Open Source Fixedpoint Model Checker

Short Manual

Sebastian Mödersheim
IBM Research Lab, Säumerstrasse 4, 8803 Rüschlikon, Switzerland
smo@zurich.ibm.com

November 2, 2009

1 Introduction

The *Open Source Fixedpoint Model Checker*, OFMC for short, is an extension and update of the *On-the-Fly Model Checker* that I developed together with Paul Hankes Drielsma and Boris Köpf at ETH Zurich [2]. The new version consists of two modules: the classic module for the bounded verification of security protocols and the new fixedpoint module for the verification of an unbounded number of sessions. Either module has its strengths and we therefore integrate them to obtain the advantages of both.

2 Installation

For the binary distributions, one only needs to ensure that the binary is found (as the command `ofmc`) in your favorite shell.[1]

To compile OFMC from sources, you need the *Glasgow Haskell Compiler* in a recent version (6.10.∗ should work) along with the lexer and parser generators for Haskell, *Alex* and *Happy*. All you need is included in the Haskell platform. Assuming these tools are available as commands `ghc`, `alex`, and `happy`, respectively, you can compile OFMC using the standard `make`. You can then run a small set of examples using `make test`.

3 IF: OFMC's Native Language

The *AVISPA Intermediate Format*, *IF* for short, is the native input language of OFMC. It is defined in [1]. In a nutshell, IF describes a state transition system where each state is a set of facts. One specifies an initial state and a set of transition rules. This gives defines a set of reachable states. Further one declares a set of attack states and a protocol is considered safe if no attack state

[1] On Windows, it is recommended to install a Unix-style shell like Cygwin or MinGW first.

is reachable. IF is a bit low-level and technical; it is thus recommended to use the new AnB input language (see next section) to specify the protocol and goals. Even if that is not entirely possible (because some aspects of a protocol cannot be formalized in AnB) one may use AnB as a starting point to generate the core of a specification. Another alternative is the AVISPA HLPSL language [4] which can be automatically translated to IF by the HLPSL2IF translator of the AVISPA tool (see `www.avispa-project.eu`).

OFMC has some extensions and restrictions with respect to the definition of [1]:

- In OFMC, the initial state may contain variables of type `agent` (or `message` in an untyped model). The meaning of this extension is that all these variables can be arbitrarily instantiated with agent names (including the name of the intruder `i`). Also, one can specify inequalities in the initial state such as `A/=i`. This extension allows for the specification of *symbolic sessions* section 7.
- OFMC can consider a model of custom symbols, their algebraic properties and the attached intruder model, using seperate theory files.
- OFMC requires that all transition rules have exactly one `state` fact on each side, i.e. correspond to the progress of one agent. We shall remove this restriction soon.

4 AnB: Alice and Bob notation

AnB is a simple and straightforward language for describing protocols. A detail description of the semantics is found in [6] and the channel notation is described in [7].

We illustrate the main points by my personal favorite, the H.530 protocol displayed in Figure 1.

Section `Types` In the `Types` section, we declare the type of all identifiers that are not builtin. Identifiers that start with an upper-case letter like `A` and `B` are variables. Identifiers that start with a lower-case letter like `s` are constants. The most general type is `Message`, all other types are (disjoint) subtypes of message. Note that one can consider untyped protocol models (to detect type-flaw attacks), but this is done by invoking OFMC with the option `-untyped` to make it ignore the type information. The untyped analysis is only possible in the classic module, currently. Thus we need the specification of all types.

Roles and Honest Roles In the example, we have three protocol roles, where the first two can be played by any agent in concrete protocol runs; more in detail, the variables `A` and `B` are instantiated during a particular protocol run with a constant of type `Agent`, this includes the intruder `i`. Note that we do not need to specify the concrete set of agents that populate our universe. The third role

```
Protocol: H530

Types: Agent A,B,s;
       Number X,Y,g,M,t1,t2,t3,t4,t5;
       Function sk,mac

Knowledge: A: A,s,sk(A,s),B,g,mac,t1,t2,t3,t4,t5;
   B: B,s,sk(B,s),g,mac,t1,t2,t3,t4,t5;
   s: A,B,s,sk,g,mac,t1,t2,t3,t4,t5

Actions:
A->B: A,B,exp(g,X),mac(sk(A,s),t1,A,B,exp(g,X))
B->s: A,B,exp(g,X),mac(sk(A,s),t1,A,B,exp(g,X)),
      B,exp(g,X),exp(g,Y),
      mac(sk(B,s),t2,A,B,exp(g,X),mac(sk(A,s),t1,A,B,exp(g,X)),
     B,exp(g,X),exp(g,Y))
s->B: B,A,mac(sk(A,s),t3,B,exp(g,X),exp(g,Y)),
      mac(sk(B,s),t4,B,A,mac(sk(A,s),t3,B,exp(g,X),exp(g,Y)))
B->A: B,A,exp(g,Y),mac(sk(A,s),t3,B,exp(g,X),exp(g,Y)),
      mac(exp(exp(g,X),Y),t5,B,A,exp(g,Y),mac(sk(A,s),t3,B,exp(g,X),exp(g,Y)))
A->B: {|M|}exp(exp(g,X),Y)

Goals:
B authenticates A on M
M secret between A,B
```

Figure 1: AnB specification of the H.530 protocol.

`s` cannot be instantiated and is rather played by one honest server named `s` in all sessions. Parties that act as a "trusted" third party (actually *honest* third party) are specified as constants like this.

Functions In the example we have declared `sk` and `mac` to be functions. Note that such function symbols are always unary functions on messages; by using the pair operator on the arguments we can, however, use them just like any n-ary function. By default, no further algebraic properties are attached to them. E.g. in this case the long-term shared key `sk(A,B)` of agents `A` and `B` differs from the `sk(B,A)`.[2] The `mac` function symbol is used here to model message authentication codes, i.e. a keyed hash function: a mac `mac(K,M)` can only be generated and verified knowing key `K` and message `M`, and one cannot recover `M` from `mac(K,M)`.

Section `Knowledge` For each role of the protocol, we specify a set of messages that the role knows before the execution of the protocol. In this case the agents know themselves, the server, the shared key with the server, and all the public constants and function symbols. Note that only the honest server knows the entire key-table `sk`,[3] while everybody may know the function `mac`.

An important condition is that the intruder knowledge may contain only variables of type agent (that will be instantiated in concrete protocol runs with constant agent names). All other variables, like the Diffie-Hellman exponents `X` and `Y` stand for values that will be freshly created during the execution of the protocol, namely by the first person that sends the first message containing them. In this case, thus, `X` is generated by `A` and `Y` is generated by `B`.

Section `Actions` The main part of the specification is the exchange of messages. Each receiver is the sender of the next message (if there is one). In a nutshell, the translator from AnB to IF will generate for each role one "program" from this exchange. In this translation we check that each message can be constructed by the respective agent from the knowledge that he or she has at that point, i.e. the initial knowledge, the received messages, and the values that the agent has freshly created. If a message cannot be constructed the translation will refuse the specification as not executable and give hints to what message cannot be constructed. The translator can automatically handle construction module properties of exponentiation and exclusive or. Note that the classic OFMC module allows for custom algebraic theories, the AnB translator has only these two algebraic operators in (i.e. a `-theory` specification has no effect on the AnB translator).

We note that *scryptkm* represents the symmetric encryption of message m with key k and *cryptkm* the asymmetric encryption. Concatenation is by default not associative and we interpret a, b, c as *pairapairbc* (right-associative).

[2] In fact using a pair of shared keys, one for each direction of a communication, is a good idea to avoid reflection attacks.

[3] In fact, one may alternatively give the server only `sk(A,s)` and `sk(B,s)`.

Channel notation While the default is the insecure communication medium, we may alternatively also the following kinds of channels (for details see [7]):

- $A \bullet\!\!\rightarrow B : M$ is an authentic channel,
- $A \rightarrow\!\!\bullet\, B : M$ is a confidential channel, and
- $A \bullet\!\!\rightarrow\!\!\bullet\, B : M$ is a secure channel.

Each of them may be used in a variant where sender or receiver is put in square brackets to indicate a pseudonymous channel, e.g. $[A] \bullet\!\!\rightarrow\!\!\bullet\, B : M$ would represent a secure channel where the sender is not authenticated; the point is that in several transmissions, $[A]$ is guaranteed to be the same (but unauthenticated) entity. This is the kind of channel that a TLS connection without client authentication provides.

Section `Goals` There are several ways to specify goals right now:

- The standard authentication goal is `B authenticates A on M`. This corresponds to Lowe's injective agreement [5].
- A weaker variants is `B weakly authenticates A on M` where replay is not considered as an attack. This corresponds to Lowe's non-injective agreement.
- `M secret between A,B,C` is the standard secrecy goal.
- There is another notation for goals using the channel notation:
  - $A \bullet\!\!\rightarrow B : M$ for `B authenticates A on M`.
  - $A \rightarrow\!\!\bullet\, B : M$ for a variant of `M secret between A,B`: in fact, as soon as A knows M, it is considered a secret and what B receives is not declared as a secret.
  - $A \bullet\!\!\rightarrow\!\!\bullet\, B : M$ for both $A \bullet\!\!\rightarrow B : M$ and $A \rightarrow\!\!\bullet\, B : M$.
- For the fixedpoint module we can currently only support two kinds of goals:
  - $A \rightarrow\!\!\bullet\, B : M$ (i.e not the `secret between` variant).
  - $BweaklyauthenticatesAonM$ where M must be a freshly created value of A.

Also note that the channel notation of goals allows for $[A] \bullet\!\!\rightarrow\!\!\bullet\, B : M$, e.g. as the goal of TLS without client authentication, but here we need to guess what the "pseudonym" of the client actually is (in TLS it is the pre-master secret as explained in the extended version of [7]). This mechanism is not perfect yet (will be improved) and for these goals we thus advise users to check the IF file whether the translation is appropriate.

5 Classic and Fixedpoint Module

Classic Module The classic OFMC performs verification for a bounded number of sesssions, i.e. the actions of honest agents are bounded. The intruder can in this case be unbounded (generate arbitrarily complex messages from his knowledge and send them to any agent) and we can handle algebraic properties and many other features.

Iterative Bounded Sessions The AnB translator can generate IF files for a given number of *symbolic* sessions, i.e. a set of uninstantiated agents at the initial state of the protocol execution, one agent for each of the roles. By default, we run the classic OFMC with the following configuration: we begin the verification for 1 session. If an attack is found, we stop and print the attack trace. If no attack is found, we increase the number of sessions and start the verification again. This process is repeated until an attack is found or the user cancels the process. In other words, for a correct protocol, this will never stop.

Note that this iteration is only possible for AnB files since IF files have a fixed set of sessions built-in for the initial state.[4]

The Fixedpint Module The fixedpoint module is a novel module that uses abstract interpretation and over-approximation to verify protocols, similar to tools like `ProVerif` and `TA4SP`. This allows for the verification without limiting the number of sessions, but we must assume a strictly typed model (thus bounding the intruder) to ensure termination, we need to check in a free algebra, and also the fixedpoint approach is not applicable to protocols that use negative conditions on transitions (which can be specified in IF but not in AnB). Moreover, when the fixed-point module detects an attack, this may be caused by an over-approximation and is not necessarily an attack to the original protocol description. In this case, the fixedpoint module automatically refines the abstraction and restart the verification process. This is repeated until the protocol is either verified or the strategy of OFMC sees no further room for refinement. In the latter case, the fixedpoint verification fails (which does not imply that the protocol is unsafe) and the abstract attack of the last refinement round is presented.

Both Modules in Parallel To combine the strengths of both of approaches, we have chosen to integrate them in the following way:

- Both modules are started on the problem in parallel.
- If the classic module finds and attack, the classic module *wins* and the attack trace is printed (the fixedpoint verification is killed).
- If the fixedpoint module verifies the protocol, the fixedpoint module *wins* and the classic module verification process is killed.

[4] It is actually possible to declare initialization theories that create new state facts, but the classic OFMC does not allow that currently.

- In all other cases we have a *tie*: we can verify the protocol only for a bounded number of sessions and it may be correct in general, but we cannot tell.

This setup follows the idea "first definitive answer counts". However note that it is very well possible that the fixedpoint module can verify a protocol and the classic module finds an attack:

- The fixedpoint module considers only weak authentication and secrecy. You may specify strong authentication goals that are only considered in the classic module.
- The fixedpoint module works in a strictly typed model, and the classic module may find type-flaw attacks.
- The fixedpoint module works in the free algebra, while the classic module considers algebraic properties.
- There may be bugs.

For all these reasons, we suggest that one may ensure that both modes have analyzed the protocol for a sufficient number of time. You can always choose to run a single module with the options `-classic` and `-fp`.

6 Isabelle Proofs

Using the new fixedpoint module, we can generate proofs for the interactive theorem prover *Isabelle* [3]. The idea is the following. The protocol model of OFMC and similar complex verification tools may suffer from implementation bugs so that in the worst case the tool could accept some incorrect protocols as being correct. These risks of errors are also present, but considerably smaller, when using an LCF-style theorem prover like Isabelle. The interactive security proof, however, requires a lot of expertise and time.

The new connection Isabelle/OFMC combines the advantages of both worlds by using the representation of the over-approximated search space of OFMC's fixedpoint module as a "proof idea" in Isabelle. Thus, we devise proof tactics for Isabelle that generate the correctness proof of the protocol from the output of OFMC. In the worst case, these tactics fail to construct a proof, namely when the representation of the search space is for some reason incorrect. However, when they succeed, the correctness only relies on the basic model and the Isabelle core.

When the verification with the fixedpoint module is successfull, we can use the flag `-ot Isa` to generate a "fixedpoint" file from which Isabelle/OFMC can generate the Isabelle proof. Isabelle/OFMC is still in the stage of a developer version. It can also be downloaded from the AVANTSSAR homepage (you need to obtain also an installation of Isabelle

`http://www.cl.cam.ac.uk/research/hvg/Isabelle/`

7 Symbolic Sessions

A scenario is a finite number of sessions, where a session is an instantiation of all protocol roles agent names. The number of scenarios grows exponentially in the number of sessions (for finite scenarios, one can bound the number of agent names). In HLPSL, the user must specify each scenario manually and run it through the AVISPA tool, in order to check that the protocol has no attacks for a given number of sessions.

OFMC allows one to specify symbolic sessions which involves specifying just one scenario – with agent names replaced by *variables*. During the search for an attack, OFMC instantiates the agent names when necessary (lazily).

In the example file of the SRP protocol, we have two roles User and Host, and we have specified the following *symbolic scenario* (where $User$, $User2$, $Host$, and $Host2$ are of type *Agent*):

```
initial_state init1 :=

 ...
 state_srp_user(User,Host,...)
 state_srp_host(Host,...)
 state_srp_user(User2,Host2,...)
 state_srp_host(Host2,...)

 & User/=Host
 & Host/=i
 & User /=Host2
 & User2/=Host2
 & User2/=Host
 & Host2/=i
```

This specifies two sessions, one between $User$ and $Host$, and one between $User2$ and $Host2$. Observe that in this specification the $Host$ is independent from the name of the client, as the $Host$ will accept communication from everyone. Using the inequalities, we can further require that hosts and users are disjoint (i.e. no user in one session can be a host in another session) and that all hosts are honest (unequal intruder). Such a specification is not mandatory, but some protocols require that certain roles can only be played by honest users, for instance.

8 Session Compilation

Session compilation is a feature that has been supported by OFMC for quite a while, but has lead to many questions, hence this extra section.

When specifying the option `-sessco`, OFMC 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" session

between honest agents. In the case certain steps cannot be executed by any honest agent, OFMC reports that the protocol is not executable and stops. If the executability check is successful, then the normal search with an active intruder is started, with the only difference that the intruder initially knows all the messages exchanged by the honest agents in the passive intruder phase.

This is helpful both for quickly finding replay attacks (rather than specifying lots of parallel sessions), but also for checking the sanity of the specification, namely that at least the 'legal execution' of the protocol is possible.

We recommend to check each protocol specification with option sessco first, to see if it is executable in the model of OFMC. If OFMC replies that it is not the case, then one should try to simulate the legal execution (the way the protocol was meant to be executed) using the path option. At some point, OFMC will not offer the next step of the legal execution, and that's the first point where probably a mistake in the rules has occurred. Indeed such debugging of specifications is not very convenient, and we hope to offer you soon a more improved option to inspect protocol specifications.

If one role can loop (i.e. remain in the same control state forever and make infinitely many steps), sessco is not possible (and OFMC aborts with an error message), but then the search in OFMC does not terminate either, unless you specify a depth bound for the search.

In any case, when it is not possible to establish executability of the protocol, one should still check the protocol without the sessco option. In this case, one can manually perform the session compilation by adding to the initial intruder knowledge the messages of one run of the protocol with the fresh data replaced by fresh constants.

References

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

[2] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.

[3] A. Brucker and S. Mödersheim. Integrating automated and interactive protocol verification. In *FAST 2009*. To appear, extended version available as IBM Research Report RZ3750.

[4] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. *A High Level Protocol Specification Language for Industrial Security-Sensitive Protocols*, volume 180 of *Automated Software Engineering*, pages 193–205. Austrian Computer Society, Austria, September 2004.

[5] G. Lowe. A hierarchy of authentication specifications. In *Proceedings of CSFW 10*, pages 31–43. IEEE Computer Society Press, 1997.

[6] S. Mödersheim. Algebraic Properties in Alice and Bob Notation. In *Proc. Ares'09*, Full version: T. Rep. RZ3709, IBM Zurich Research Lab, 2008, `domino.research.ibm.com/library/cyberdig.nsf`.

[7] S. Mödersheim and L. Viganò. Secure Pseudonymous Channels. In *ESORICS 2009*, LNCS 5789, 2009.