AVISPA

www.avispa-project.org

IST-2001-39252

Automated Validation of Internet Security Protocols and Applications

Deliverable 4.4: AVISPA tool v.1

Abstract

We describe version 1 of the AVISPA tool for security protocol analysis. More specifically, we describe here the architecture, the input and output languages, and the three back-ends of the tool.

Deliverable details

Deliverable version: *v1.0*
Date of delivery: *12.12.2003*
Classification: *public*
Person-months required: *10*
Due on: *30.11.2003*
Total pages: *17*

Project details

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

Information Society Technologies

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

Contents

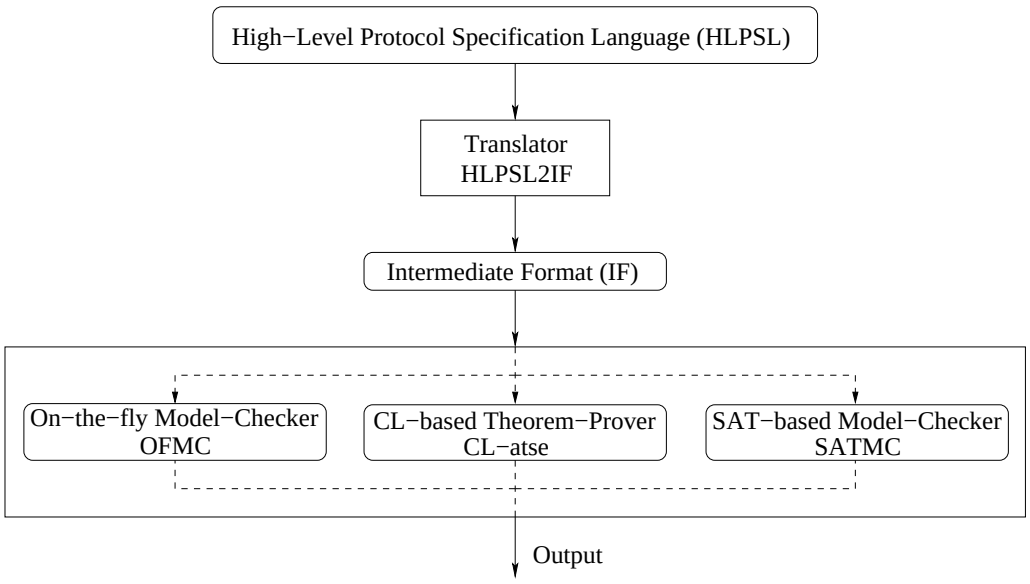

Figure 1: Architecture of the AVISPA tool

1 Introduction

The architecture 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 [6] specifications, which are then given as input to the different back-ends of the AVISPA tool: OFMC, CL-atse, and SATMC. (Note that, as shown in the figure and discussed below, some additional translation to tool-specific internal formats may be required.) 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 describe the HLPSL2IF translator. In Section 3 is devoted to a description of the back-ends. The Output Format is finally presented in Section 4.

It is worth pointing out that the AVISPA tool v.1 will be thoroughly assessed in a subsequent deliverable [11] by testing the tool against the AVISPA library of deliverable [10], a library of security problems drawn from the protocols developed by the IETF.

2 The HLPSL2IF Translator

The High Level Protocol Specification Language HLPSL is an expressive language for modelling communication and security protocols, which draws its semantic roots from Lamport's Temporal Logic of Actions (TLA, [19]). HLPSL is the language through which end users and protocol modellers make use of the AVISPA tool. As such, it is designed to be accessible: it is easy for human users to both read and write HLPSL specifications. To this end, HLPSL supports specifying protocols at a high level of abstraction and has built in many features that are common to most protocol specifications, such as intruder models and encryption primitives. In contrast, the Intermediate Format (IF) is a lower-level language at an accordingly lower abstraction level; that is, 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 respective deliverables [5, 6].

HLPSL specifications are translated into (semantically equivalent) IF specifications by the HLPSL2IF translator. The translator checks the syntax of the input file, carries out a series of semantic checks, and outputs (on the standard output channel) a corresponding IF specification. The compiler can be invoked by typing on the command-line:

```
hlpsl2if [options] [inputfile]
```

where the `options` is a combination of:

`--init`	for printing the initial state
`--intruder`	for printing the intruder rules
`--goals`	for printing the goals
`--prelude`	for printing the prelude file
`--rules`	for printing the protocol rules
`--types`	for printing identifiers and their types
`--all`	for printing everything
`--help`	for printing this list of options

3 The Back-Ends of the AVISPA Tool v.1

The back-end of the AVISPA tool take as input a security problem formally specified in the IF, automatically carry out an analysis of the problem, and—whenever they terminate—they output the results of their analysis. The

following back-ends are available in the AVISPA Tool v.1: OFMC, an on-the-fly model-checker based on lazy data type (developed and maintained by ETHZ), CL-atse, a protocol analyser based on Constraint Logic (developed and maintained by INRIA, Nancy), and SATMC, a SAT-based model-checker (developed and maintained by UNIGE).

3.1 OFMC

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 protocol problem) for flaws, while OFMC-FP tries to verify the protocol by computing the fixed-point of an abstracted model (see the deliverable [9] for more details). Both modules share common basic algorithms and data structures, such as unification and substitution.

The IF2OFMC Translator reads IF files as input and returns the initial state(s), 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 (see the deliverable [8] for more details on this heuristics).

The module OFMC-core is a considerable extension of the first version of the prototype on-the-fly protocol model-checker we developed as part of the AVISS project [1, 12]. As discussed in more detail in [13, 14], 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. The higher layer uses the lower layer to build the symbolic (and "step-compressed", cf. [7, 13, 14]) 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).[1]

The successor-relation computed by OFMC-core is used to build a tree from the extended initial state provided by the session compilation and checking for 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

[1] In other words, we distinguish two layers of search: the lower layer is a search in the space of constraints to find all (symbolic) solutions for a given set of constraints according to the Dolev-Yao model; the higher layer builds on the lower one to search in the symbolic search space to determine if an attack state is reachable.

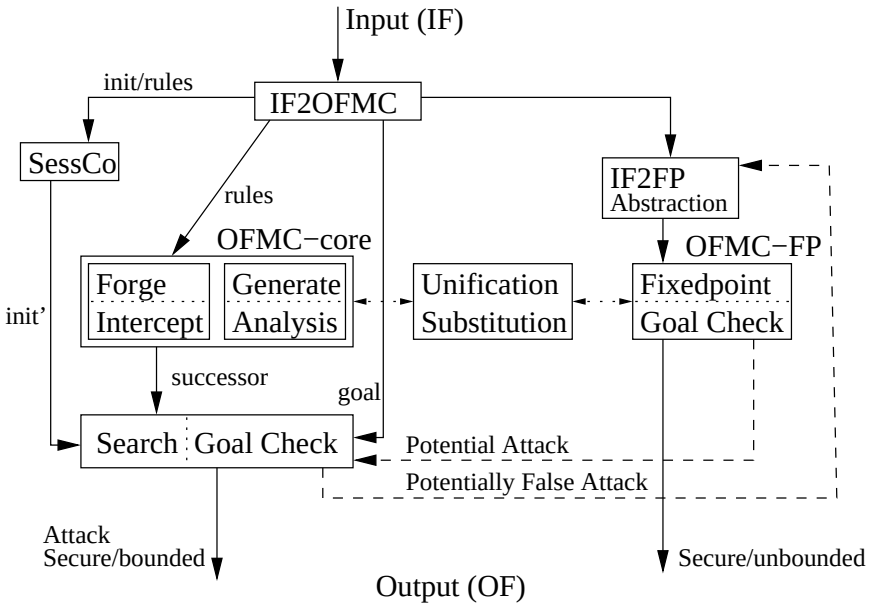

Figure 2: Architecture of OFMC

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 also that for a bounded number of sessions OFMC-core is also complete). Also, if the number of sessions is bounded, 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 the deliverable [9]. (Note that session compilation is redundant in this case.) On these abstracted rules, OFMC-FP computes a fixed-point of reachable facts.

The goal state is different from the one of OFMC-core since it is based on a slightly different model. If the goal check is negative 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 is no corresponding attack for 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. (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> [-r <number>] [<number>] [-p <number>*]
```

where all options are for debugging purposes only:

- `<filename>` is an IF file to be checked.
- option `-r` lets the user specify a bound on the number of session repetitions (see [13, 14] for more details), where the default is 1, which allows for fast detection of replay attacks.
- The optional number (without flag) specifies a depth bound for the search. In this case, the compiler 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 root node
  - `-p 0 1` is the second successor (next to left-most) successor of the node obtained by `-p 0`,

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

We have implemented a prototype version of OFMC-FP (which takes as argument simply the IF file to be analysed) and will report on it in more detail in a future deliverable.

3.2 CL-atse

The CL-atse (Constraint Logic - ATtack SEarcher) is a direct implementation in OCAML of the CL approach. CL-atse reads the Rules and Inits sections from the IF specification (see [6] for more details about the IF) and assumes that the other sections match the rules implemented in CL-atse. This clearly requires a few changes to the actual implementation if modifications or extensions to the intruder rules are made, but improves performance considerably.

The protocol specification read from any IF file is first translated into a new protocol specification adapted to CL-atse. This transformation has

two main characteristics: each role instantiated by the protocol is extracted as a list of received/sent messages plus a list of witness/request terms as introduced in the IF language, and a list of conditions, and all local variable names are extended to global variable names. The states are expressed by unification constraints between global variables at each protocol step. Let us note that for most protocols examples these unification problems are trivial and the approach is relatively fast in the search for attacks.

In summary, the protocol specification internally used by CL-atse is a list of protocol steps in the following syntax:

$$\texttt{iknows}(R) \Rightarrow \texttt{iknows}(S).\texttt{witness}(...).\texttt{request}(...) \quad [u = v]\ \mathit{Conditions}(...)$$

where R is a message received by the principal, S is a message sent to the intruder, u and v are two terms to be unified (i.e. we consider only substitutions such that the instantiations of u and v are equal), `witness`(...) and `request`(...) are two lists of witness and request terms, and $\mathit{Conditions}(...)$ is a list of conditions on the variables (like $Na \neq Nb$, for example).

This translated protocol specification is then simplified in order to accelerate the subsequent search for attacks. First, we remove from any list of received or sent messages (R or S in the step above) all the messages necessarily known by the intruder when the step is executed. Typically, if the intruder knows a, b, ka, then :

$$\begin{array}{rrcl} \text{The step} & \texttt{iknows}(a).\texttt{iknows}(M1) & \Rightarrow & \texttt{iknows}(b).\texttt{iknows}(M2) \\ \text{becomes} & \texttt{iknows}(M1) & \Rightarrow & \texttt{iknows}(M2) \end{array}$$

Then, any useless protocol step is either merged with the following step or completely removed if none is present. For example, the step

$$\texttt{iknows}(R) \Rightarrow \texttt{witness}(...) \quad [u = v]$$

is said to be useless since it neither extends the intruder knowledge nor produces any authentication attack. In the same way, any free protocol step is either merged with the previous protocol step or pre-executed on the initial intruder knowledge. For example, the step

$$\epsilon \Rightarrow \texttt{iknows}(S).\texttt{request}(...)$$

is said to be free since the intruder can execute this step as soon as possible without losing any attack (where ϵ is the empty list). These simplifications of the protocol specification are simple, but they can save a lot of time in the search for attacks, since most of the time is used for enumerating all the interleavings of the protocol steps.

Once translated and simplified, the protocol specification is checked for attacks using constraints logic methods (CL [16, 21]). That is, the system state (describing the principal and intruder states) 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 elements to test if they are satisfied by at least one instance of the variables. For each execution of a protocol step, this symbolic system is modified to represent the new states of the principals and the intruder and the new conditions, and tested for satisfiability of at least one of the security goals. By enumerating all the interleavings of protocol steps for a given depth, CL-atse determines if we can reach a system state violating one of the security properties, or if the protocol is secure with respect to this depth. The protocol depth represents the maximum number of sequential steps CL-atse will analyse. For protocols with loops, this bounds the number of iterations CL-atse will analyse, but for protocols without loops, this is fixed to the total number of protocol steps and the search is complete.

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

```
cl-atse [filename] [n] [-if v] [-out] [-ns] [-v] [-help]
```

where :

- `filename` is the name of the IF file to be read. If not provided, the standard input channel is used instead.
- `-if v` gives the version number of the IF file to analyse (default is 2).
- `n` in the number of sequential sessions to analyse for if version 1, or the number of loops to analyse for IF version 2.
- `-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 OF is not ensured in this case.)
- `-help` asks CL-atse to print help information listing all the available options.

3.3 SATMC

SATMC performs an automatic translation of security protocols into propositional logic (SAT) with the goal of 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 [15]) is that given a security problem Π expressed in IF and an integer k, it is possible to generate a propositional formula Φ^k_Π such that any model of Φ^k_Π corresponds to one or more attacks on Π. The encoding of Π into a SAT formula can be done in a variety of ways. The basic idea is to add 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 actions 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_Π, i.e. a symbolic representation of the search space up to depth k, is built on top of these propositional variables and looks like:

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

where I_0, T_i^{i+1}, and F_n 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. A lot of effort has been put in devising the encoding technique that leads to propositional formulae of manageable size. Two encoding techniques are currently implemented in SATMC: the first belongs to the family of so-called *linear encodings*, the second is the more sophisticated *graphplan-based encoding* (see [18] for a survey). Experimental results [4] clearly indicate that the latter is superior to the former on the domain of security protocol analysis.

The architecture of SATMC is shown in Figure 3.3. SATMC takes as input an IF specification representing a security problem, 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`;
- `mutex`: a boolean parameter for enabling or disabling the abstrac-

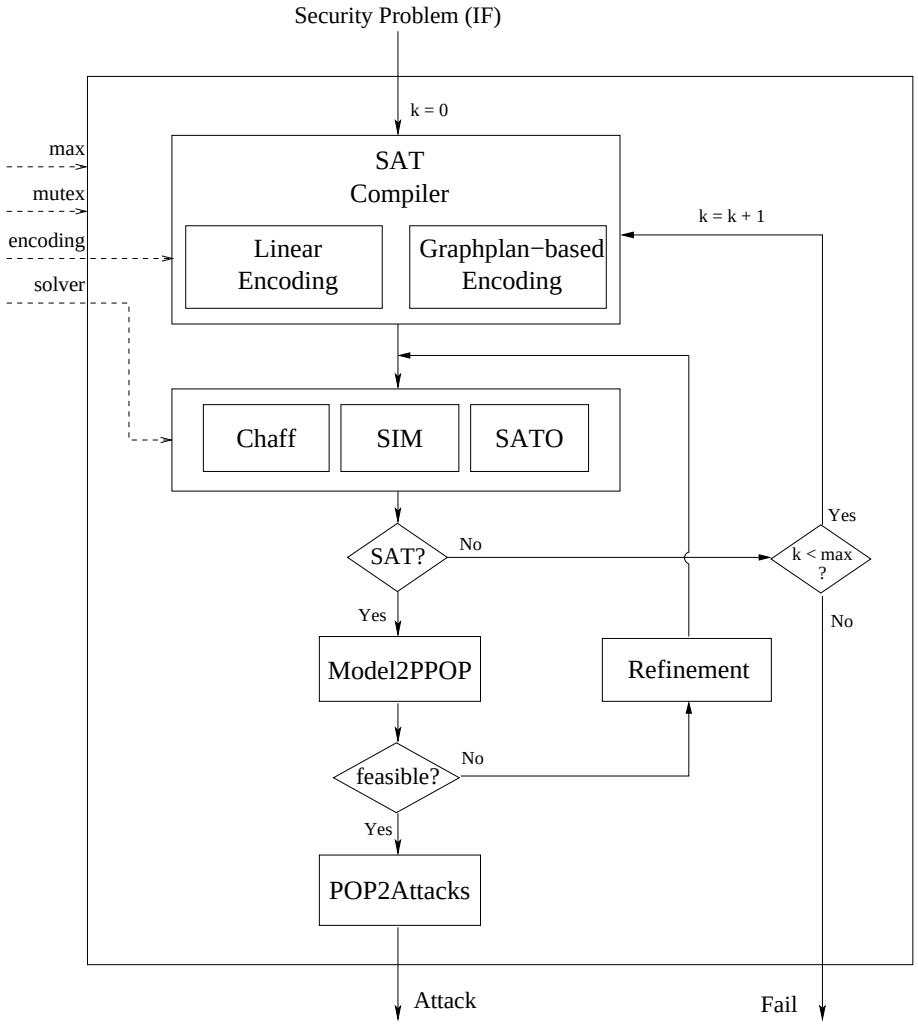

Figure 3: Architecture of SATMC

tion/refinement strategy the back-end provides (see [3] for more details); by default it is set to `false`;

- `encoding`: the selected SAT reduction encoding technique (currently the *linear* [2] and the *graphplan-based* [4] encodings are implemented); it can be set to either `linear` or `graphplan` (default value); and
- `solver`: the selected state-of-the-art SAT solver (Chaff [20], SIM[17], and SATO [22] are currently supported); it ranges over the values `chaff` (default value), `sim`, and `sato`,

and returns either an attack or `fail` stating that no attack has been found on the input security problem.

In more detail, SATMC applies the selected encoding technique to the 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 plan (PPOP). Similarly to the concept of a partial-order plan in AI planning, a *partial-order plan* is a partially ordered (multi)-set of IF rules leading from the initial state to a goal state.[2] If the abstraction/refinement strategy is disabled, we are guaranteed that a "pseudo" partial-order plan corresponds to a partial order plan. Otherwise, as explained in [3], the "pseudo" partial-order plan 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 no spurious counterexample is met or the formula becomes unsatisfiable. As soon as a partial-order plan is found, it is translated into attacks which are reported to the user accordingly to the syntax specified in Section 4.

SATMC can be invoked by typing on the command-line

```
satmc <filename> [--max=<number>] [--mutex=<bool>]
                 [--encoding=<encoding>] [--solver=<solver>]
```

where `<filename>` is the IF problem to be analysed and each option is as described above.

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 underwent 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

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

human-readable, yet uniform and well-defined, representation of the output of each back-end.

Each back-end of the AVISPA tool outputs 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.

An example of the output produced by a back-end is given in Figure 4.

We now formally describe the grammar of the OF. We use a BNF representation, plus the symbols `*`, denoting zero or more occurrences of the preceding symbol, and `+`, denoting *one* or more occurrences; moreover, the following syntactic categories are employed:

- `ident`: a string;
- `int`: an integer;
- `float`: a floating-point number;
- `msg`: a message in "Alice and Bob" notation, intuitively describing what one agent is sending to another;
- `goalDescription`: a string describing a goal;
- `EOLTerminatedString`: an end-of-line-terminated string.

Both `msg` and `goalDescription` reflect the input specification, i.e. they contain the same symbols employed in the input and follow the HLPSL grammar as described in [5].

The top-level production rules of the OF grammar are shown in Figure 4. An `OFFile` denotes an output file as obtained by invoking one of the back-ends of the AVISPA tool on a set of problems. It consists of an arbitrary number of `outputDescription`'s, each one describing the output of a back-end on a particular problem; comments can be introduced anywhere in the file by means of an expression of type `comment`.

Figure 4 describes the production rules for `outputDescription`:

```
SUMMARY
  YES 0.5 1250

STATISTICS
  searchTime 0.5 seconds
  depth 1250 nodes

PROTOCOL
  NSPK.if

BACKEND
  OFMC

VIOLATED GOAL
  a authenticates b on na

ATTACK TRACE
  1.1. a -> i: {na,a}ki
  2.1. i(a) -> b: {na,a}kb
  2.2. b -> i(a): {na,nb}ka
  1.2. i -> a: {na,nb}ka
  1.3. a -> i: {nb}ki
  2.3. i(a) -> b: {nb}kb
```

Figure 4: Output Format: an example

```
OFFile ::= outputDescription | OFFile

outputDescription ::= summary
                      statistics*
                      protocolId
                      backEndId
                      detailedResult

comment ::= "%" EOLTerminatedString
```

Figure 5: BNF specification of the OF: top-level production rules.

1. `summary` summarises the result of the analysis, i.e. whether a goal was violated (`YES`) or not (`NO`), or some system resource was exhausted (time-out (`TO`) or memory-out (`MO`)), or that the problem was not supported (`NS`) or not attempted (`NA`), i.e. the problem could not be analysed by the selected back-end. The series `shortStat` of floating-point numbers summarises how much of the system resources was used;

2. `statistics` optionally explains in detail what each `shortStat` denotes: for each number, a label and its measurement unit (i.e., seconds, megabytes, etc.) are provided;

3. `protocolId` is a string denoting the name of the protocol under examination;

4. `backEndId` is the name of the back-end that tackled the problem;

5. `detailedResult` shows in detail if the goal was found to be safe or violated, and in the latter case, the attack found.

Finally, the BNF specification of attack traces is given in Figure 4. An attack trace is defined in terms of which messages are exchanged among the agents. Each message (`msg`) consists of an optional session and step number, a user name (the sender), another user name (the receiver) and the message itself. A `user` is either an identifier or the name of the intruder, or the intruder impersonating a user.

```
summary ::= "SUMMARY" result shortStat*
result ::= "YES" | "NO" | "TO" | "MO" | "NS" | "NA"
shortStat ::= float

statistics ::= "STATISTICS" labeledStat+
labeledStat ::= statLabel shortStat unitLabel
statLabel ::= ident
unitLabel ::= ident

protocolId ::= "PROTOCOL" ident

backEndId ::= "BACKEND" ident

detailedResult ::= safeGoalResult | violatedGoalResult
safeGoalResult ::= "SAFE GOAL" goalDescription
violatedGoalResult ::=
  "VIOLATED GOAL" goalDescription attackTrace
attackTrace ::= "ATTACK TRACE" trace+
```

Figure 6: BNF specification of the OF: production rules for `outputDescription`.

References

[1] A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS Security Protocol Analysis Tool. In *Proceedings of CAV'02*, LNCS 2404, pages 349–354. Springer-Verlag, 2002.

[2] 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.

[3] A. Armando and L. Compagna. Abstraction-driven SAT-based Analysis of Security Protocols. To appear in the Proceedings of SAT 2003, LNAI, Springer-Verlag, 2003.

[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.

```
trace ::= (int ".")* (int ".")*
          user "->" user ":" msg

user ::= ident | "i" | "i" "(" ident ")"
```

Figure 7: BNF specification of the OF: production rules for attack traces.

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

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

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

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

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

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

[11] AVISPA. Deliverable 7.2: Assessment of the AVISPA tool v.1. Available at `http://www.avipa-project.org`, 2003.

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

[13] 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. Extended version available as Technical Report 404, ETH Zurich, Dep. of Computer Science, `www.inf.ethz.ch/research/publications/`.

[14] 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. Extended version available as Technical Report 405, ETH Zurich, Dep. of Computer Science, `www.inf.ethz.ch/research/publications/`.

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

[16] 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.

[17] 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.

[18] 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. Morgan Kaufmann, 1996.

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

[20] 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.

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

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