 5.2 [14]. (Note that session compilation is redundant in this case.) On these abstract rules, OFMC-FP computes the fix-point of reachable facts representing an over-approximation of the reachable states and the associated

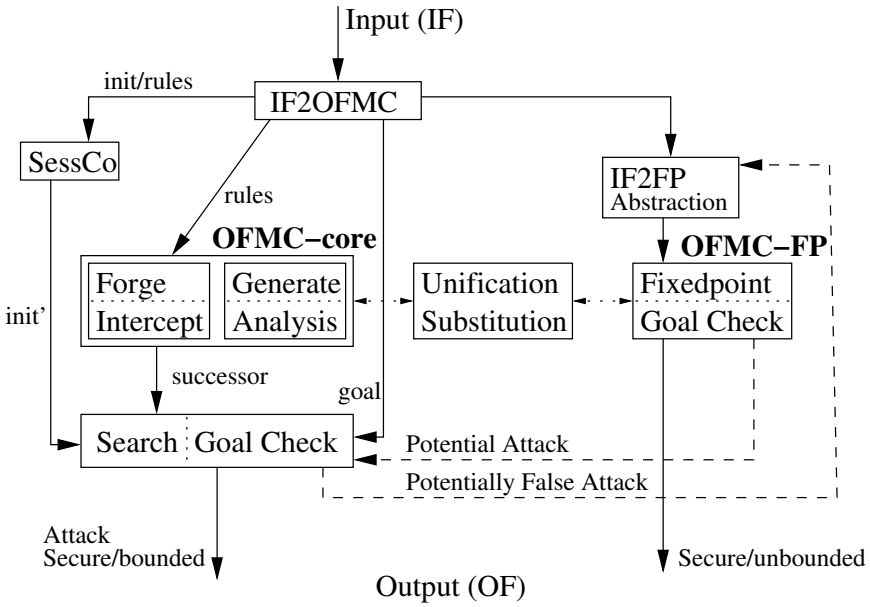

Figure 2: Architecture of OFMC

messages derivable by the intruder. The goal state is different from the one of OFMC-core since it is based on a slightly different model. If the goal check fails for all reachable facts, then we can be sure the protocol is correct and report this to the user. When the goal check fires for some fact, then we have found an attack in the abstract model, which, due to the use of abstraction techniques, could be a false attack, i.e. there may be no corresponding attack in the original model. The (abstract) trace that leads to the attack is fed into OFMC-core as a heuristics, to see if OFMC-core can find a corresponding attack (i.e. one that, under the abstraction, maps to the one in the abstract model). If OFMC-core does not find an attack, then it was probably a false attack and OFMC-FP is started again with a finer abstraction; that is, an abstraction allowing more fresh messages (see Deliverable 5.1 [11]). (Note that the "Potential Attack" and "Potentially False Attack" arrows are dashed to symbolise that these procedures are still under development.)

OFMC-core is invoked by typing on the command-line

```
Ofmc <filename> [-sessco] [-d <number>] [-p <number>*]
```

where all options are for debugging purposes only:

- `<filename>` is an IF file to be checked.
- When the option `-sessco` is given, OFMC first performs an executability check and session compilation (see the discussion below).

- Using the `-d` option one can specify a depth bound for the search (the default being unbounded depth). In this case, OFMC uses a depth-first search (while the standard search strategy is a combination of breadth-first search and iterative deepening search).
- Using the `-p` option, one can "manually browse" the search tree, e.g.:

  `-p` is the root node,

  `-p 0` is the first (left-most) successor of the root node,

  `-p 0 1` is the second successor (next to left-most) successor of the node obtained by `-p 0`.

  An exception is raised if a path to a non-existing node is specified.

With respect to AVISPA Tool v.1 [10], OFMC-core and OFMC-FP have undergone a number of improvements:

Session Compilation and Executability Check. A new option of OFMC-core is the session compilation option. When specified, OFMC-core will first perform a search with a passive intruder to check whether the honest agents can execute the protocol, and then give the intruder the knowledge of some "normal" sessions between honest agents.[1] In the case certain steps cannot be executed by any honest agent, OFMC reports this to the user and stops. We have detected and corrected several minor errors in protocol specifications that led to non-executability of certain transitions (and attacks involving these transitions would have been missed without the corrections). If the executability check is successful, then the normal search with an active intruder is started, with the only difference that he initially knows all the messages exchanged by the honest agents in the passive intruder phase. A more detailed description of the session compilation technique can be found in Deliverable 4.3 [9]. As stated, work is underway to migrate this functionality in order to make it available to all the back-ends of the AVISPA tool.

Algebraic Properties of Cryptographic Operators. The previous version of the OFMC back-end was equipped with preliminary support for equations modelling the algebraic properties of cryptographic primitives. Such properties of cryptographic operators are used in a variety of protocols, e.g. those based on a Diffie-Hellman key-exchange. A

[1] Note that session compilation and executability check currently work only for protocols without loops.

large amount of theoretical research was performed here (see Deliverable 2.2 [6]), and the methods were integrated into the OFMC back-end. Note that these methods work uniformly both for OFMC-core and OFMC-FP. More details on these methods will be provided in a future deliverable.

Re-Engineering and Code Review. We have carried out a major re-engineering of the code of our back-end, in order not only to improve its efficiency but also to make it more accessible (and thereby simplify its maintenance as well as simplify the carrying out of future modifications). In particular, this review of the code has revealed a number of inefficiencies and even a few bugs.[2] Improving the code has lead to a considerable speed-up of the back-end, by roughly a factor of 5 (even though, of course, these minor code improvements do not change the overall complexity).

OFMC has undergone many improvements, as described above, and work continues to integrate new techniques developed within the AVISPA project. For instance, the handling of abstractions in OFMC-FP currently requires human intervention but is being improved with a view towards automating this feature. Our aim, through this and other improvements, is to make the back-end accessible to expert and non-expert users as well.

3.2 CL-AtSe

The CL-AtSe (Constraint Logic - Attack Searcher) back-end, whose structure is presented in Figure 3, is a direct implementation in Objective CAML of a constraint logic approach described in [20, 29]. All the information concerning the protocol to study is read from an IF specification file.

In the following, we will describe the main steps of how CL-AtSe carries out protocol analysis, focussing on the novel features that have been implemented with respect to the previous version of the back-end.

The first action of CL-AtSe is to interpret the input IF file, and to build a role-based representation of the protocol. In this representation, each variable's scope is *global* to the whole protocol and represents a participant piece of knowledge (or memory location). A role is a list of protocol steps, and

[2]This raises the interesting question of how to verify a verification procedure. For a falsification procedure one might argue that bugs are not so problematic: a bug that causes the tool to report false attacks will be detected as soon as it appears, and a bug that causes attacks to be missed is also not such a big problem for falsification if no completeness guarantees are given.

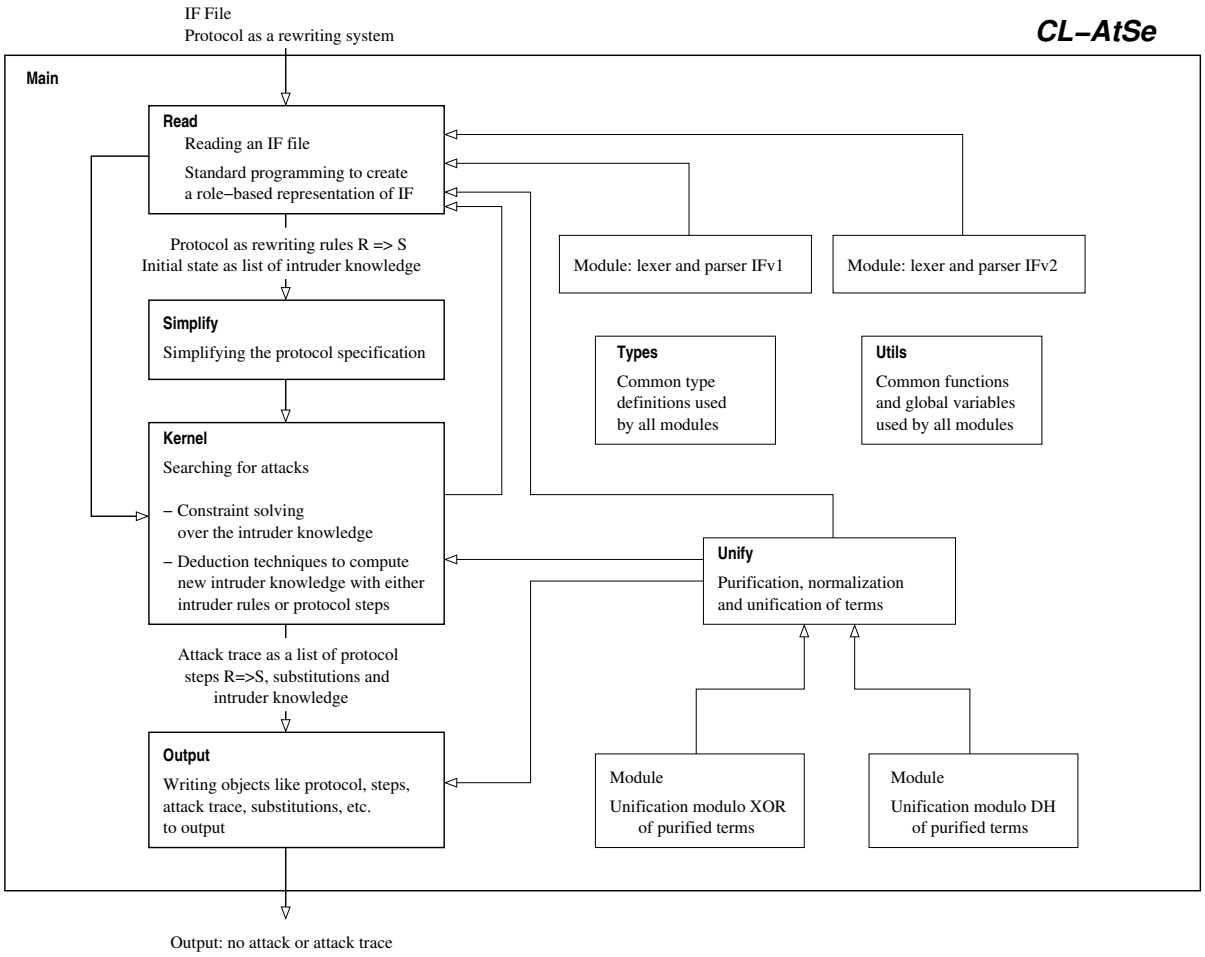

Figure 3: Architecture of CL-AtSe

each protocol step will have the following structure (we call it a CL-AtSe rule):

$$R \;\Rightarrow\; S \quad [u = v]\; Witness(..)\; Request(..)\; ...$$

where R and S are messages (similar to *iknows* facts denoting terms known by the intruder in IF specification language) and $[u = v]$ is a unification constraint. This first rewriting of the protocol is quite simple: from an initial state and an IF-rule read from the IF-file a CL-AtSe rule is produced. This process is repeated for the state associated to the right-hand side of the current IF-rule and the next IF-rule in the IF-file and so on. For example, when constructing the representation of some role "Alice":

- if the current state is $state\_Alice(t1)$, and

- if there is a rule $iknows(R).state\_Alice(t2) \Rightarrow iknows(S).state\_Alice(t3)$ (with renaming of variables to ensure unicity) such that $t1$ and $t2$ can be unified,

then the current state becomes $state\_Alice(t3)$ and we add the rule $R \Rightarrow S\ [t1 = t2]$ to the role "Alice". This means that we push the state facts into the unification constraints. The advantage is that we will be able to greatly simplify these unification constraints and protocol steps, and even sometimes merge several steps. Since the kernel of CL-AtSe will have to explore a search space which is exponential in the number of protocol steps, the simpler and fewer they are, the faster the search will be.

The simplification of the protocol is done in several steps:

Simplification of the unification constraints The role-based representation of a protocol constructed as described above often contains huge unification constraints. However, most of them are completely useless. Each rule $R \Rightarrow S\ [u = v]$ is simplified as follows:

- First use the unification algorithm with u and v. This produces a substitution σ representing all possible unifications of u and v.
- Second split σ into two parts:
  - ⋆ The trivial part σ' contains the variable instantiations that can be applied immediately on the protocol steps without altering the protocol specification.
  - ⋆ The non-trivial part σ'', represented as a unification constraint $[u' = v']$, contains all other cases.

In practice, more than 95% of the unification constraints (or state facts) are eliminated. The remaining constraints represent *matching tests* performed by participants on previously received messages. A typical example is the following role:

$$\begin{array}{llll} 1. & X & \Rightarrow & na \\ 2. & Z & \Rightarrow & Y \qquad [X = \{Y\}^s_Z] \end{array}$$

This role has a natural ordering (step 1 before step 2), and X, Y, and Z are variables, and na is an atom. In this example, the unification constraint over X *cannot* be eliminated, because in the first step the participant does not have the decryption key Z yet. Therefore, X can only be decrypted in the second step. Moreover, if the value of X does not have the correct structure, then the honest participant detects an attack and the rule cannot be executed. In fact, this kind of non-trivial unification constraint is currently the *only* kind of constraint that is not eliminated by CL-AtSe.

Deletion of protocol steps We now have a protocol specification using simple protocol steps. This gives us the possibility to easily identify useless steps and remove them (actually, we will merge them with neighbour steps in the role specification). Let us take an example. Consider the following role:

$$\begin{array}{llll} 1. & i & \Rightarrow & a \\ 2. & X & \Rightarrow & a \qquad \text{Witness(...)} \\ 3. & Y & \Rightarrow & \{X\}^s_Y \qquad \text{Request(...)} \end{array}$$

As before, this role is naturally ordered (step 1 before step 2 before step 3), X and Y are variables, and a is an atom. Assume that the initial intruder knowledge contains the atom i. Then it is obvious that the first step can be used by the intruder as soon as possible (no need to consider interleavings of steps here). And it is also obvious that the second step does not extend the intruder knowledge, and can therefore be used as late as possible (here too, we do not need to consider the interleaving of protocol steps). Therefore, the previous role becomes:

$$< X, Y > \quad \Rightarrow \quad \{X\}_Y \qquad \text{Witness(...)} \quad \text{Request(...)}$$

and the initial intruder knowledge is extended with a. This reasoning can be iterated: any list of consecutive useless rules are either merged together with a previous or following rule, or integrated into the initial intruder knowledge, or discarded. Since most of the time is lost in testing all interleavings of protocol steps, these simplifications can greatly accelerate the search for attacks. For example, in the case of the protocol CHAPv2 where the analysis time is 1.5 seconds in the non-simplified version and 0.01 second in the simplified case.

Searching for attacks. Once translated and simplified, the protocol specification is checked for attacks using constraint logic methods [20, 29]. That is, the system state (describing the states of the principals and the intruder) is symbolically represented by a set of constraints on the intruder knowledge (like $Na \in \texttt{forge}(t_1, .., t_n)$, i.e. the value of Na must be forged from the knowledge $t_1, .., t_n$) and a set of constraints on terms (like $Na = \texttt{pair}(a, b)$ or $Na \neq Nb$). All these constraints are decomposed into elementary ones for satisfiability testing. For each execution of a protocol step this symbolic system is updated to represent the new states of the principals and of the intruder, and the new conditions; this new system is tested for satisfiability of the security goals. By enumerating all the interleavings of protocol steps for a given depth, CL-AtSe determines whether a system state violating one

of the security properties can be reached or whether the protocol is secure with respect to the exploration depth considered.[3]

For protocols without loops, the number of protocol steps analysed by CL-AtSe is exactly the number of steps in the specification times the number of sessions. But for protocols with loops, CL-AtSe accepts a command line argument specifying an upper bound on number of steps to be analysed. This ensures termination, but completeness is not guaranteed.

In the attack search process CL-AtSe uses the Inits section of the IF file to get the initial protocol state. From this point on, the intruder will guide the protocol execution, firing transitions if he is able to compose and send a message that matches the left-hand side of a protocol rule. If in this process we reach a state where any of the properties are violated then an attack has been detected. This attack can be defined by a sequence of protocol states from the initial state. To show that this attack is a real one we only need to show that the intruder was able to correctly decompose the messages received from the other agents and to correctly compose the messages sent that led to the violated state. The proof of correctness of the intruder's abilities that can be found in [29] leads us to conclude that the attacks found by CL-AtSe are indeed real attacks.

A new feature introduced in this new version of the tool is the typed model analysis. In the untyped model the principals will accept messages without any kind of type checking, which allows the intruder to fool the agents by providing messages of an unexpected type; this gives rise to so-called *type-flaw* or *type-confusion* attacks. In the typed model, the types of the variables and constants given in the Types section of the IF file are used to prevent this kind of attack. This new feature is important as it permits one to easily see when the protocol suffers from a type-flaw attack.

To use CL-AtSe, one provides the name of the IF file to analyse and a sequence of options, like the depth bound if loops are used. CL-AtSe can be invoked by typing on the command-line:

```
cl-atse [options] [filename.if]
```

where the different options are:

`-nb n` bounds the search depth to n, limiting the maximum number of protocols steps that will be analysed in case of loops.

`-light` specifies that light unification should be used, i.e. no partial associativity for pairs.

[3]Note that in Figure 3 a "no attack" output refers to a scenario with a bounded number of sessions.

`-notype` forces the use of the untyped model, where the types of terms are not taken into account.

`-td` uses a depth-first approach to perform the search over the state space of the protocol (this is the default).

`-lr` uses a breadth-first approach to perform the search over the state space of the protocol.

`-out` indicates that the output should be `filename.atk` instead of the standard output.

`-ns` indicates that the IF file should not be simplified.

`-v` asks CL-AtSe to be verbose and to print as much information as possible. (Compliance with the output format OF is not ensured in this case.)

`-help` asks CL-AtSe to print help information listing all the available options.

Note that if the IF filename is not provided, the standard input is used instead for reading a IF specification.

CL-AtSe returns either:

- `YES`, when an attack has been found (and the trace of this attack is also given in output), or
- `NO`, when no attack has been found. If the protocol does not have loops, it means that it is secure in the analysed scenario; however, if the protocol has loops, the user has set a maximal depth, so the protocol has been partially studied and is secure up to this depth.

3.3 SATMC

SATMC performs an automatic translation of security protocols into propositional logic (SAT) thus exploiting the high performance of state-of-the-art SAT solvers for exploring the search-space.

The starting point of the SAT-based model-checking approach (also called bounded model-checking [18]) is that given a security problem Ξ expressed in IF and an integer k, it is possible to generate a propositional formula Φ^k_Ξ such that any model of Φ^k_Ξ corresponds to one or more attacks on Ξ. The encoding of Ξ into a SAT formula can be done in a variety of ways. The basic idea is to put an additional time-index parameter to each rule and fact

of the IF specification, to indicate the state at which the rule begins or the fact holds. Facts are thus indexed by 0 through k and rules by 0 through $k-1$. If p is a fact or a rule in the IF specification and i is an index in the appropriate range, then p_i is the corresponding time-indexed propositional variable. The SAT formula Φ^k_Ξ, i.e. a symbolic representation of the search space up to depth k, is built on top of these propositional variables and looks like:

$$\Phi^k_\Xi = I_0 \wedge \bigwedge_{i=0}^{k-1} T_i^{i+1} \wedge G_k,$$

where I_0, T_i^{i+1}, and G_k are the Boolean formulae representing the initial state, the transition relation between states reachable in i steps and states reachable in $i+1$ steps, and the goal states, respectively. The main differences between the SAT reduction encoding techniques are reflected in the formula that encodes the transition relation, and considerable effort has been put into devising the encoding technique that leads to propositional formulae of manageable size.

Three encoding techniques are currently implemented in SATMC: the first one belongs to the family of so-called *linear encodings* (see [2] for more details), while the second and the third techniques are different variants of the more sophisticated *graphplan-based encoding*. In more detail, the latter two techniques share the idea of constructing a graph data-structure representing an over-approximation of the forward search tree, but they use two different encoding schemes for translating such a graph data-structure into a propositional formula: the standard encoding schema based on the *backward chaining schema* and a new one recently devised by us and based on the *explanatory frame schema*. Experimental results [4] clearly indicate that graphplan-based encodings are superior to linear encodings on the domain of security protocol analysis. For what concerns a comparison between the two graphplan-based encodings, results indicate that, by using the new encoding technique based on the explanatory frame schema, we obtain *(i)* SAT instances whose sizes are up to 40% smaller, and *(ii)* up to 62% better encoding times with respect to the other encoding techniques based on the backward chaining schema.

With respect to AVISPA Tool v. 1 [10], SATMC has undergone a number of improvements:

Encoding. We have devised and implemented a new graphplan-based encoding technique that improves performance of SATMC (as described above).

Security Problem Verification. We have enhanced SATMC with a pro-

cedure able to verify an input security problem when some conditions are satisfied. In order to do this, we have extended the two graphplan-based encoding techniques with a module for checking if the *fixed-point* has been reached during the construction of the graph. Roughly speaking, the graph built at step i represents an over-approximation, in terms of reachable states and applicable rules, of the forward search tree up to depth i. To reach the fixed-point at step k means to have computed an over-approximation of the whole forward search tree associated with the input security problem Ξ. Notice that the unsatisfiability of the corresponding SAT formula Φ^k_Ξ is not sufficient to conclude that the input security problem is verified. In fact, an attack can require multiple executions of the same rule at different steps as well as the execution of rules in mutual exclusion. As a consequence, it could be that k steps are not enough to perform the attack. However, for security problems without loops, when we reach the fixed-point in the construction of the graph we have established that the input security problem is verified.

Compound Types. We have enhanced SATMC in order to support the *compound types* assumption described in Deliverable 3.2 [13]. Preliminary experimental results on the AAAMobileIP and Kerberos protocols show a significant improvement in the performance of SATMC.

Optimised Intruder. We have devised and implemented an *optimised intruder model* for SATMC based on the idea that some of the abilities of the intruder have instantaneous effect. Namely, by modelling the decomposition of the intruder knowledge by means of axioms[4] instead of rewrite-rules, SATMC is able to find out up to 40% shorter attacks (e.g. the attack to the Needham-Schroeder Public-Key Protocol is found in 4 steps instead of 7 steps) by generating up to 50% smaller propositional formulae. See [3] for more details.

The architecture of SATMC is shown in Figure 4. SATMC takes as input an IF specification representing a security problem, the prelude file, and the following parameters:

- `max`: maximum depth of the search space up to which SATMC will explore (the parameter `max` can be set to `-1` meaning *infinite*, but in this case the procedure is not guaranteed to terminate); by default it is set to `10`.

[4] An axiom is a formula that states a relation between some facts of the transition system and that holds in each state of the transition system.

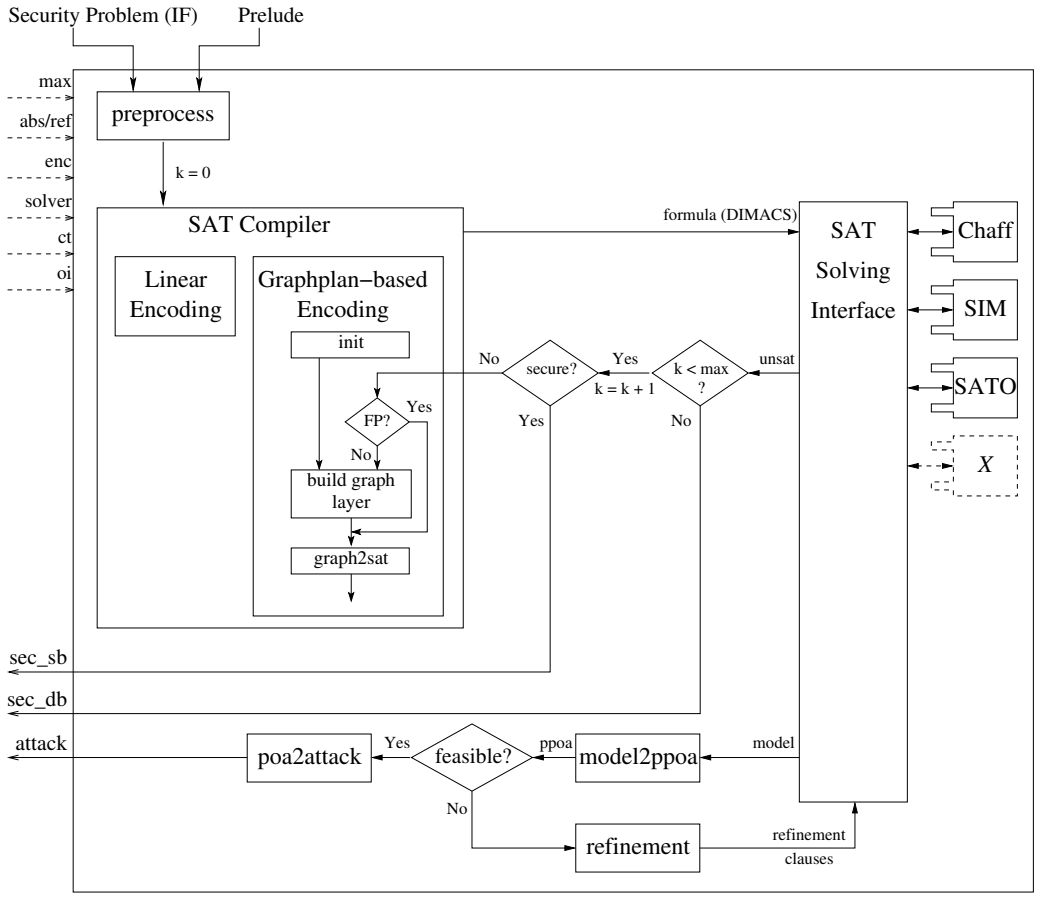

Figure 4: Architecture of SATMC

- `absRef`: a Boolean parameter for enabling or disabling the abstraction/refinement strategy the back-end provides (see [2] for more details); by default it is set to `false`.
- `encoding`: the selected SAT reduction encoding technique (currently implemented are the linear encoding [1] and two graphplan-based encodings, one using the backward chaining schema [4] and the other one applying the explanatory frame schema); it can be set to either `linear`, `graphplan` or `newgraphplan` (default value).
- `solver`: the selected state-of-the-art SAT solver (Chaff [27], SIM[25], and SATO [30] are currently supported); it ranges over the values `chaff` (default value), `sim`, and `sato`.
- `ct`: a Boolean parameter for enabling or disabling the *compound typing* assumption presented in Deliverable 3.2 [13]; by default it is set to `true`.

- `oi`: a Boolean parameter for enabling or disabling the *optimised intruder model* presented in [3]; by default it is set to `true`. Disabling such an option can be useful to experiment the effectiveness of the optimised intruder model.

SATMC then returns either:

- `attack`: stating that an attack has been discovered and is returned to the user in a syntax compliant with the Output Format presented in Section 4;
- `sec_db`: standing for "`sec`ure up to the `d`epth `b`ound" and stating that no attack has been found on the input security problem after having explored up to *max* depth levels of the search space; or
- `sec_sb`: standing for "`sec`ure up to the `s`ession `b`ound" and meaning that the input security problem does not suffer from any attack in the scenario analysed (bounded sessions) and under the assumptions of perfect cryptography, strong typing, intruder based on the Dolev-Yao model, and (if enabled) compound typing.

In more detail, SATMC starts preprocessing the IF specification in order to *(i)* generate protocol-specific intruder axioms and rules, and *(ii)* check whether the input protocol is without loops or not. After that, SATMC applies the selected encoding technique to the preprocessed IF specification for increasing values of k, and the propositional formula generated at each step is fed to the selected SAT solver. As soon as a satisfiable formula is found, the corresponding model is translated back into a "pseudo" partial-order attack (*ppoa*). A *partial-order attack* is a partially ordered set of IF rules leading from the initial state to a goal state.[5] If the abstraction/refinement strategy is disabled, we are guaranteed that a pseudo partial-order attack corresponds to a partial-order attack. Otherwise, as explained in [2], the pseudo partial-order attack has to be checked and if it corresponds to a spurious counterexample, a refinement phase is applied on the current propositional formula that is fed back to the SAT-solver. The whole procedure is iterated until a non-spurious counterexample is met or the formula becomes unsatisfiable. As soon as a partial-order attack is found, it is translated into attacks which are reported to the user according to the syntax specified in Section 4. If one of the graphplan-based encoding is selected and the following conditions hold *(i)* the analysed security problem is without loops, and *(ii)* the fixed-point (indicated by means of the test `FP?` in Figure 4) has been reached at step

[5] A partial-order attack concisely represents a set of attacks on the protocol.

$k \leq$ `max`, then the stop condition (indicated by means of the test `secure?` in Figure 4) is satisfied and SATMC returns `sec_sb` stating that in the scenario analysed the input security problem does not allow for any attack. Finally, if no partial-order attack is found and the stop condition is not satisfied (see below) up to `max` steps, then SATMC returns `sec_db` meaning that even if the input security problem has not been verified, the protocol does not suffer of any flaw after having explored up to `max` depth levels of the search space.

SATMC can be invoked by typing on the command-line

```
satmc <filename> --prelude=<fileprelude>
                 [--max=<number>] [--absRef=<bool>]
                 [--encoding=<encoding>] [--solver=<solver>]
                 [--ct=<bool>] [--oi=<bool>]
```

where `<filename>` and `<fileprelude>` are, respectively, the IF problem to be analysed and the prelude file, and each option is as described above.

3.4 TA4SP

This section describes work about the automatic verification of security protocols with an unbounded number of sessions that has been carried out by the CASSIS group at INRIA, and that has lead to the implementation of the new back-end TA4SP (Tree Automata based Automatic Approximations for the Analysis of Security Protocols). The approach that we investigate, initiated by Genet and Klay [23], is to over-estimate the intruder knowledge by using regular tree languages. This method allows one to show that some states are unreachable, and hence that the intruder will never be able to know certain terms. Regular tree-languages can be used here to effectively model the knowledge that the intruder might have acquired from previous sessions.

In this section, we first give a brief overview of the method underlying TA4SP and then describe how to use the back-end. From an IF-specification of a protocol, an input file for Timbuk2[6] [24] is automatically generated by our translator IF2Timbuk2.

This file contains:

(1) a representation of this protocol and the abilities of the intruder to analyse a message by a term rewriting system,

[6] An `OCAML` tree automata library developed by T. Genet at IRISA-Rennes under GPL. Notice that we use the second version of Timbuk (Timbuk2).

(2) an initial automaton representing the initial configuration of the network, and the initial knowledge of the intruder,

(3) a symbolic approximation function corresponding to the term rewriting system generated, and

(4) an automaton expressing the negation of secrecy properties to verify (currently, all properties are encoded in a single automaton).

Timbuk2 is a collection of tools for achieving proofs of reachability over term rewriting systems and for manipulating tree automata (bottom-up non-deterministic finite tree automata). As described in Deliverable 5.2 [14], the method provides, from the term-rewriting system corresponding to a protocol under consideration (1) and from the initial automaton (2), an automaton encoding an over-approximation of the intruder's knowledge (the number of sessions is unbounded). This procedure requires a symbolic approximation (3) to ensure the termination (see Deliverable 5.2 [14] for more details). Then we have just to check whether the over-approximation contains terms recognised by the automaton (4). This is done using standard algorithms on tree automata implemented in Timbuk2. If the intersection is empty, then all properties specified are verified, otherwise one cannot conclude.

The example below is one of the rules of the term rewriting system generated by the translator IF2Timbuk2 expressing the evolution of Alice's state during a run of the protocol NSPK [21, 28].

```
state_Alice(ag(A),ag(B),pk(KA),pk(Kb),dummy(Ni),X,msg(start))
->
state_Alice(ag(A),ag(B),pk(KA),pk(Kb),n(ag(A),ag(B),symbcst(t1)),X,
msg(crypt(pk(ag(B)),pair(N(ag(A),ag(B),symbcst(t1)),ag(A)))))
```

Note that functional symbols are denoted by words beginning with a lower-case letter, and variables are denoted by words beginning with a capital letter. Note also that the functional symbol `msg` represents the message received in the LHS and the message sent in the RHS for a given rule. So, in the previous example, the agent playing Alice's role begins the protocol and sends a message encoded with Bob's public key. We consider a message received or sent as a piece of knowledge of an agent, so this message is contained inside a state term. The intruder can create and analyse `msg` terms, but he cannot have access to the other information contained inside a `state` term, except when he is the main "actor" of the role. The separation between a state and a message received or sent is handled by the tree automaton.

Although our semantics differs a little from that of IF, a good use of tree automata allows us to be compatible with the standard of the project.

Even though the variables occurring in the example above could let one think that there are more than two agents,

Note that we follow the abstraction given in [22] and consider only two agents to verify secrecy. This simplification is exploited when building the initial automaton (2) representing the initial configuration of the network, and the initial knowledge of the intruder. Consequently, the approximation function (3) generated takes this abstraction into account too.

The architecture of the back-end is displayed in Figure 5. Starting from an IF-specification of a protocol in a file `protocol.if`, TA4SP can be invoked by typing on the command-line

```
TA4SP <filename> <step_number>
```

The parameter `<step_number>` allows one to compute either an over-approximation if its value is 0, or an under-approximation if its value is greater than 0. In the under-approximation case, the computation of the automaton may not stop, so `<step_number>` is considered as a limit where the computation will stop. In the over-approximation case, on the other hand, the computation of the automaton always stops. The computation is performed by a *completion* algorithm. These two methods and this algorithm are explained in Deliverable 5.2 [14].

Notice that the current version of TA4SP only deals with a fragment of IF. For example, keywords like `notin` or `contains` are not handled by TA4SP. If they occur the analysis is stopped returning the message "`This feature is not supported`". If the translation succeeds, i.e., if the specification uses the IF fragment handled by TA4SP, the algorithm for computing an approximation tree automaton (the completion algorithm) is run; see Deliverable 5.2 [14] for more details. As soon as the computation stops, the verification of all secrecy properties encoded in (4) can be done. Thus, if the intersection is empty, then these properties are verified. Otherwise one of the properties might be violated but the tool does not indicate which one. A future version of TA4SP will incorporate a clean way to separate the properties that have been verified from the properties that might have been violated, so that more precise information is obtained from a failed verification.

As explained in more detail in Deliverable 5.2 [14], the output of TA4SP currently contains:

- a description of the method used (over-approximation or not);
- the result obtained by TA4SP which is "true", "false" or "don't know". The answer "true" means that the protocol is secure (empty inter-

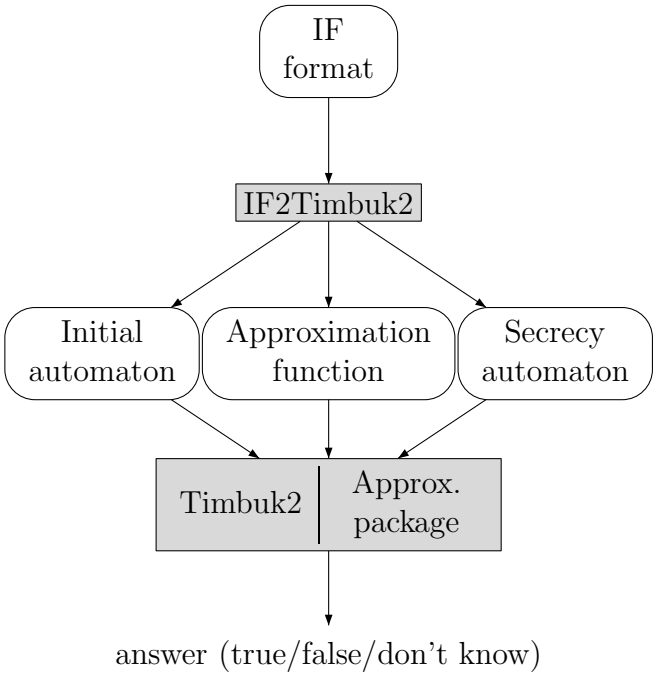

Figure 5: Architecture of the TA4SP back-end

section with attacker language), while the answer "false" means that there is a potential attack (however, this cannot be certain since we have used approximations and abstractions). The tools answers "don't know" when none of the above answers can be provided after the given maximal number of steps.

Presently, TA4SP does not provide an attack trace for unsafe protocols. We intend to develop a formal technique compatible with our approximation-based verification approach (as explained in Deliverable 5.2 [14]), and then to implement this functionality. Thus, we will be able to adapt our output format to the Output Format of the AVISPA Tool.

4 The Output Format

In order to facilitate the automatic parsing and gathering of the results yielded by each back-end, the output language has undergone a joint standardisation effort carried out by all project partners. The result of this process is the AVISPA Output Format (OF), intended to provide a clean and human-readable, yet uniform and well-defined, representation of the output of each back-end.

Currently, the back-ends of the AVISPA Tool output the following information:

- the result obtained by the back-end — that is, whether the input problem was solved (positively or negatively), or the problem was not tackled for some other reason;
- optionally, some statistics about the required resources (e.g., depth of the search, internal timings etc.);
- a description of the considered goal and, in case it was violated, the related attack trace.

The OF has not undergone any changes with respect to the AVISPA Tool v.1, so we point to Deliverable 4.4 [10] for a formal definition of the grammar of the OF and examples. Note, however, that we are currently working on a new version of the OF, which we will describe in the deliverable [15].

This common Output Format will also allow for different visualisations of the attack traces in a graphical user interface we are currently working on, where the users will also be able to specify/edit their own protocols in the HLPSL input language.

References

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

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

[3] A. Armando and L. Compagna. An optimized intruder model for sat-based model-checking of security protocols. In *Proceedings of the IJCAR04 Workshop ARSPA*, 2004. To appear in ENTCS, available at `http://www.avispa-project.org`.

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

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

[6] AVISPA. Deliverable 2.2: Algebraic Properties. Available at `http://www.avispa-project.org`, 2003.

[7] AVISPA. Deliverable 2.3: The Intermediate Format. Available at `http://www.avispa-project.org`, 2003.

[8] AVISPA. Deliverable 4.2: Partial-Order Reduction. Available at `http://www.avispa-project.org`, 2003.

[9] AVISPA. Deliverable 4.3: Heuristics. Available at `http://www.avispa-project.org`, 2003.

[10] AVISPA. Deliverable 4.4: AVISPA tool v.1. Available at `http://www.avispa-project.org`, 2003.

[11] AVISPA. Deliverable 5.1: Abstractions. Available at `http://www.avispa-project.org`, 2003.

[12] AVISPA. Deliverable 6.1: List of selected problems. Available at `http://www.avispa-project.org`, 2003.

[13] AVISPA. Deliverable 3.2: Assumptions on Environment. Available at `http://www.avispa-project.org`, 2004.

[14] AVISPA. Deliverable 5.2: Infinite-State Model-Checking. Available at `http://www.avispa-project.org`, 2004.

[15] AVISPA. Deliverable 7.3: Assessment of the AVISPA tool v.2. Available at `http://www.avispa-project.org`, 2004.

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

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

[18] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of TACAS'99*, LNCS 1579, pages 193–207. Springer-Verlag, 1999.

[19] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drieslma, J. Mantovani, S. Mödersheim, and L. Vigneron. A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols. In *Automated Software Engineering. Proceedings of the Workshop on Specification and Automated Processing of Security Requirements, SAPS'04*, pages 193–205. Austrian Computer Society, Austria, September 2004.

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

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

[22] H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. In *Proceedings of ESOP'2003*, LNCS 2618, pages 99–113. Springer-Verlag, 2003.

[23] T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In *Proceedings of CADE'00*, LNCS 1831, pages 271–290. Springer-Verlag, 2000.

[24] T. Genet and V. Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. In *Proceedings of LPAR'01*, LNCS 2250, 2001.

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

[27] 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.

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

[29] M. Turuani. *Sécurité des Protocoles Cryptographiques: Décidabilité et Complexité*. Phd, Université Henri Poincaré, Nancy, December 2003.

[30] H. Zhang. SATO: An Efficient Propositional Prover. In W. McCune, editor, *Proceedings of CADE 14*, LNAI 1249, pages 272–275. Springer-Verlag, 1997.