AVISS — Automated Verification of Infinite State Systems

(IST-2000-26410)

Deliverable D3.1

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

1 Introduction

The objective of the third work-package WP3 of our project AVISS IST-2000-26410 is to experiment the chosen automated deduction techniques implemented in the prototype verification tool against the verification problems of the corpus [1] of security protocols.

The first step (*encoding*) is the definition of encodings that translates the protocol verification problems, obtained by applying the translator of WP2 to the corpus, into deduction problems falling into the scope of application of the chosen automated deduction techniques. (The architecture of the prototype is shown again in Figure 1.) These encodings, together with the translation mechanism set up in WP2, allows us to run the available automated deduction engines against the verification problems of the corpus (*experiments*). The experiments indicate ways to improve the encodings as well as the inference strategies implemented in the available automated deduction engines (*tuning*).

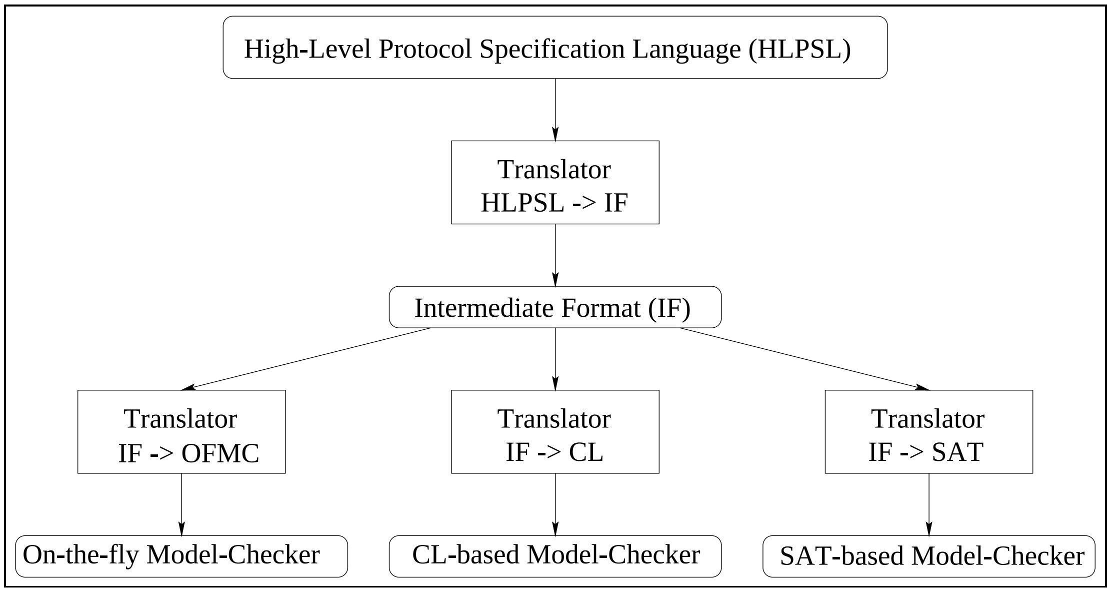

Figure 1: Architecture of the prototype verification tool

This deliverable D3.1 describes our implementation of task T3.1, i.e. define the encoding from the IF to the input format for the on-the-fly model checker based on lazy data-types, develop a prototype translator implementing the encoding, and experiment with problems from the corpus, tuning the encoding and/or the on-the-fly model checker.

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 described here, we will 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 will be examined using iterative deepening search to find a flaw-state. This construction is summarized in Figure 2.

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 into 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.

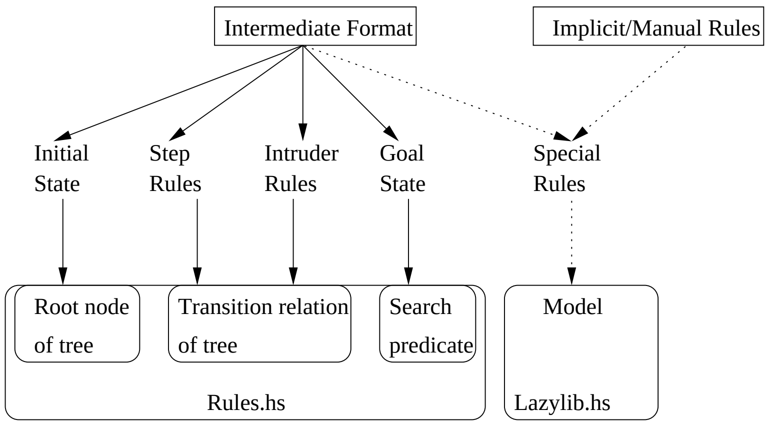

Figure 2: Translation of the IF-Transition-System into a search tree in Haskell.

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.

The files that are generated and used during this construction are summarized in Figure 3. `Rules.hs` is a Haskell file that is generated by the IF2OFMC. It contains all protocol specific information, namely the initial state, the (building blocks of the) successor relation and the description of the goal states. All other, not protocol-specific information, namely the basic data types, the construction of a tree from `Rules.hs`, and the search routines and heuristics are collected in the file `Lazylib.hs`. The file `Main.hs` contains the main routine and glues the other two files together. All three files are compiled by a Haskell compiler into binary machine code so that the actual search can be performed faster than in an interpreter environment. This search program runs until either an attack is found or the user stops the program.

In section 3 we describe the translation process performed by `IF2OFMC` (i.e. the generation of the file `Rules.hs` from the IF-description of a protocol). In section 4 we describe all static parts of the model, i.e. the remaining aspects of our approach that are protocol-independent. In section 5 we report on our first experiments with this approach.

3 Translation of the IF to Haskell

In the following we assume familiarity with the IF, as defined in deliverable D2.2. We describe the translation from IF to Haskell in a bottom-up fashion: we first describe a recursive data type for message terms and show on this basis how to encode states and state transitions in Haskell.

3.1 Messages

In the IF, an atomic message like a key, a nonce, or a principal name is declared by a name 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.

The problem with messages is, that 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 misuse, for example a name in place of key. This has the consequence that we cannot use the type system of Haskell to model the type of different kinds of messages since this

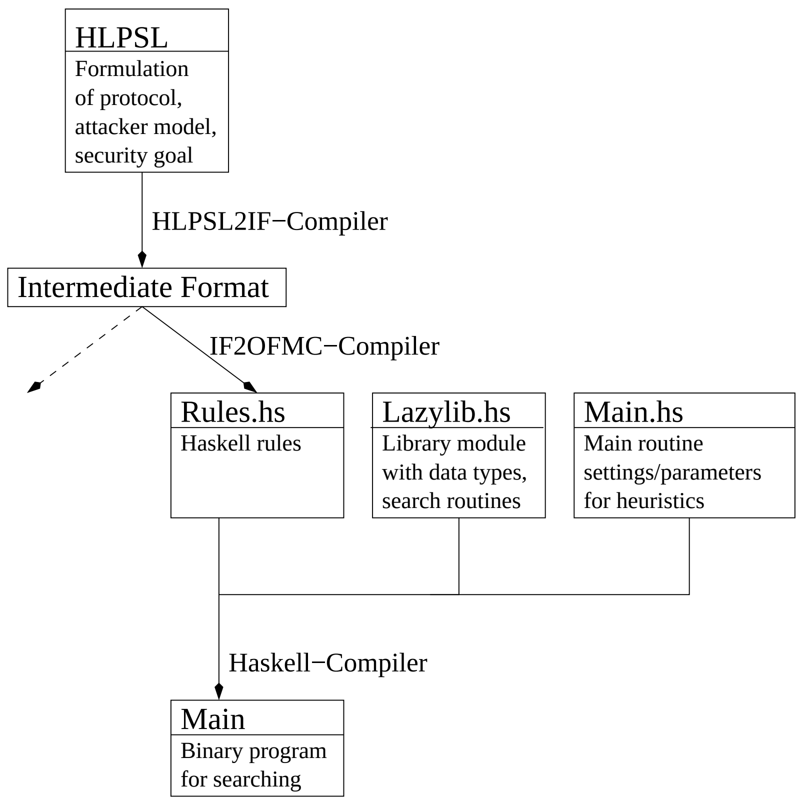
HLPSL
Formulation of protocol, attacker model, security goal
HLPSL2IF−Compiler
Intermediate Format
IF2OFMC−Compiler
Rules.hs
Haskell rules
Lazylib.hs
Library module with data types, search routines
Main.hs
Main routine settings/parameters for heuristics
Haskell−Compiler
Main
Binary program for searching

Figure 3: The files involved in the OFMC approach.

would enforce correct types for all message parts and hence exclude all type-flaw attacks from the model.

The IF handles the 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 `pk(ka)'` and `Crypt(pk(ka),xM)` he can derive `xM`. However rules that describe the behavior of agents are *blind* for this type information, e.g. `c(xKab,scrypt(xKab,xA))` just requires some item `xKab` (which is not necessarily a symmetric key) and another item `xA` (that is not necessarily the name of an agent) that is encrypted with the item `xKab`. (Recall that variable names start with the letter 'x'.)

The HLPSL2IF compiler by default produces transition rules in this *untyped* flavor, but it also has an option to produce a typed version of the rules, e.g. the above example would then be `c(sk(xKab),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 IF2OFMC compiler 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.

3.2 States

A state consists of a list of facts. A fact is either 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 is not yet received), intruder knowledge (terms the intruder has seen), or a secret (i.e. a variable that denotes the secret item and a session the secret is identified with).

Facts can be translated literally into Haskell. The only exception is the repeated occurrence of a variable in a pattern. For example, the following pattern is not allowed

```
M xA xA xB (Crypt xB xMsg)
```

while one might write

```
M xA xA' xB (Crypt xB' xMsg)
```

with the explicitly filtering all items such that `xA=xA'` and `xB=xB'`.

In the following, $[\![\cdot]\!]$ will denote the Haskell translation of a list of fact terms, and $[\![\cdot]\!]^*$ the corresponding translation for pattern matching, i.e. with no variable appearing more than once and appropriate equality constraints.

3.3 Rules

A rule has the form $LHS \Rightarrow RHS$. 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 this into Haskell by defining a function that takes the current state and returns a list of possible successor states (sometimes there are several — or no — possible successor states). We make use of advanced Haskell language features like list comprehension:

$$[\![\, LHS \Rightarrow RHS \,]\!] = \lambda\, \texttt{state}\ .\ \texttt{[}\ (\ [\![\, RHS \,]\!] \cup \texttt{state}\ ) \setminus [\![\, LHS \,]\!]\ \texttt{|}\ [\![\, LHS \,]\!]^* \texttt{ <- state]}$$

Intuitively, this says that we build the list of all matches between the requirements of the LHS and the current state and for each such match we build the successor state by removing the LHS terms and adding the RHS terms.

The successor relation of the tree is now the element-wise union of the successor functions that are generated from the rules in the IF.

4 Static Aspects of the Model

In this section we describe the aspects of the approach that are not protocol specific, e.g. how to model the properties of cryptographic keys. Several of these properties are modeled by rules that are part of the HLPSL2IF output. Since these rules are protocol independent and included identically in the IF-translation of every protocol, the IF2OFMC translator ignores them and uses a built-in part instead.

In general, our model is based on the Dolev-Yao intruder model [2], which allows the intruder to analyze messages in his knowledge as far as he has the appropriate keys (i.e. assuming perfect cryptography) and to arbitrarily compose new message from his knowledge including encryption with known keys.

4.1 Normalizing States

Among the basic properties of encryption is that the intruder can decompose all items he has seen if he has the appropriate key. We can handle such properties by normalizing each state, i.e. after any transition the appropriate decomposition rules are applied until a fixed-point is reached.

There are many other properties that can be handled this way:

- Idempotence: in a state, i.e. a list of facts, all duplicates can be eliminated.
- Simplifications of terms (e.g. `scrypt(xK,scrypt(xK,xM))=xM`).
- Secrecy goal transformation: secret-terms must be updated when a new protocol session starts.

4.2 Composing New Messages

There are a few other properties that are not so easy to handle. For instance the intruder can generate all kinds of compositions from his knowledge, and the set of items he can construct is infinite. Hence it is impossible to keep all compositions of known items in the state space.

A serious consequence of this is that we cannot use pattern matching for intruder knowledge in general: we need an additional check, if the required knowledge for forming a certain message could be composed from items that the intruder knows.

In the typed version (i.e. when all type-flaw attacks are excluded from the model) this can be drastically simplified: for every item in a message that the intruder can produce, there are only finitely many possibilities how to fill the slot. Hence one can completely describe all terms that the intruder can produce and that can be received by an honest agent using IF impersonate rules.

4.3 Composing Messages in the Untyped Version

In the untyped version one has also to consider a lot of messages that cannot be ruled out that easily. Consider the following protocol:

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

An attack against this protocol is:

1. $A \to B\times : N_A$

1'. $I_A \to B : N_A, N_A$

2. $B \to C\times : \{N_A, N_A\}_{K_K}$

2'. $I_C \to A : \{N_A, N_A\}_{K_K}$

This type flaw attack is based on the fact that the intruder can always compose known items from his knowledge. Since there is no obvious bound to the complexity of messages he can compose and that can lead to an attack in a particular protocol, we have to deal with infinitely many possible messages — according to the Dolev-Yao intruder model — and so the branching of the search tree is infinite.

In a first experimentation phase we have restricted this in the following way: We assume that the rules for impersonating specify all message components that the receiver can analyze. The intruder can fill these slots with items from his knowledge, however he cannot carry out any further compositions. This is not complete according to the Dolev-Yao model; however this approximation works fine with many protocols and is hence sufficient for a first prototype implementation.

A further complicated situation is matching composed terms: it should be possible to match `c(c(NA,A),B)` against `c(NA,c(A,B))` since coupling is associative. Currently we use an approximation by associating every term to the right before comparing, however this does not solve the problem completely: `c(KA,NA)` does not match `c(A,c(B,NA))`. Solving this problem will be subject of future work.

5 Experiments with the Prototype

We have started experimenting with the prototype implementation of the on-the-fly model checker by testing it against the protocols (and the known attacks upon them) in the corpus [1].

5.1 Heuristics

For some protocols, the naïve search described so far produces a search tree that is not feasible. Therefore we started to design some simple heuristics that help reduce the search tree, and will improve them in the next steps of the projects.

A first simple heuristic is to prune all those successors of a node that could (with the same instantiation of the variables) also be applied to the predecessor node; intuitively this means, we are performing those actions that use some *new* fact of the current state, so that the same transition could not be performed earlier. This heuristic is a simple kind of partial-order reduction, however it is not attack preserving (i.e. the pruned tree may be free of attacks while the original tree was not).

We plan to work systematically on the heuristics by distinguishing attack-preserving and non-attack-preserving heuristics. Attack-preserving heuristics allow one to completely forget about certain branches of the search tree, while non-attack-preserving heuristics like the one described above can only be used to focus the search to those parts of the tree that are most likely to contain a flaw — and the focus is constantly broadened the longer the search runs.

5.2 Running Times

As summarized in Figure 4, applying the above heuristic we get quite promising running times. Note that the typed version (where type-flaw attacks are not considered in the model) the times are much faster than in the untyped version; however many protocols have simple type-flaw attacks, while the typed version contains either no attacks (as in Lowe's fix of the NSPK) or considerably more complicated non-type-flaw attacks.

5.3 Manual Modifications

In some cases, manual modifications to the IF are necessary in order to find an attack. The is due to fact that the appropriate specifications cannot be formulated in the current version of HLPSL. Therefore HLPSL shall be extended accordingly in the following items:

- The correspondence goals in HLPSL specify mutual authentication of the agents, i.e. it is not possible that one agent can perform a complete run of the protocol while the other

Protocol	Attack type	depth	typed	untyped
Andrew Secure RPC	type flaw	10	-	< 1s
EKE	replay	13	3s	3s
Kao-Chao	replay	12	3min	?
Neumann-Stubblebine	type flaw	5	-	< 1s
NSPK	man-in-the-middle	10	1s	8s
NSPK w. Lowe's fix	type flaw	9	-	18s
Otway-Rees	type flaw	5	-	< 1s
Yahalom	type flaw	5	-	< 1s

Figure 4: Running times of the on-the-fly model checker for various protocols in typed in untyped version. (Depth denotes the ply in the tree at which the attack is first found.)

agent is still in its starting state (i.e. an intruder must have played his role). However many protocols (try to) provide authentication guarantees only in one direction, and it must hence be possible to specify this "half of a correspondence goal" in HLPSL.

- It should be possible to specify short-term secrets, i.e. secrets such that the protocol is still safe, even if the intruder gets hold of the short-term secret after a successful run of the protocol. This can be modeled by adding an appropriate term to the last step-rule of the protocol.

6 Outlook

The next months will be spent on improving and polishing the tool. There are three issues to work on:

1. The completeness of the tool: as pointed out, some aspects are handled by provisional implementations which rule out some attacks; such code must be replaced by a sound implementation.

2. Improvements in the search strategy. We see still much room for improvements here since especially in the untyped version the current search strategies blindly expand the messages the intruder can generate.

3. Cleaning the code: many algorithms are still implemented in an inefficient fashion. We expect a significant improvement in performance by replacing them with more efficient versions.

Bibliography

[1] J. Clark and J. Jacob. A survey of authentication protocol literature: Version, 1997.

[2] D. Dolev, A. Yao, t security, and p protocols. Ieee transactions on information theory, 1983.