Tutorial: Hands-On Examples OFMC

Patrick Schaller
ETH Zürich

October 3, 2006

Introduction

This tutorial is an addendum to a theoretical introduction to OFMC. We expect from the reader some knowledge about the theoretical background on OFMC in order to understand the techniques discussed here. The main purpose of this tutorial is to show the reader how to use OFMC on some basic examples of protocols and their goals. After this tutorial the reader should know how a protocol has to be entered into OFMC, what the options of the tool are, and what kind of results you can get with OFMC.

It will not be possible to discuss all the technical details of OFMC and the specification language used to model protocols. For further information we refer to the web pages of the avispa project `www.avispa-project.org`, where you will find all the publications related to OFMC (and the other tools developed in the AVISPA project) and more detailed tutorials on the specification languages.

OFMC was developed as part of the AVISPA project by Sebastian Mödersheim. The tool is written in Haskell and the source code is freely available on Sebastian's homepage (`http://www.inf.ethz.ch/personal/moederss/`).

Overview

Modeling Protocols

From the theoretical point of view we consider agents participating in a protocol run as statemachines, where the state variables describe precisely the state of an agent in terms of its knowledge of its keys, nonces etc. A protocol executed by a number of agents defines the state transitions of the participants. In order to analyze the security of a given protocol we furthermore must define the security properties of interest. This is done by defining those states that represent a violation of a secrity property the protocol is intended to achieve.
For example, the goal of a protocol could be that two agents A and B share a key K_{AB} after the protocol and that no one else knows this key. So one possible

attack state of this protocol would be that K_{AB} is known by the intruder.
What OFMC does for us, is to search all the state space (generated by the initial state and the state transitions) in a very efficient way for possible attack states, i.e. OFMC checks whether it is possible to reach a state that violates a security property, by starting from the initial state and following the transition rules given by the protocol.

It should be clear that modeling plays an important role when model checking security protocols. The whole analysis depends on chosing a reasonable way to model the protocol and the agents involved. On the one hand, we want to abstract away as much as possible (to keep the state space small), on the other hand we have to be careful not to abstract away possible attacks we are looking for.
You should always be aware, that like with all formal methods, the results mainly depend on the model you have chosen to analyze the environment of interest.

Modeling Languages

There are two main input languages that are used to describe protocols for OFMC (and all the protocol checkers developed in the AVISPA project):

- HLPSL (High Level Protocol Specification Language)
- IF (Intermediate Format)

The main idea is to use HLPSL as a specification language for the user and to translate the HLPSL-specification into the intermediate format (IF) afterwards that is the input language of the tools. The tool HLPSL2IF is the translator from HLPSL to the IF format and can be downloaded on the AVISPA webpage. However, since this should be an introduction to OFMC and we have only a limited amount of time, we decided to explain things directly in the IF-format. From our point of view IF is 'closer' to the tool and helps you to understand the main ideas of OFMC in a better way than it would the case for HLPSL.

Syntax and Semantics of the Intermediate Format

We will introduce the syntax and the semantics of IF with the example of NSPK. For the list of the main syntax constructors see the appendix. More information about IF its syntax and semantics can be found on the AVISPA project home page.

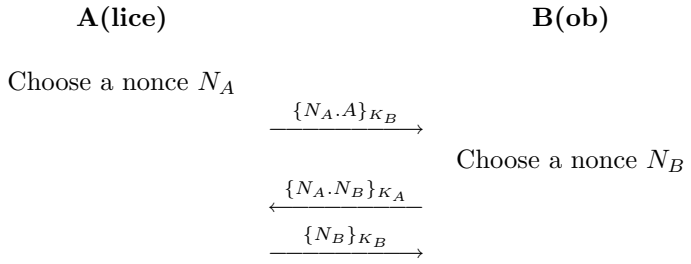

Figure 1: NSPK Protocol

The `types` Section

First of all we have to determine what types the terms have that are used in the protocol. Informally this would be:

Agent Names: to differentiate between the different participants.

Public Keys: the keys used by the agents

Natural Numbers: we will need them in order to differentiate between different sessions.

Nonces: freshly created data items, not used in the protocol run so far.

In the Intermediate Format this will look as follows (note that variables starting with upper case letters represent variables, whereas variables starting with lower case letters represent ground terms):

```
section types:

A,B,a,b,i: agent
KA,KB,ka,kb,ki: public_key
0,1,2: nat
SID: nat
NA,NB,na,nb,ni: nonces
```

`SID` will be needed to differentiate between sessions. The numbers will be needed to model the states the agents are in.

The `inits` section

This section of the IF file specifies the initial state of the protocol to be model checked. Here we define what all the agents know before running the protocol. For the NSPK protocol the `init` section is given as:

```
section inits:

initial_state init1:=
iknows(start).
iknows(a).
iknows(b).
iknows(ka).
iknows(kb).
iknows(i).
iknows(ki).
iknows(inv(ki)).
%session 1: A=a, B=b, KA=ka, KB=kb
state_Alice(a,b,0,ka,kb,ni,ni,1).
state_Bob(b,a,0,kb,ka,ni,ni,1).
%session 2: A=a, B=i, KA=ka, KB=ki
state_Alice(a,i,0,ka,ki,ni,ni,2)
```

The first part with all the `iknows` facts describes what the intruder knows at the start of the protocol run. The first term `iknows(start)` is used to let the intruder start a new session of the protocol; you should always include this. The reason therefore is that it enables the intruder in every step of the protocol execution to start a new session (possibly with another agent).
The lines starting with `%` are comments.
In the second part, we describe the states of the agents involved in the protocol run. The first parameter denotes the current state the agents are in. Then the agents names and the public keys of the agents follow. The following parameters `ni` are place holders: for the nonces they create and receive during protocol execution. In the final entry we have a so called Session-ID: they are used to differentiate between several parallel sessions of the same agents.

The `rules` section

After we have defined the agent's state variables and their initial knowledge, we have to define how the state variables of the agents change in a protocol run. Therefore we define the transition rules for each agent. In the Intermediate Format this is done in the `rules` section.

Together with the transition rules we have to define predicates about security properties of terms that are created or sent in such a transition rule. The reason why we need them in the in the transition rules is the following:
Recall that we model the protocol run by a state transition system and that we want to find security violations by checking if there will be a state reached (when executing the protocol), that would violate one of our goals. In order to achieve this, we have to 'mark the critical terms' of the protocol, for example, if an agent sends a message `m` that should be a secret between him and his communication partner, we have to mark this message a *secrect* between these two

agents, therefore we would use the keyword `secret`. Similar for authentication goals we have the keywords `witness` and `request`, for an explanation of the exact meaning of these expressions see the next section about attack states.

In our example we formalize the NSPK protocol in the following way:

```
section rules:

step step0 (SID):=
 state_Alice(A,B,0,KA,KB,ni,ni,SID)
 =[exists NA]=>
 state_Alice(A,B,1,KA,KB,NA,ni,SID).
 iknows(crypt(KB,pair(NA,A))).
 secret(NA,B).
 witness(A,B,na,NA)

step step1 (SID):=
 state_Alice(A,B,1,KA,KB,NA,ni,SID).
 iknows(crypt(KA,pair(NA,NB))).
 =>
 state_Alice(A,B,2,KA,KB,NA,NB,SID).
 iknows(crypt(KB,NB)).
 request(A,B,nb,NB,SID)

step step2 (SID):=
 state_Bob(B,A,0,KB,KA,ni,ni,SID).
 iknows(crypt(KB,pair(NA,A)))
 =[exists NB]=>
 state_Bob(B,A,1,KB,KA,NA,NB,SID).
 iknows(crypt(KA,pair(NA,NB))).
 secret(NB,A).
 witness(B,A,nb,NB)

step step3 (SID):=
 state_Bob(B,A,1,KB,KA,NA,NB,SID).
 iknows(crypt(KB,NB))
 =>
 state_Bob(B,A,2,KB,KA,NA,NB,SID).
 request(B,A,na,NA,SID)
```

Note that the order in which we place the state transition doesn't matter. Here we first wrote down all the transitions of Alice and then the transitions of Bob. We could also choose the more natural way of describing the protocol run as it really happens, by interleaving the transitions of Alice and those of Bob. As you may have heard in the theory session, OFMC is built on term rewriting. These transition rules correspond to the rewrite rules of the term rewriting sys-

tem.

The attack_states section

Finally we have to formalize the goals that should be achieved by the protocol to be model-checked. In the case of our example protocol, NSPK, the goals are:

- Alice wants to authenticate Bob.
- Bob wants to authenticate Alice.
- The nonces remain secrets between Alice and Bob.

To be able to find protocol runs that violated any of these goals, we have to define the states that represent violations of these goals. So when OFMC builds the state space, it must be able to recognize when one of the goals is not fullfilled anymore. So that it can stop building the state tree and return the trace that led to such a state.

In the Intermediate Format this looks the following way:

```
section attack_states:

%terms defined to be a secret shared by the honest agents
attack_state secrecy_of_NX(M,B):=
 iknows(M).
 secret(M,B) &
 not(equal(i,B))

%weak authentication
attack_state authenticate_Alice_on_na (A,B,Na,NA,SID) :=
request(B,A,Na,NA,SID) &
not(witness(A,B,Na,NA)) &
not(equal(A,i))

%weak authentication
attack_state authenticate_Bob_on_nb (B,A,Nb,NB,SID) :=
request(A,B,Nb,NB,SID) &
not(witness(B,A,Nb,NB)) &
not(equal(B,i))

%replay protection
attack_state replay_protection_on (A,B,M,SID1,SID2) :=
request(B,A,Nb,M,SID1).
request(B,A,Nb,M,SID2) &
not(equal(SID1,SID2)) &
```

```
not(equal(A,i))
```

First we want to have a look, on how to interpret the `witness` and `request` statements:

`witness`: A statement like `witness(A,B,na,NA)` has to be read as:'I am agent `A` and I want to agree with agent `B` on the value of the variable `NA` to be `na`.'

`request`: The corresponding statement `request(B,A,na,NA,SID)` means:'I am agent `B`, I accept the value of `NA` to be `na` for the session `SID` and I rely on the guarantee that agent `A` exists and that we agree on that value.'.

According to these interpretations we can now analyze that attack states in our protocol specification:

`secrecy_of_NX(M,A)`: This state is reached if there is a message `M` that is known by the intruder (`iknows(M)`) and was declared to be a secret between the 'creator' of `M` (in our case this would be Alice) and the agent substituted for the variable `B` (in our case Bob). Furthermore we don't want the role `B` to be played by the intruder (in this case the goal would be senseless since then the intruder will have to know the term for `M`).

`authenticate_Alice_on_na`: According to the interpretation we gave above for the request and witness statements, this attack state would be reached if there would exist the statement `request(B,A,Na,NA,SID)` in the state space (of a protocol run), but the according witness statement `witness(B,A,Nb,NB)` would not exist, i.e. the agent in the role of `B` would accept a value for `NA` to be shared with an agent in the role `A`, but the agent `A` would never have issued such a value with the intention to agree with the agent `B` on it. Furthermore we require that the role `A` is not played by the intruder, since then the condition would be violated be definition.

`replay_protection_on`: The last attack state in the definition above searches for replay attacks in the state space. According to Lowe's *Hierarchy of Authentication* the authetication property guaranteed by preventing replays would be *injective agreement*, i.e. there is a one-to-one relationship between the two agents' runs. Here you see, why we need the `SID` (Session ID) as a parameter for the request statement. If this attack state would become true in the execution of the protocol, the agent in the role `B` would rely on a certain object to have been created by the agent in the role `A`. Since the `SID`s would be the same, the agent would rely on the same value in two (independent) sessions, which would imply a replay attack.

The whole IF-file for NSPK

Putting all the sections together, we get the following IF-File (NSPK.if):

```
section types:

A,B,a,b,i: agent
KA,KB,ka,kb,ki: public_key
0,1,2: nat
SID: nat
NA,NB,na,nb,ni: nonces


section inits:

initial_state init1:=
iknows(start).
iknows(a).
iknows(b).
iknows(ka).
iknows(kb).
iknows(i).
iknows(ki).
iknows(inv(ki)).
%session 1: A=a, B=b, KA=ka, KB=kb
state_Alice(a,b,0,ka,kb,ni,ni,1).
state_Bob(b,a,0,kb,ka,ni,ni,1).
%session 2: A=a, B=i, KA=ka, KB=ki
state_Alice(a,i,0,ka,ki,ni,ni,2)


section rules:

step step0 (SID):=
 state_Alice(A,B,0,KA,KB,ni,ni,SID)
 =[exists NA]=>
 state_Alice(A,B,1,KA,KB,NA,ni,SID).
 iknows(crypt(KB,pair(NA,A))).
 secret(NA,B).
 witness(A,B,na,NA)

step step1 (SID):=
 state_Alice(A,B,1,KA,KB,NA,ni,SID).
 iknows(crypt(KA,pair(NA,NB))).
 =>
 state_Alice(A,B,2,KA,KB,NA,NB,SID).
 iknows(crypt(KB,NB)).
 request(A,B,nb,NB,SID)

step step2 (SID):=
```

```
 state_Bob(B,A,0,KB,KA,ni,ni,SID).
 iknows(crypt(KB,pair(NA,A)))
 =[exists NB]=>
 state_Bob(B,A,1,KB,KA,NA,NB,SID).
 iknows(crypt(KA,pair(NA,NB))).
 secret(NB,A).
 witness(B,A,nb,NB)

step step3 (SID):=
 state_Bob(B,A,1,KB,KA,NA,NB,SID).
 iknows(crypt(KB,NB))
 =>
 state_Bob(B,A,2,KB,KA,NA,NB,SID).
 request(B,A,na,NA,SID)


section attack_states:

%terms defined to be a secret shared by the honest agents
attack_state secrecy_of_NX(M,B):=
 iknows(M).
 secret(M,B) &
 not(equal(i,B))

%weak authentication
attack_state authenticate_Alice_on_na (A,B,Na,NA,SID) :=
request(B,A,Na,NA,SID) &
not(witness(A,B,Na,NA)) &
not(equal(A,i))

%weak authentication
attack_state authenticate_Bob_on_nb (B,A,Nb,NB,SID) :=
request(A,B,Nb,NB,SID) &
not(witness(B,A,Nb,NB)) &
not(equal(B,i))

%replay protection
attack_state replay_protection_on (A,B,M,SID1,SID2) :=
request(B,A,Nb,M,SID1).
request(B,A,Nb,M,SID2) &
not(equal(SID1,SID2)) &
not(equal(A,i))
```

Using OFMC

In this tutorial we will show how to use OFMC on the command line. There is graphical interface for the whole AVISPA suite of model checkers (see `www.avispa-project.org`, where they have a web interface). But we will be using OFMC directly, applying it to the IF-file, constructed above.

Basic Usage

Assume that you have an IF-file, for example the IF-file NSPK.if, that we constructed above and this file is in the same directory as OFMC. The following command will start OFMC with the input-file NSPK.if:

```
>./ofmc NSPK.if
```

The corresponding output will look like:

```
% OFMC
% Version of 2006/03/16
SUMMARY
  UNSAFE
DETAILS
  ATTACK_FOUND
PROTOCOL
  /home/schapatr/privrep/lectures/FIRSTschool/tutorial/NSPK.if
GOAL
  secrecy_of_NX
BACKEND
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.02s
  visitedNodes: 9 nodes
  depth: 2 plies
ATTACK TRACE
(a,1) -> i: {nonces(NA(1)).a}_ki
i -> (b,1): {nonces(NA(1)).a}_kb
(b,1) -> i: {nonces(NA(1)).nonces(NB(2))}_ka
i -> (a,1): {nonces(NA(1)).nonces(NB(2))}_ka
(a,1) -> i: {nonces(NB(2))}_ki
i -> (i,17): nonces(NB(2))
i -> (i,17): nonces(NB(2))
```

```
% Reached State:
%
% secret(nonces(NB(2)),a)
% witness(b,a,nonces(nb),nonces(NB(2)))
% secret(nonces(NA(1)),i)
% witness(a,i,nonces(na),nonces(NA(1)))
% state_Alice(a,i,2,ka,ki,nonces(NA(1)),nonces(NB(2)),1)
% state_Bob(b,a,1,kb,ka,nonces(NA(1)),nonces(NB(2)),1)
% state_Alice(a,b,0,ka,kb,nonces(ni),nonces(ni),1)
% request(a,i,nonces(nb),nonces(NB(2)),1)
```

As you can see, OFMC declares the protocol as unsafe and shows the attack trace to reach the unsafe state.

Useful Options

By typing `./ofmc --help` you get a list of possible options:

```
>./ofmc --help
ofmc: OFMC
usage: ofmc <IF File> [-theory <Theory file>] [-sessco] [-untyped] [-d <DEPTH>] [-p <PATH>]
where -theory  runs the protocol analysis with an custom algebraic theory
      -sessco performs an executability check and session compilation
      -untyped ignores all type-declarations
      <DEPTH> (int) is the maximum search depth
      <PATH>  (white-space separated list of ints)
              is a path in the search tree to visit
              by the indices of the successors to follow.
```

The options `-sessco` and `-p <PATH>` are very helpful in order to determine if the protocol specification in the IF-file really describes the protocol as you want to describe it. The `-sessco`-option checks if the protocol specified in the IF-file indeed is executable.
The `-p`-option lets you decide what the next step in the execution path should be, which is also very helpful when searching mistakes in protocol specifications.

For example `>./ofmc NSPK.if -p` would give something like:

```
(0)
(a,1) -> i: {nonces(NA(1)).a}_kb

(1)
(a,1) -> i: {nonces(NA(1)).a}_ki
```

```
(2)
i -> (b,1): {nonces(NA(8)).a}_kb
(b,1) -> i: {nonces(NA(8)).nonces(NB(1))}_ka

(3)
i -> (b,1): {nonces(NA(9)).a}_kb
(b,1) -> i: {nonces(NA(9)).nonces(NB(1))}_ka

(4)
i -> (b,1): {nonces(x227).a}_kb
(b,1) -> i: {nonces(x227).nonces(NB(1))}_ka
```

In this example, step `0` would correspond to the session, where agent `A` wants to open a session with agent `B` according to the normal execution of the protocol and sends {`nonces(NA(1)).a`}_`kb` to `i` (the intruder = network, remember step compression).
The second step `(1)` corresponds to the session where the agent `A` wants to start a run with the intruder and therefore sends {`nonces(NA(1)).a`}_`ki`.
The next two steps show instances, where the intruder would generate a nonce and just send it to agent `B` (according to the rewrite rule step 2) this would be a possible option.
The last possibility (`(4)`) would be the one where the intruder starts a symbolic session (lazy intruder) and trys to figure out later, what a useful instantiatian of the variable (`nonces(x227)`) would be.

With the help of this option you can walk by your own through the state tree and decide at every node, which step to take next. So the command `./ofmc NSPK.if -p 0 1` would show where we get if we first execute `step 0` and afterwards (as a next step) choose the option `step 1`.

Appendix: The Syntax of the Intermediate Format

```
Prelude ::=  TypeSymbolsSection
             SignatureSection
             TypesSection
             EquationsSection
             IntruderSection

IFFile  ::=  SignatureSection
             TypesSection
             InitsSection
             RulesSection
             GoalsSection

TypeSymbolsSection ::= "section typeSymbols:" TypeList
SignatureSection   ::= "section signature:" SignatureSection0
TypesSection       ::= "section types:" TypeDeclaration*
EquationsSection   ::= "section equations:" Equation*
InitsSection       ::= "section inits:" ("initial_state"
                         Identifier ":="  State )+
RulesSection       ::= "section rules:" RulesDeclaration*
GoalsSection       ::= "section goals:" GoalDeclaration*
IntruderSection    ::= "section intruder:" RulesDeclaration*

RulesDeclaration ::= "step" Identifier "(" VariableList ")"
                       ":=" CNState ExistsVar? "=>" State
State   ::= Fact ("." Fact)*
CNState ::= NState ConditionList
ConditionList ::= ("&" Condition)*
Condition ::=  "equal(" Term "," Term ")"
            |  "leq(" Term "," Term ")"
            |  "not(" Condition ")"
NState ::= NFact ("." Nfact)*
NFact ::= Fact | "not(" Fact ")"
Fact ::= IF_Fact "(" TermList ")"

ExistsVar ::= "=[exists" VariableList "]"

GoalDeclaration ::= "goal" Identifier "(" VariableList ")"
                     ":=" CNState

Equation ::= Term "=" Term
Term ::= AtomicTerm
       | ComposedTerm
```

```
AtomicTerm ::= Constant
             | Variable
ComposedTerm ::= IF_Operator "(" TermList ")"

Constant ::= [a-z] [a-zA-Z0-9_]* | [0-9]+
Variable ::= [A-Z_] [a-zA-Z0-9_]*
Identifier ::= Constant

TypeDeclaration ::= AtomicTermList ":" Type
Type ::= IF_Type
       | IF_Operator "(" TypeList ")"
       | "{" ConstantList "}"

SignatureSection0 ::= SuperTypeDeclaration*
                    | FunctionDeclaration*
                    | PredicateDeclaration*
TypeStar ::= Type
           | Type "*" TypeStar
SuperTypeDeclaration ::= IF_Type ">" IF_Type
FunctionDeclaration ::= IF_Operator ":" TypeStar "->" Type
PredicateDeclaration ::= IF_Operator ":" TypeStar "->" Type

VariableList ::= Variable ("," Variable )*
TermList ::= Term ("," Term )*
TypeList ::= Type ("," Type )*
AtomicTermList ::= AtomicTerm ("," AtomicTerm)*

IF_Fact ::= "state_" Identifier | Identifier
IF_Operator ::= Identifier
IF_Type ::= Identifier
```

Structure of an IF File

An IF file (and, similarly, a prelude file) consists of a sequence of sections.

Section Type Symbols (`TypeSymbolsSection`). In this section, all basic (message) types (like *nonce*) are declared.

In the sections signature (`SignatureSection`) and types (`TypesSection`), the types of variables, constants, function symbols and fact symbols are declared.

Section Signature (`SignatureSection`). This section contains declarations of the used function and fact symbols, and, more specifically, their types. Also it contains subtype declarations. This section is not necessary to define the semantics of the IF (the arity of the function and fact symbols is implicit when they are used consistently), but the type information can be helpful for some back-ends. Almost all the information in this

section is protocol independent (and hence part of the prelude, not of the concrete IF file), however there is one exception: the signature of a state-fact describing the state of an honest agent is protocol-dependent.

Section Types (`TypesSection`). In this section, the types for all constants and variables can be specified. This implies that throughout an IF file an identifier cannot be used with two different types (while the scope of each variable is limited to the rule it appears in). Note that one may leave unspecified the type of some or all identifiers in order to obtain an *untyped model*.

Section Equations (`EquationsSection`). In this section, algebraic properties of the function symbols are specified.

The sections inits (`InitsSection`), rules (`RulesSection`), and intruder (`IntruderSection`) describe a transition system, and section goals (`GoalsSection`) describes a goal (or attack) predicate on states.

Section Inits (`InitsSection`). In this section, we specify one or more initial states of the protocol, and thus consider several parallel runs of the protocol. Deliverable D3.3 on *session instances* will illustrate how this section can be generated automatically by the HLPSL2IF translator.

Section Rules (`RulesSection`). In this section, we specify the transition rules of the honest agents executing the protocol. Note that, with respect to previous versions of the IF, we have extended the rules with conditions and negative facts.

For the declaration of rules, we also use the following syntactic sugar. We assume that the *iknows* fact (*iknows*(M) means that the intruder knows the message M) is persistent, in the sense that if an *iknows* fact holds in a state, then it holds in all successor states (i.e. the intruder never forgets messages). Therefore, if an *iknows* fact appears in the left-hand side (LHS) of a rule, it should also be contained in the rule's right-hand side (RHS). To simplify the rules, however, we do not write the *iknows* facts that already appeared in the LHS. In other words, in our rules *iknows* is *implicitly persistent*, and we interpret the rules as if every LHS *iknows* also appears in the RHS.

Also, a rule can be labeled with a list of existentially quantified variables. Their purpose is to introduce new constants like fresh nonces. Note that in the previous version of the IF we instead used a method of creating unique terms; this is, of course, still possible, but the new specification language does no longer prescribe a particular method to create new constants.

Section Goals (`GoalsSection`). The goals are defined in terms of predicates on states and are conceptually not different from the LHS of rules (and we will define their semantics similarly); consequently, we allow also conditions and negative facts in the goals.

Section Intruder (`IntruderSection`). The rules in this section describe the abilities of the intruder, namely composition and decomposition of messages he knows (according to the standard Dolev-Yao intruder extended with the ability to exploit the specified algebraic properties of operators). As these abilities are again independent of the protocol, they are included in the prelude.