AVISS — Automated Verification of Infinite State Systems

(IST-2000-26410)

Deliverable D3.4

Final definition, implementation and experimentation with the on-the-fly model checker

1 Introduction

The AVISS prototype verification tool for security protocols has the architecture shown in Figure 1. Protocols are specified in the high-level protocol specification language HLPSL, and the HLPSL2IF translator developed in WP2 translates these specifications into the low-level but tool-independent Intermediate Format, IF. The design and implementation of the three approaches that work on the IF is the task of the third work-package WP3, which consists of three steps:

- *Encoding.* Specifications in the Intermediate Format are translated into tool-specific encodings that fall into the scope of application of our three model-checkers.
- *Experiments.* The model-checkers are applied to the verification problems of the corpus [5] of security protocols.
- *Tuning.* The experiments indicate ways to improve the encodings as well as the inference strategies implemented in the available automated deduction engines.

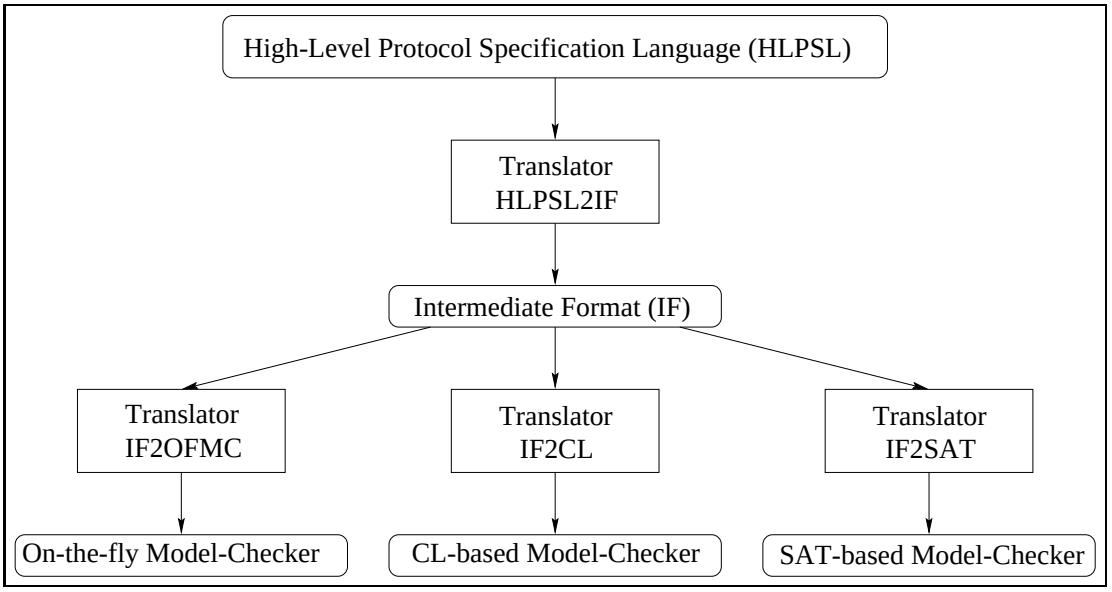

Figure 1: Architecture of the prototype verification tool

This deliverable D3.4 describes our implementation of task T3.1, i.e. it defines the encoding from the IF to the input format for the on-the-fly model checker based on lazy data-types, develops a prototype translator implementing the encoding, and experiments with problems from the corpus, tuning the encoding and/or the on-the-fly model checker. We proceed as follows: after a brief overview of the on-the-fly model checker in Section 2, in Section 3 we describe in detail the translation of the IF to Haskell, in Section 4 we discuss the lazy intruder model and in Section 5 we discuss advanced features of HLPSL and IF. In Section 6 we report on our experiments with the on-the-fly model-checker, and in Section 7 we draw conclusions and discuss future work.

Note that a detailed comparison of the relative performance of the three tools (see [2, 4, 10, 11]) of the AVISS project will be subject of future deliverables. A preliminary comparison on a selection of examples is given in the paper [1], which we have written as part of deliverable D4.2. See also the project webpage

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

2 Overview

The HLPSL2IF-compiler of WP2 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 unroll the transition system provided by the IF into a tree of infinite depth, where the root node corresponds to the initial state and the successor relation corresponds to the transition relation. 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 (in terms of a trace of events that led to the flaw state). 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 heuristics as tree-transformers and filters on an infinite tree. 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 and the static part of the model 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 architecture displayed in Figure 1 to that of Figure 2.

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. Despite this variety, this model was only a restriction of the Dolev-Yao intruder model [8], which involves infinitely many possible intruder messages. As an elegant solution to both reduce the branching and realize the complete Dolev-Yao model, we now use a symbolic representation, which was proposed by the Nancy group and already used in the Constraint-Logic approach [3]. This symbolic representation avoids enumerating the possible messages the intruder can generate, but stores only from what knowledge a certain message was constructed. The representation is evaluated in a demand-driven fashion, hence the name 'lazy intruder'.

The rest of the deliverable is organized as follows: We start by showing how the IF format can be mapped into Haskell data types. Based on this we describe in detail how the lazy intruder — which is now an essential part of our model — is implemented in Haskell. After a short coverage of how to handle some advanced concepts provided by the new version of the IF, like authentication goals, we summarize our experimental results with the new tool. Finally, in the conclusion section we give also an outlook to possible future extensions and improvements of the approach.

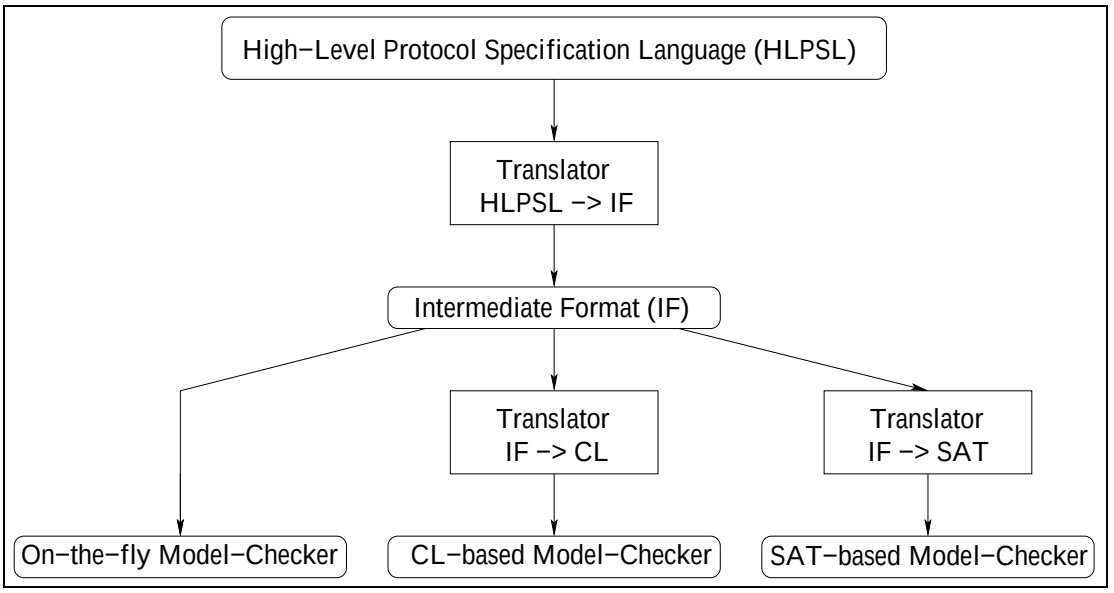

Figure 2: Architecture of the prototype verification tool

3 Translation of the IF to Haskell

In the following we assume familiarity with the definition of the IF of deliverable D2.2. 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 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 as an abstract syntax tree. However there are some subtleties which need special consideration; they are discussed in this section.

3.1 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, enjoying special properties which we will list later. 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 trigger the properties of such items; for example, if an agent knows a public-key encrypted message `Crypt(pk(ka),xM)` and the corresponding private key `pk(ka)'`, then he can derive `xM`. However rules that describe the behavior of agents are *blind* for this type information, e.g. `scrypt(xKab,xA)` just requires some item `xA` (that is not necessarily the name of an agent and) that is encrypted with an item `xKab` (which is not necessarily a symmetric key). (Recall that variable names start with the letter 'x'.)

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(xKab),mr(xA))`, ensuring that `xKab` is a symmetric key and `xA` is an agent name. 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 `message` 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 message = Atomic String
             | Pk Message | Pk' Message | Mr Message | ...
             | Crypt Message Message | C Message Message | ...
```

Note that some terms are possible in the Haskell data type that may not make much sense, e.g. `pk(sk(ka))`; the on-the-fly model-checker OFMC trusts the HLPSL2IF compiler to produce only well-formed terms and rules.

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

`crypt(sk(ka),c(Na,Nb))`

becomes

```
Crypt (Sk (Atomic "ka") (C (Atomic "Na") (Atomic "Nb"))) .
```

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 term, then `xKa` must be a public key and `xKa'` the corresponding private key. We replace such terms by `Pk xKa` and `Pk' xKa` in Haskell.[1]

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))`. In the untyped model, using variables that are introduced by the lazy intruder model described below, the comparison procedure becomes complicated and inefficient. Currently, we use an approximation by associating every term to the right when translating from IF to Haskell data types. During the search, two terms are then compared by simply comparing their sub-terms. This does not solve the problem completely (e.g. `c(Ka,Na)` cannot be matched against `c(A,c(B,Na))` in this setting), and we are currently working on this.

3.2 States

A state consists of a list of facts. A fact is one of the following:

- a principal term (describing an honest agent, awaiting a message of a certain format and content, and his knowledge),
- a message term (i.e. one message was sent but has not been received yet),
- intruder knowledge (terms the intruder has seen),
- a secret (i.e. a term that denotes the secret item and the session the secret is associated with),
- a witness term (i.e. the internal information that a certain message was sent by an agent to another agent for a certain purpose),

[1] During the experimentation phase, HLPSL and IF were extended to handle tables of public keys. In this setting the described replacement is not valid. In the considered cases this was never a problem (since private-key encrypted messages were never transmitted in these protocols), however we are working on a general solution to be included in future versions of our tool.

- a request term (i.e. the fact that a corresponding witness term is requested; if it is not present, this is an authentication flaw), or
- a give term (i.e. a run of the protocol is over and the intruder gets the corresponding short-term secret).

All states are translated literally into Haskell except for the following kinds of goal states:

- *Secrecy-goals.* They 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 plays one role. For each secrecy goal we keep only the number *session*, giving a list of secret sessions. (During the search, the on-the-fly model-checker OFMC tests, for every secret term, if both the session is secret and the term is known by the intruder.)
- *Authenticate-goals.* These are a static (protocol-independent) part of our model; hence they are ignored by the parser. (During the search, the OFMC performs the following check: if a request term is present, then we must see if the current state contains a corresponding witness term. If yes, they are both removed from the state, otherwise the current state manifests an authenticate flaw.)

3.3 Rules

In the IF, a rule has the form $LHS \Rightarrow RHS$ where LHS and RHS are fact lists. The meaning of such a rule is: if the current state contains all the facts that are listed in the LHS, then the transition to a successor state is possible where the LHS items are replaced by the RHS items.

We translate the IF rules into appropriate Haskell data types where we ignore two kinds of rules. First, we ignore all static (protocol-independent) rules like `i(xK).i(scrypt(xK,xM))=>i(xM)`; they are a hard-wired part of our on-the-fly model-checker OFMC and don't need to be read. Second, we also ignore the impersonate rules, which describe the behavior of the intruder in a protocol-dependent way; instead we apply the more efficient and protocol-independent lazy intruder model described in the following section.

4 The Lazy Intruder

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, known as the 'lazy intruder', which has several advantages over the previous approach. It was first developed and described by the Nancy group [3].

We start by giving the motivation for the change of the intruder model, then illustrate the idea of the lazy intruder model at hand of some examples, and finally cover its integration into our approach.

4.1 Drawbacks of the Impersonate Rules

The intruder behavior is not protocol-dependent: the intruder can send whatever message he can construct from his knowledge (as usual we assume perfect cryptography, i.e. he cannot decrypt or encrypt messages without knowing the necessary keys). The impersonate rules of the IF specialize this general behavior model to a given protocol (and possibly session instances) under the assumption that the intruder will only generate messages that other agents will accept as legal messages of the protocol.

[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.

One problem of the impersonate rules in the untyped model is that any part of a message term where the receiver expects no specific value could itself be an arbitrarily complex message term the intruder is able to construct. One must consider these possibilities if one doesn't want to exclude any attacks, as the following example shows. A type-flaw attack upon the protocol

1. $A \to B : N_A$
2. $B \to C : \{N_A\}_K$
3. $C \to A : \{N_A, N_A\}_K$

involves sending the composed term N_A, N_A where B expects a nonce (in the following, $\times$ means that the message is intercepted by the intruder; I_A means that the message was sent by the intruder, but to the receiver it looks as if it came from A):

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. $I_C \to A : \{N_A, N_A\}_K$

In general, it does not seem possible to give a bound for the complexity of the messages to be considered. Hence, to realize the full Dolev-Yao model one has to take infinitely many possibilities into account. The impersonate rules of the IF give just a finite approximation of this intruder model: if a message part is supposed to be atomic according to the protocol (for instance in the above example the nonce N_A which B expects in the first step), then the impersonate rules neglect the possibility that the intruder might send a composition of known items instead (like N_A, N_A in the above example).

Another major drawback of using the impersonate rules (especially in the on-the-fly model-checking approach) is the enormous branching of the search tree caused by these rules. For instance, assume the intruder wants to generate the first message of the Otway-Rees protocol

$$1.\ A \to B : M, A, B, \{N_A, M, A, B\}_{K_{AS}}$$

and assume also that he has 10 items in his knowledge, which he can independently use to fill any slot in the encrypted part (which B cannot analyze) and for the unencrypted item M. Then there are 10^5 possible messages (and, as pointed out above, this is only an approximation).

4.2 The Idea: Symbolic Representation

The basic idea to handle in an elegant way both the completeness and the complexity problems is based on the following observation. When the intruder sends a message to some receiver, and when this receiver does not expect a particular value for some part of that message, then the intruder can freely choose how to build that part. In other words, for this step it is irrelevant which concrete value the intruder inserts in the message part.

As an example, consider again the first message of the Otway-Rees protocol: the receiver only expects a particular value in the position of the unencrypted agent names; the value for M and the encrypted part is completely irrelevant for the receiver, i.e. the intruder may send any message of the form

$$1.\ A \to B : x_M, A, B, x_{AS}$$

where x_M and x_{AS} are arbitrary message terms 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:

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

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

4.3 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 $\{a, x_{Na}\}_{kb}$ where x_{Na} is any message; then we have the substitution $x_M \mapsto \{a, x_{Na}\}_{kb}$ which must also be applied to the *from*-constraints:

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

It is at this point necessary to check if the intruder was able to generate such a message from the denoted IK. How this check is performed in general is described in [3]; we only cover here the relevant cases for the 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

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

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

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

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.

4.4 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 onto that item. This is an elegant 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 $\{M\}_{Na,xNb}$ where Na is a nonce generated by an honest agent, xNb is an item generated earlier by the intruder, and the pair Na, xNb acts as a symmetric key. The analysis procedure performs the generability check for the encryption key: Na, xNb *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.

The implementation of the analysis procedure is currently limited to three basic cryptographic operators: public-key encryption, symmetric encryption, and hash-functions. One might want to analyze protocols that make use of cryptographic operations with special properties, like RSA encryption with the properties

$$\{\{M\}_K\}_{K'} = M \text{ and } \{\{M\}_{K_1}\}_{K_2} = \{\{M\}_{K_2}\}_{K_1} .$$

To model such properties, it is necessary to extend the analysis procedure. Currently we have only added one analysis case needed in the analysis of the Shamir-Rivest-Adelman protocol:

1. $A \to B : \{M\}_{K_A}$
2. $B \to A : \{\{M\}_{K_A}\}_{K_B}$
3. $A \to B : \{\{\{M\}_{K_A}\}_{K_B}\}_{K'_A} = \{M\}_{K_B}$

Suppose, A has sent the first message. Since A cannot analyze the answer from B, the intruder impersonating B can send any message x_M and A answers with the message $\{x_M\}_{K'_A}$. If this message $\{x_M\}_{K'_A}$ were fed into the analysis as described above, it would only reveal that the intruder knowing K_A can decrypt this message and hence find out x_M, but this is not interesting since he himself generated x_M. Using the above properties of the RSA encryption, however, there is an alternative based on the match of x_M with the message A has sent in the first step: $x_M = \{M\}_{K_A}$. In this case we have $\{x_M\}_{K'_A} = \{\{M\}_{K_A}\}_{K'_A} = M$; hence, the intruder has learned the secret message M.

To handle this and similar examples, we have extended the analysis procedure as follows: in the case that a variable x_M, i.e. an intruder generated message, is encrypted with a public key K, we consider the match of x_M against $\{x_L\}_{K'}$ (where x_L is a fresh variable). If the generability check for x_M is possible then x_L is learned by the intruder. Note however this is only interesting if x_L is not itself intruder-generated, like in the above example where x_L is substituted into M during the generability check.

This extension of the analysis procedure handles many cases, but it is not complete since, for instance, commutativity of encryptions is still neglected. Implementing a complete, more systematic solution will be subject of future work.

4.5 Triple-steps

We have seen that the lazy intruder model comprises three atomic operations: the intruder generating a message from his knowledge, an honest agent receiving such an intruder generated message, and the intruder analyzing an honest agent's message he has seen. We now integrate these three steps into a *triple-step*, consisting of an impersonate step, a "step" step, and a divert step.

1. *Impersonate.* From the current state the intruder selects a w-term, describing an honest agent willing to communicate. If the corresponding step rule contains a message term in the left hand side, the intruder provides such a message by sending some message from his knowledge.

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

Note that in such a triple-step the first item is not contained if the honest agent denoted by the w-term does not expect a message but wants to initiate the protocol, and the third item is not contained if the honest agent awaits the last message of the protocol and doesn't send a reply message.

Triple-steps are sufficient to model all communication in the network: we can see the intruder as a 'moderator' of all communications between agents, i.e. a and b never talk directly with each other, but the intruder diverts the message from a, and after analysis, possibly sends it to b. We have thus chosen to consider only communication in form of these triple-steps, which we thus regard as new atomic steps of our model from now on.

5 Implementing Advanced Concepts of HLPSL and IF

We now describe how we can handle some advanced features of the IF which were added to the specification of deliverables D2.1 and D2.2.

5.1 Short-Term Secrets

Short-Term secrets can be session keys or nonces that 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' [13]. In our model, short-term secrets are at first a goal: if the short-term secret can be learned by the intruder before the session is over, then this manifests a violation of the short-term secrecy goal. After the session is over (i.e. the last message of the protocol was issued by the honest agent), this goal is removed and the item 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 — a normal secrecy goal. The rule that describes the last protocol step contains a give-term on the right-hand side; this give-term releases the secrecy term 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 terms as usual, i.e. they are normal facts and for every state we check if a secret term (in a secret session) is in the intruder knowledge. When a give term appears on the RHS of a rule, we search for an appropriate secret term in the current state and remove it. Then the item is added to the intruder knowledge.

5.2 Authenticate Goals

As explained above, the IF issues witness and request terms in the RHS of step rules. A witness term expresses the fact that an agent created a fresh item, like a nonce or a session key, for a specific purpose. A request term requires the existence of the corresponding witness term in the current state. Hence witness terms 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 term appears on the RHS of a rule, however, then we check if a matching witness term is in the present state. If yes, the request is satisfied and the witness term 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.

5.3 Parallel Sessions

The concept of parallel sessions allows us to model agents that don't have an internal state, like a key-server that doesn't necessarily 'know' about the rest of the protocol. The behavior of such an agent is described as a transformation on the intruder knowledge in the form `i(`t_1`) => i(`t_2`)`. The Nancy group can use parallel sessions to handle an agent's behavior as part of the intruder-knowledge analysis; we currently can't do this since our intruder model is 'hard-wired'. However, we can easily cope with such rules by considering them as normal step rules:

$$\texttt{m(0,para,para,para,}t_1\texttt{,para) => m(0,para,para,para,}t_2\texttt{,para)}\,,$$

Hence, we add dummy information for every irrelevant item of the message-term.

6 Experiments with the On-The-Fly Model-Checker

We have experimented our on-the-fly model-checker OFMC against the 51 protocols of the Clark/Jacob library [5] (counting different versions of the protocols as different protocols according to [9]). 33 of these protocols are already known to be flawed, as reported in, for example, [5, 9]. Note that we don't count the Yahalom protocol to be flawed for reasons explained in [11]. More importantly, note that our tool finds a previously unknown flaw the Denning-Sacco Protocol; this flaw was first discovered and described by the Nancy group [11].

Hence, the corpus contains 34 protocols that are flawed. The remaining 17 protocols are to our knowledge flawless: in all settings of session instances we have considered so far, the OFMC tool has never revealed a flaw in a reasonable amount of time (1 hour). In fact, the OFMC tool is only a semi-decision procedure, i.e. if a protocol is flawless the search for a flaw never terminates. Hence, we consider in the rest of this section only the 34 protocols that are known to be flawed.

We were able to successfully formalize all known to be flawed protocols and their security goals in HLPSL and automatically translated them into IF with 3 exceptions: the Kerberos protocol and the Wide Mouthed Frog protocol could not be modeled, because they rely on properties of time stamps that cannot be specified in HLPSL yet; for the CCITT X.509 protocol it was not possible to specify an appropriate security goal. These cases are reported in detail in [11], which contains also a general discussion on HLPSL an IF as employed by all project partners.

For the remaining 31 protocols that can be modeled in HLPSL, the OFMC tool is very successfull: for each protocol the flaw is found in less than 20 seconds, and the whole set of protocols has a total analysis time of less than 1 minute on a Sun Workstation with a 500MHz SPARC processor (the times are averaged over 5 runs.) The results of our tools are summarized in Figure 3.

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 triple-step method described in Section 4.5 has lead to a major improvement here: since we regard a sequence of applications of an impersonate rule, a step 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 triple-steps; 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, the triple-steps so to speak "reduce the depth" by merging layers of the tree. Hence the lazy intruder might be considered as a horizontal compression and the triple-steps as a vertical compression of the search tree.

As we remarked above, a detailed comparison of the relative performance of the three tools of the AVISS project will be subject of future deliverables. A preliminary comparison on a selection of examples is given in [1], and shows that the OFMC tool outperforms the other two approaches [2, 4, 10, 11].

7 Conclusions and Outlook

We have implemented a tool that alone more than satisfies the objectives set out in the project proposal for the combined tool. One success criterion was to find 70% of the flaws in less than one hour. Our tool finds 88% of the flaws in less than one minute.

The main reason for this great success is, we believe, the cross-fertilization of the different approaches that eventually lead to a combination of different techniques and ideas, namely of the lazy intruder approach as it was proposed and developed in Nancy, which allows an elegant and efficient model of the intruder behavior, and the merging of atomic events into triple-steps, which rules out many interleavings while preserving all attacks. The first technique stems from the constraint-logic approach (but can be successfully applied also in our on-the-fly model-checker OFMC). The second technique can be seen as a simple form of partial order reduction, which stems from the world of automated search. This illustrates how different approaches can meet and benefit from each other's strengths.

There are still many interesting questions to consider, both for theoretical research and for practical applications. For example:

- *Search Heuristics.* We haven't really exploited search heuristics, yet. The search can be focussed and guided by evaluation functions like 'has the intruder learned anything new through this step, how interesting is what he learned?'.
- *Partial Order Reduction.* The current use of partial order reduction — the triple steps can be seen as a simple form of this concept — is only light-weight. We plan to develop a formal concept for 'equivalent' states and reduce the state space that way.
- *Cryptographic Operators.* We have currently a hard-coded model of the cryptographic operators used. We would like to have a more flexible mechanism where the user can add further operators with desired properties. This can also be used to better exploit parallel sessions (which we currently only handle as normal protocol steps).
- *Multi-cast protocols.* We also wish to broaden the applicability of our methods to more advanced protocols such as group protocols or multi-cast protocols.
- *Formal Meta-Proofs.* We also plan to employ formal methods to formalize our verification model, and to compare it with different models as used in systems like CAPSL/CIL [7], Casper [12], Taps [6], and other methods based on inductive verification such as [13].

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

Protocol Name	Steps	Kind of Attack	Depth	Time
ISO Symmetric Key One-Pass Unilateral Authentication	1	Replay	3	0.1
ISO Symmetric Key Two-Pass Mutual Authentication	2	Replay	3	0.1
Andrew Secure RPC	4	Type flaw	4	0.1
ISO One-Pass Unilateral Authentication with CCFs	1	Replay	3	0.1
ISO Two-Pass Mutual Authentication with CCFs	2	Replay	3	0.1
Needham Schroeder with Conventional Key	5	Replay	8	1.4
Denning Sacco Protocol	3	Type flaw	2	0.1
Otway Rees Protocol	4	Type flaw	2	0.1
Woo and Lam Authentication Protocol Π_1	5	Type flaw	3	0.1
Woo and Lam Authentication Protocol Π_2	5	Type flaw	3	0.1
Woo and Lam Authentication Protocol Π_3	5	Type flaw	3	0.1
Woo and Lam Authentication Protocol Π	5	Parallel-session	6	1.0
Woo and Lam Mutual Authentication Protocol	7	Parallel-session	6	1.1
Needham-Schroeder Signature Protocol	5	Replay	6	0.5
Neumann Stubblebine (initial part)	4	Type flaw	2	0.1
Neumann Stubblebine (complete)	7	Type flaw	5	0.1
Kehne Langendorfer Schoenwalder (rep. part)	3	Parallel-session	4	0.5
Kao Chow Repeated Authentication 1	4	Replay	7	2.1
Kao Chow Repeated Authentication 2	4	Replay	7	2.2
Kao Chow Repeated Authentication 3	4	Replay	7	2.3
ISO Public Key One-Pass Unilateral Authentication	1	Replay	3	0.1
ISO Public Key Two-Pass Mutual Authentication	2	Replay	3	0.1
Needham Schroeder Public Key (NSPK)	3	Man-in-the-middle	4	0.1
NSPK with key-server	7	Man-in-the-middle	5	4.7
SPLICE/AS	6	Replay	10	18.7
Hwang and Chen's Modified SPLICE/AS	2	Man-in-the-middle	2	0.1
Denning-Sacco Key Distribution w. P.K.	3	Man-in-the-middle	4	0.1
Shamir Rivest Adelman Three Pass	3	Type flaw	2	0.1
Encrypted Key Exchange (EKE)	5	Parallel-session	5	0.3
Davis-Swick Private Key Certificates 1	3	Replay	4	0.1
Davis-Swick Private Key Certificates 2	3	Replay	5	0.6
Davis-Swick Private Key Certificates 3	2	Replay	3	0.1
Davis-Swick Private Key Certificates 4	3	Replay	5	0.1

Figure 3: OFMC running times for the flawed protocols. Steps is the number of protocol steps, depth is the ply in the tree where the first attack is found, and times are in seconds.

Bibliography

[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. January 2002.

[2] A. Armando and L. Compagna. Automatic SAT-Compilation of Security Problems. January 2002.

[3] Y. Chevalier and L. Vigneron. A Tool for Lazy Verification of Security Protocols. In *Proceedings of the Automated Software Engineering Conference (ASE'01)*. IEEE Computer Society Press, 2001. Long version available as Technical Report A01-R-140, LORIA, Nancy (France).

[4] Y. Chevalier and L. Vigneron. Automated Unbounded Verification of Security Protocols. Technical Report 4369, LORIA, Vandoeuvre les Nancy, February 2002.

[5] 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`.

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

[7] G. Denker and J. Millen. CAPSL Intermediate Language. In N. Heintze and E. Clarke, editors, *Proceedings of Workshop on Formal Methods and Security Protocols (FMSP'99)*. URL for CAPSL and CIL: `http://www.csl.sri.com/~millen/capsl/`.

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

[9] 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.

[10] Genoa Group. Deliverable 3.6 : Final definition, implementation and experimentation with the SAT model-checker. 2002.

[11] Protheo Loria Group. Deliverable 3.5 : Final definition, implementation and experimentation with the constraint-logic model-checker. 2002.

[12] G. Lowe. Casper: a Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6(1):53–84, 1998. See also `http://www.mcs.le.ac.uk/~gl7/Security/Casper/`.

[13] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1):85–128, 1998.