User-Guide
for Algebraic Intruder Deductions in OFMC

Sebastian Mödersheim
Information Security Group, Dep. of Computer Science, ETH Zurich, Switzerland
www.infsec.ethz.ch/~moedersheim

February 13, 2006

OFMC is now enhanced to include the support for user-defined algebraic theories. In a nutshell, this means that both unification of terms and intruder deduction are performed with respect to such a theory. The implementation is based on the framework described in [1], a copy of which is included in this release, giving an introduction and formal definition of the problem of intruder deduction in presence of an algebraic theory. This user-guide describes how to use the new features in OFMC.

Please note that the state of the implementation is still preliminary, and there may be bugs and even cryptic error messages. For all inconveniences we want to apologize in advance.

1 OFMC's Built-in Theory

In this section, we describe the theory and deduction rules used by OFMC when no theory file is specified. Note that all aspects of this theory can be overridden by custom theories. The built-in theory comprises of the following equations.

$$\begin{aligned} exp(exp(X1, X2), X3) &\approx exp(exp(X1, X3), X2) \\ xor(X1, X2) &\approx xor(X2, X1) \\ xor(X1, xor(X2, X3)) &\approx xor(xor(X1, X2), X3) \\ xor(X, X) &\approx e \\ xor(X, e) &\approx X \end{aligned}$$

Notation: we use here the symbol *xor* in ASCII-Notation, as it is read by OFMC (while in [1], we use the notation $a \oplus b$). Further, we use the convention that all variable identifiers like $X1$ start with an upper-case letter, while all other identifiers start with a lower-case letter. The $\approx$ symbol is used to denote equivalence of terms in the algebraic theory, as opposed to $=$ for the syntactic equivalence of terms.

Symbol	Arity	Intuition	Intruder-Accessible
inv	1	private-key of given public-key	no
crypt	2	asymmetric encryption	yes
scrypt	2	symmetric encryption	yes
pair	2	pairing/concatenation	yes
apply	2	function application	yes
exp	2	exponentiation modulo fixed prime p	yes
xor	2	bitwise exclusive or	yes
e	0	bit-string of zeros	yes

Figure 1: Symbols of OFMC's built-in theory

This built-in theory is sufficient to support protocols based on the Diffie-Hellman key exchange, where *exp* stands for the exponentiation modulo a common prime p (which is omitted). Other uses of exponentiation, e.g. in RSA-based encryptions, may require a different kind of theory (including the modulus), which is discussed below.

xor (bitwise exclusive or) is also a commonly used primitive, and we have thus included as part of the built-in theory the basic properties of this operation.

There are several other built-in function symbols, which are interpreted as free symbols (i.e. there are no algebraic properties attached to them). Figure 1 lists all symbols. In this table, *intruder-accessible* means, whether the intruder can build terms with this function symbol himself. For example, when he knows two messages m and k, then he can build the message $crypt(k, m)$, i.e. the encryption of m using k as a key. Also, the built-in theory has explicit decryption rules for the intruder that allow him to obtain a message m, if he knows both $crypt(k, m)$ and $inv(k)$, or if he knows $crypt(inv(k), m)$ and k, or if he knows $scrypt(k, m)$ and k. (Note that the built-in theory does not include the property $inv(inv(k)) = k$.) For *pair*, we also have "decryption" rules namely that from knowing $pair(m1, m2)$ the intruder can derive both $m1$ and $m2$. We will explain below how such decryption rules are specified. Also we will discuss later the meaning of the function application operator *apply* and the issues of typing.

2 Running OFMC with a Custom Theory

We have included in this release an example theory file, `example.thy`, that contains several alternatives and extensions of the standard theory that might be useful in many analyses:

- Explicit decryption. While the built-in theory has special decryption rules (that tell how an intruder can obtain the clear-text of an encryption), one may consider explicit decryption operations. This is essential when there is no means to check the result of the decryption (i.e. when some clear-messages do not contain special tags that could be used for such a check).

An example of such a specification can be found in Appendix A. This is also the basis for considering offline-guessing attacks [2].

- A more general theory of exponentiation, including also the relationship with addition and multiplication.
- A very restricted theory of exponentiation, that is the "minimum" to support Diffie-Hellman.
- Commitment schemes.
- Associativity of concatenation.

To start the protocol analysis with such a theory file simply call

```
ofmc <IF-File> -theory <Theory-File>
```

When the `-theory` option is not given, OFMC uses the built-in theory.

3 Structure of a Theory File

3.1 Disjoint Subtheories

First, a theory file is composed of several so-called disjoint subtheories, i.e. each subtheory is concerned with a set of symbols that do not appear in any other subtheory. For instance, the built-in theory can be split into several disjoint subtheories, as denoted by the horizontal lines in Fig. 1.

Although one is not obliged to partition the theory into subtheories (and rather have just one large 'subtheory'), such a structuring is advisable for the efficiency, whenever possible.

We will in the following use as a running example the subtheory `Xor` from `example.thy`.

3.2 The Signature

Each subtheory begins with a description of the function (and constant) symbols and their arities.

```
Signature:
  e/0,
  xor/2
```

This example declares that e is a constant symbol and xor is a binary function symbol. Currently, there are a few limitations imposed by the implementation:

- Only symbols of arity 0, 1, and 2 are allowed.
- Symbols of arity 0 and 1 may not appear in the finite theory F.

- Symbols of arity 1 are always non-intruder-accessible, while all symbols of arity 2 are intruder-accessible.
- There is a built-in symbol *napair* representing non-associative pairs (i.e. it behaves as in a free algebra) which cannot be overloaded with algebraic properties.

It is not difficult to work around these limitations by either artificially increasing the arity (with dummy-arguments) or decreasing it (using additional function symbols). However, for convenience, we will improve the implementation as soon as possible.

3.3 Topdec

As explained in [1], the approach allows for the specification of algebraic theories E such that $E = F \cup C$ where F is a *finite* theory and C is a *cancellation* theory, and the relation $\rightarrow_{C/F}$ is *convergent*. Cancellation theories and convergence will be considered below in this document, first let us focus on the finite theory F. A theory is called *finite* iff the equivalence class of every term under F is finite. For instance the following two properties of *xor* induce a finite theory, namely that it is AC (associative and commutative):

$$\begin{aligned} xor(X1, X2) &\approx xor(X2, X1) \\ xor(X1, xor(X2, X3)) &\approx xor(xor(X1, X2), X3) \end{aligned}$$

OFMC does not directly allow for the specification of such a finite theory, rather one has to specify a more algorithmic description of the theory. This limitation can probably not be avoided even in future versions due to undecidability of several questions concerning finite theories, e.g. whether a given set of equations induces a finite theory.

What we have to specify for OFMC is a recursive algorithm for *toplevel decomposition*, which is a special case of the unification problem. More precisely, given a non-variable term $t = op(t_1, \ldots, t_n)$ and an operator op' (where op and op' range over all operators of the subtheory being specified), describe the solutions for the unification problem

$$op(t_1, \ldots, t_n) \approx op'(X1, \ldots, Xm)$$

for fresh variables Xi.

Note that in the following, we will also write $T1, \ldots, Tn$ since these terms will be *meta-variables*, i.e. variables of the decomposition algorithm.

For instance, if $T = xor(T1, T2)$ and $op' = xor$, then there are at least two solutions:

$X1$	$X2$
$T1$	$T2$
$T2$	$T1$

There may be more solutions, if $T1$ or $T2$ are themselves terms with *xor* at the top. This is the point where the definition of the toplevel decomposition is recursive: suppose $T1$ is not a variable term, and we already have all the solutions for the problem

$$T1 \approx xor(Z1, Z2) \ .$$

Then, we have the following additional solutions of the toplevel decomposition of T — for every solution of $Z1$ and $Z2$:

$X1$	$X2$
$Z1$	$xor(Z2, T2)$
$xor(Z1, T2)$	$T1$

Note that due to the recursive nature of the definition, all solutions that do not require $T2 = xor(...)$ are also covered, namely those that require $Z1$ or $Z2$ to be xor-terms themselves. Also observe that equivalent solutions modulo F do not need to be spelled out (i.e. writing $xor(Z2, T2)$ or $xor(T2, Z2)$ does not make a difference).

The additional solutions for the case that $T2$ is of the form $xor(Z1, Z2)$ (but $T1$ is not necessarily an xor-term):

$X1$	$X2$
$xor(T1, Z2)$	$Z2$
$Z1$	$xor(T1, Z2)$

Finally, there is one solution not covered by the others in the case that both $T1 = xor(Z1, Z2)$ and $T2 = xor(Z3, Z4)$:

$X1$	$X2$
$xor(Z1, Z3)$	$xor(Z2, Z4)$

Thus we have the following description of the topdec-algorithms for *xor*:

```
Topdec:
 topdec(xor,xor(T1,T2))=
   [T1,T2]
   [T2,T1]
   if T1==xor(Z1,Z2){
     [Z1,xor(Z2,T2)]
     [xor(Z1,T2),Z2]
     if T2==xor(Z3,Z4){
       [xor(Z1,Z3),xor(Z2,Z4)]}}
   if T2==xor(Z1,Z2){
     [xor(T1,Z1),Z2]
     [Z1,xor(T1,Z2)]}
```

Note that the variables in the topdec(...) line and every if statement are *pattern variables*, i.e. representing arbitrary subterms; we currently allow here only *linear patterns*, meaning that for instance a condition like `if T==f(X,X)` is not allowed. Non-linear patterns shall be introduced in a future version of OFMC.

3.4 Cancellation Rules

Next we have the cancellation rules which must have the form $l = r$ such that r is either a constant or a variable that appears in l. Note that the notation here is not commutative, so r must be really specified as the right side. For xor, we have the properties:

```
Cancellation:
  xor(xor(X1,X1),X2) = X2
  xor(X1,X1) = e
  xor(X1,e)  = X1
```

First note that we do not have to specify the equation $xor(e, X1) = X1$, since all these equations are "applied" modulo F, which includes commutativity in the case of xor. In this light, the first equation seems redundant, i.e. it is implied by the other two. This has to do with the notion of convergence that we are using.

In [1], we have used the rewrite relation $\rightarrow_{C/F}$ based on F-equivalence classes of terms. Though theoretically elegant (i.e. we can use the standard notion of convergence), it is practically difficult to work with (e.g. there is no completion method). We have thus also foreseen the possibility to integrate the following variant of the modular rewriting relation: $t \rightarrow_{C,F} s$ iff there is a rule $l \rightarrow r \in C$ and a position p in t and a substitution σ such that $t|p \approx_F l\sigma$, and $s = t[r\sigma]p$. To illustrate the difference between $\rightarrow_{C/F}$ and $\rightarrow_{C,F}$, consider the example

$$xor(a, xor(b, a)) \rightarrow_{C/F} xor(e, b) \ ,$$

while

$$xor(a, xor(b, a)) \not\rightarrow_{C,F} xor(e, b) \ ,$$

if we leave out the first rule of the above equations for xor. The reason is that there is no subterm of

$$xor(a, xor(b, a))$$

that is F-equivalent to $xor(a, a)$. Adding this first above rule, however, the above reduction is possible for $\rightarrow_{C,F}$.

The notion of confluence of $\rightarrow_{C,F}$ is adapted as expected: for any $t_1 \approx_{C \cup F} t_2$, there exist s_1 and s_2 such that $t_1 \rightarrow^*_{C,F} s_1$ and $t_2 \rightarrow^*_{C,F} s_2$ and $s_1 \approx_F s_2$. Convergence is then confluence and termination. Due to their form, cancellation theories are always terminating.

3.5 Analysis

The analysis section describes what an intruder can derive from messages he knows (except for the trivial fact that he can always compose more complex messages; [1] gives a formal definition of a completely analyzed knowledge). For instance, if the intruder knows $xor(X1, X2)$, then we should check whether he can generate $X1$. If so, he can derive $X2$:

```
Analysis:
  decana(xor(X1,X2))=
    [X1]->[X2]
    [xor(X1,X3)]->[xor(X2,X3)]
```

The last line adds the case that the intruder knows $xor(X1, X3)$, i.e. he knows something that contains $X1$; then he can "*xor*-out" the $X1$ and obtain $xor(X2, X3)$.

Some remarks are in order:

- There are squared brackets around the terms since in some cases there are several terms on the antecedent and the consequent side.
- These rules are redundant with the cancellation theory, however, OFMC requires to specify the analysis rules (again for reasons of computability).
- Using such analysis rules, we can also specify additional derivation steps of the intruder that are not implied by the equational theory. Note however that OFMC may not terminate when rules are given that allow infinitely long sequences of analysis steps; an example would be the following "analysis" rule:

  ```
  decana(X) = [Y] -> [xor(X,Y)]
  ```

  which actually describes a way how the intruder can build more and more complex terms, and there is no finite fixed-point in that process.
- In the above example, one may wonder, why there is not the additional rule

  ```
  [X2]->[X1]
  ```

  The reason is that unification between terms in the intruder knowledge and the decana-rules is performed modulo F (which implies in this case that *xor* is commutative).
- Last but not least, one may wonder why we split assumptions into two parts, rather than just specifying something like

  ```
  decana(xor(X1,X2),X1) -> X2
  ```

  The reason for the actual specification is the way these analysis-rules are interpreted: every message in the intruder knowledge is checked for unification (modulo F) with the term given as the decana argument. For every unifier, we check for the derivability (modulo F) of the additional assumptions (in brackets left of the arrow) under the unifier. If that check is positive (under a certain substitution), then we add to the intruder knowledge the list of terms right of the bracket (under the unifier and the substitutions of the derivation).

4 Dealing with the Complexity

The algebraic properties may blow up the search tree enormously. Here are few tips for coping with the extra burden:

- Sometimes an explosion is generated by the rule normalization step, which is performed before the analysis. You can suppress this normalization step by the `-nonorm` option or alternatively let OFMC display the normalized rules with `-showrules`.
- Typing can help: especially when normalization seems to be the trouble, it can be helpful to limit things by giving types to constants and variables involved. Note that for variables you should only give a type when the variable stands for an atomic message in the legal protocol execution (thus only type-flaws are potentially excluded). In the worst case, try without certain cancellation rules and just describing analysis rules for the intruder (i.e. no explicit decryption).
- Often one can get an overview of the complexity of the search space by limiting (and iteratively increasing) the search depth in the tree, using the option `-d <depth>`. Also, manually browsing the search tree with the `-p <path>` option often gives some insight.

References

1. D. Basin, S. Mödersheim, and L. Viganò. Algebraic intruder deductions. In G. Sutcliffe and A. Voronkov, editors, *LPAR 2005*, volume 3835 of *LNAI*, pages 549–564. Springer-Verlag, December 2005.
2. P. Hankes Drielsma, S. Mödersheim, and L. Viganò. A formalization of off-line guessing for security protocol analysis. In F. Baader and A. Voronkov, editors, *LPAR 2004*, volume 3452 of *LNAI*, pages 363–379. Springer, March 2005.
3. T. Wu. The Secure Remote Password Protocol. In *Proc. of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.

A The SRP Protocol

The SRP protocol (Secure Remote Passwords, [3]) is a challenging example for algebraic properties, since it requires a full arithmetic theory to work. It uses modular addition, multiplication and exponentiation, and without the necessary properties it is not executable. In the EU project AVISPA, as part of which OFMC and several other tools have been developed, this protocol was modeled in a drastically simplified version, basically reducing it to a Diffie-Hellman key-exchange.

A.1 An Arithmetic Theory

With the new theory features of OFMC, it is now possible to model the protocol in a much more realistic way. In particular, we can model the relationship between addition, multiplication, and exponentiation; all that is required is a theory file with the necessary properties. We consider the following properties, although it should be noted that, depending on the formulation of the transitions of the honest agents, not all of them are necessary:

$$\begin{array}{rcl} A+B & = & B+A \\ A+(B+C) & = & (A+B)+C \\ A\cdot B & = & B\cdot A \\ A\cdot(B\cdot C) & = & (A\cdot B)\cdot C \\ A\cdot(B+C) & = & (A\cdot B)+(A\cdot C) \\ exp(exp(A,B),C) & = & exp(A,B\cdot C) \\ exp(A,B+C) & = & exp(A,B)\cdot exp(A,C) \end{array} \qquad \begin{array}{rcl} A+(-A) & = & 0 \\ A+0 & = & A \\ A\cdot(A^{-1}) & = & 1 \\ A\cdot 1 & = & A \end{array}$$

Note that we cannot model distributivity completely. First, it easily destroys confluence of $\rightarrow_{F/C}$. Second, even when ignoring possible incompleteness due to non-confluence, we have in topdec the requirement that the patterns written in the 'if'-statements must be linear, e.g. one cannot specify `if T==f(X,X)` etc. This would be required to specify how 'dividing out' works. This restriction is due to the current implementation. For now, we thus specify only 'one direction' of distributivity. However the formulation we will choose does not require distributivity to work.

Here is the respective subtheory (part of the file `arithmetic.thy`:

```
Theory Arithmetic:
  Signature:
    add/2, neg/1, zero/0,
    mult/2, minv/1, one/0,
    exp/2
  Cancellation:
    add(X,neg(X))=zero
    add(X,zero)=X
    add(add(X,Y),neg(Y))=X
    mult(X,minv(X))=one
```

```
  mult(X,one)=X
  mult(mult(X,Y),minv(Y))=X
Topdec:
  % add is associative and commutative:
  topdec(add,add(T1,T2))=
    [T1,T2]
    [T2,T1]
    if T1==add(Z1,Z2){
      [Z1,add(Z2,T2)]
      [add(Z1,T2),Z2]
      if T2==add(Z3,Z4){
        [add(Z1,Z3),add(Z2,Z4)]}}
    if T2==add(Z1,Z2){
      [add(T1,Z1),Z2]
      [Z1,add(T1,Z2)]}
  %
  % mult is associative and commutative:
  topdec(mult,mult(T1,T2))=
    [T1,T2]
    [T2,T1]
    if T1==mult(Z1,Z2){
      [Z1,mult(Z2,T2)]
      [mult(Z1,T2),Z2]
      if T2==mult(Z3,Z4){
        [mult(Z1,Z3),mult(Z2,Z4)]}}
    if T2==mult(Z1,Z2){
      [mult(T1,Z1),Z2]
      [Z1,mult(T1,Z2)]}
  %
  % Distributivity: mult(X1,add(X2,X3))=add(mult(X1,X2),mult(X1,X3))
  topdec(add,mult(X1,X2))=
    if X2==add(X3,X4){
      [mult(X1,X3),mult(X1,X4)]}
  % The ‘‘other direction’’ we currently cannot model, here is how
  % it shall look like in the future:
  %    topdec(mult,add(X1,X2))=
  %      if X1==mult(X3,X4){
  %        if X2==mult(X3,X5){
  %          [X3,mult(X4,X5)]}}
  %
  % Relation between exp,mult and add:
  % exp(exp(X1,X2),X3)=exp(X1,mult(X2,X3))
  % exp(X1,sum(X2,X3))=mult(exp(X1,X2),exp(X1,X3))
  topdec(exp,exp(T1,T2))=
    [T1,T2]
    if T1==exp(Z1,Z2){
```

```
      [Z1,mult(T2,Z2)]
      [exp(Z1,T2),Z2]}
     if T2==mult(Z1,Z2){
      [exp(T1,Z1),Z2]}
  topdec(mult,exp(T1,T2))=
    if T2==sum(Z1,Z2){
    [exp(T1,Z2),exp(T1,Z2)]}
Analysis:
  decana(add(X1,X2)) =[X1]->[X2]
  decana(mult(X1,X2))=[X1]->[X2]
  decana(exp(X1,X2)) =[X2]->[X1]
  decana(neg(X))=[]->[X]
  decana(minv(X))=[]->[X]
```

Note that with such a theory, several larger protocols will just explode, so only use this theory when you really want to go deep into arithmetic!

A.2 The Protocol Formalization

An important aspect of the protocol that we currently cannot model is the fact that the shared passwords of Users and Hosts, denoted `passwd(User,Host)`, may be weak (guessable). Though foundational research in this direction has been done, for instance [2], this is not yet implemented: it requires algebraic properties in the first place, which had to be done first as of this release of OFMC.

The Host additionally has, in his password file, a random value for each User, called the *salt*, denoted `salt(User,Host)`. This value is sent in clear text during the authentication process. User and Host build a hash value, called the *verifier* from salt and password, namely `x=h(salt,h(User,passwd))`, which is then linked with a Diffie-Hellman key-exchange to provide authentication based on the password without opening the door for guessing attacks.

In the following description, we may omit the parameters User and Host of the password and salt table, when there is no danger of confusion. Moreover, User and Host already agree on a Diffie-Hellman group `g`, and a modulus `n`. Note that all additions, multiplications and exponentiations are modulo `n`.

Here is the protocol in Alice&Bob notation, with the same identifiers as in the original RFC, but some terms rewritten according to the algebraic theory:

$$
\begin{array}{lcll}
User & \rightarrow & Host: & User, g^a \quad \text{— for random value a} \\
Host & \rightarrow & User: & salt, g^x + g^b \quad \text{— for random value b} \\
 & & K := h(g^{a \cdot b} * g^{b \cdot x}) & \\
 & & M := h(salt, g^a, g^x + g^b, K) & \\
User & \rightarrow & Host: & M \\
Host & \rightarrow & User: & h(g^a, M, K)
\end{array}
$$

Note that the original protocol contains also a value, called u in the RFC, which is the first 32 bits of the half-key g^b. As this value appears only in

messages that contain g^b anyway, it does not make a difference whether this value is omitted from the point of view of our formal model.

The IF file is part of this release. Here we give only one transition rule as an example, which describes how the User receives message 2 from the Host and forms message 3 (the dotted part in the real rule just contains dummy messages):

```
step step_1 :=
 state_srp_user(User,Host,1,Password,...,SET,SID).
 iknows(pair(Salt,GXplusGB))
 & X  = h(pair(Salt,h(pair(User,Password))))
 & GB = add(GXplusGB,neg(exp(g,X)))
 & K  = h(mult(exp(GB,A),exp(GB,X)))
 & M  = h(pair(Salt,pair(exp(g,A),pair(add(exp(g,X),GB),K))))
=>
 state_srp_user(User,Host,2,Password,X,SALT,A,GB,K,M,SET,SID).
 iknows(M).
 witness(User,Host,key,K).
 secret(K,key,Set).contains(User,Set).contains(Host,Set)
```

We have some syntactic sugar here of the form $Var = Term$ which allows us to form complex messages without copy-pasting and consequently loosing the overview. The semantics of this new sugar, which is not (yet) part of the official IF standard as of AVISPA deliverable, is simply a uniform replacement (of all occurrences of the variable with the term) over the entire transition rule.

Observe how receiving and "parsing" of the incoming message work: according to the protocol, the incoming message should contain the `Salt` as the first component and the sum of the verifier `exp(g,x)` and the Hosts new half-key `exp(g,b)` as the second component. However, the receiver cannot tell how this second component was obtained, i.e. whether it is really a sum of two terms. In fact, the idea behind the sum is that nobody can tell how it decomposes into two summands, unless one knows one of the summands.

Therefore the rule can be understood as follows: the User first receives some value `GXplusGB` (which is *supposed* to be a sum, but he cannot check that). He builds the verifier `exp(g,X)` which depends on the salt and his password, and then subtracts this verifier from the value `GXplusGB`. (This subtraction is in fact done in our model by adding the additive inverse of the verifier.) According to the algebraic theory, the result, called `GB`, reduces to a simpler term iff `GXplusGB` indeed contains the verifier as a summand, otherwise term may be irreducible, representing that the User indeed holds some random gibberish in his hands now. It is important to note that the User cannot distinguish whether what he received is meaningful—and this in particular crucial when considering guessing attacks—he will simply carry on with whatever he has now as `GB`, and construct the key `K` and the authentication message `M`.

The other rules can be understood similarly.