Maude-NPA: Tutorial

Catherine Meadows, Naval Research Laboratory (USA)
José Meseguer, University of Illinois at Urbana-Champaign (USA)
Santiago Escobar, Universidad Politécnica de Valencia (Spain)

PROTOCOL EXCHANGE, JANUARY 23, 2008

Goal

- Crypto protocol analysis with the standard free algebra model (Dolev-Yao) well understood.
- Extend standard free algebra model of crypto protocol analysis to deal with algebraic properties
  1. Encryption-decryption,
  2. Diffie Hellman,
  3. Exclusive-or, etc.
- Provide tool that can be used to reason about protocols with these algebraic properties in the unbounded session model

Our approach

- Use rewriting logic as general theoretical framework
  - crypto protocols are specified as rewrite rules
  - algebraic identities as equational properties
- Use narrowing modulo equational theories as a symbolic reachability analysis method
- Combine with state reduction techniques of NPA (grammars, optimizations, etc.)
- Implement in Maude programming environment
  - Rewriting logic gives us theoretical framework and understanding
  - Maude implementation gives us tool support

Maude-NPA

- A tool to find or prove the absence of attacks using backwards search
- Analyzes infinite state systems
  - Active intruder
  - No abstraction or approximation of nonces
  - Unbounded number of sessions
- Intruder and honest protocol transitions represented using strand space model.
- Different algebraic theories included
- Uses induction techniques defined in terms of formal languages to cut down search space
- Uses optimization techniques to improve performance: only input messages, partial order, information from strand space model,lazy intruder, etc.

A Little Background on Unification

- Given a signature Σ and an equational theory E, and two terms s and t built from Σ:
- A unifier of s and t is a substitution σ to the variables in s and t such that σs can be transformed into σt by applying equations from E to s and t and their subterms
- Example: $\Sigma = \{d/2, e/2, m/0, k/0\}, E = \{d(K, e(K, X) = X\}$. The substitution $\sigma = \{X/e(K, Y)\}$ is a unifier of $d(K, X)$ and Y.
- The set of most general unifiers of s and t is the set Γ such that any unifier σ is of the form $\rho\tau$ for some ρ, and some τ in Γ.
- Example, $\{X/e(K, Y), Y/d(K, X)\}$ is the set of mgu's of $e(K, X)$ and Y.
- Given the theory, can have:
  - at most one mgu (empty theory)
  - a finite number (AC)
  - an infinite number (associativity)
- Problem in general undecidable, so different algorithms devised for different theories

Narrowing

Let σ be a substitution, R a set of rewrite rules and E an equational thoery

Narrowing: $t \leadsto_{\sigma,R,E} s$ if there is

- a non-variable position $p \in Pos(t)$;
- a rule $l \to r \in R$;
- a unifier σ (modulo E) such that $\sigma(t|_p) =_E \sigma(l)$, and $s = \sigma(t[r]_p)$.

Example:

- $R = \{\ X \to d(k, X)\ \}$
- $E = \{\ d(K, e(K, Y)) = Y\ \}$
- $e(k,\ t) \quad \leadsto_{\emptyset,R,E} \quad d(k, e(k, t)) =_E t$

E-Unification and Narrowing

- Maude-NPA based on unification modulo equational theory defining the behavior of different operations used
- Two possible approaches:
  1. Built-in unification algorithms for each theory and combination of theories.
  2. Hybrid approach with Δ and B
     - B is built-in unification algorithm
     - Δ confluent and terminating rules modulo B
       * Confluent: Always reach same normal form, no matter in which order you apply rewrite rules
       * Terminating: Sequence of rewrite rules is finite
     - Implement unification via narrowing with Δ modulo B.
     - More readily extensible to different theories.
- Our Approach
  - Let B be the empty theory or AC
  - Old and new approaches
    * Old: Unification modulo B performed via calls to CiME unification tool
    * New: Unification module B provided by Maude
  - In both cases, narrowing with Δ performed at Maude meta-level

Getting Started

- You should have:
  - Maude alpha89i installed
  - Directory in which it is installed in your path
  - Four different executables: Darwin, intelDarwin, linux, linux64
  - Maude-NPA alpha0.1 directory on your machine
- cd to Maude-NPA directory and start maude
- type `load maude-npa`
- cd to examples directory and type `load nspk`
- to see a grammar generated, type `red genGrammars .`
- to see a goal specified in the nspk file, type `red run(0,0) .`
- to see what the first search step looks like, type `red run(0,1)`

Sorts

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  --- Importing sorts Msg, Fresh, Public, and GhostData
  protecting DEFINITION-PROTOCOL-RULES .

  ----------------------------------------------------------
  --- Overwrite this module with the syntax of your protocol
  --- Notes:
  --- * Sort Msg and Fresh are special and imported
  --- * Every sort must be a subsort of Msg
  --- * No sort can be a supersort of Msg
  ----------------------------------------------------------
  --- Sort Information
  sorts Name Nonce Key Enc .
  subsort Name Nonce Enc Key < Msg .
  subsort Name < Key .
  subsort Name < Public .
```

- Public types must be declared public in two places, sorts and intruder strands
- Plan to simplify this in later releases

Operations

```
  --- Encoding operators for public/private encryption
  op pk : Key Msg -> Enc [frozen] .
  op sk : Key Msg -> Enc [frozen] .

  --- Nonce operator
  op n : Name Fresh -> Nonce [frozen] .

  --- Principals
  op a : -> Name . --- Alice
  op b : -> Name . --- Bob
  op i : -> Name . --- Intruder

 --- Concatenation operator
  op _;_ : Msg  Msg  -> Msg [gather (e E) frozen] .

endfm
```

Algebraic Theory

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .


  var Z : Msg .
  var Ke : Key .

  *** Encryption/Decryption Cancellation
  eq pk(Ke,sk(Ke,Z)) = Z [nonexec] .
  eq sk(Ke,pk(Ke,Z)) = Z [nonexec] .

endfm
```

Intruder Strands

```
fmod USER-INPUT is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .
  protecting DEFINITION-PROTOCOL-RULES .
  protecting DEFINITION-CONSTRAINTS-INPUT .

var Ke : Key .
  vars X Y Z : Msg .
  vars r r' : Fresh .
  vars A B : Name .
  vars N N1 N2 : Nonce .

  eq STRANDS-DOLEVYAO
   = :: nil :: [ nil | -(X), -(Y), +(X ; Y), nil ] &
     :: nil :: [ nil | -(X ; Y), +(X), nil ] &
     :: nil :: [ nil | -(X ; Y), +(Y), nil ] &
     :: nil :: [ nil | -(X), +(sk(i,X)), nil ] &
     :: nil :: [ nil | -(X), +(pk(Ke,X)), nil ] &
     :: nil :: [ nil | +(A), nil ]
  [nonexec] .
```

Do's and Don'ts of intruder strands

- WARNING! Do not leave in an intruder strand you don't need! It will only slow the tool down.
- DO include an intruder strand for each operation specified and used in the protocol.
- If an operation has more than one output (as in deconcatenation), an intruder strand must be created for each output.

Protocol Strands

```
eq STRANDS-PROTOCOL
 = :: r ::
   [ nil | +(pk(B,A ; n(A,r))), -(pk(A,n(A,r) ; N)), +(pk(B, N)), nil ] &
   :: r ::
   [ nil | -(pk(B,A ; N)), +(pk(A, N ; n(B,r))), -(pk(B,n(B,r))), nil ]
[nonexec] .
```

- Bar divides strand into past and future, always at beginning in specification
- Each strand indexed by fresh variables, `r` in this case, `nil` (for no fresh variables in the intruder strands

Attack States

- Attack states give us the goals, and also allow us to guide the search
- Here, a completes the protocol (thinking it is with b), but the intruder learns n(b,r)

```
eq ATTACK-STATE(0)
   = :: r ::
     [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
     || n(b,r) inI, empty
     || nil
     || nil
  [nonexec] .
```

Initial Attack State

```
result System: (
:: nil :: [nil | -(pk(i, n(b, #1:Fresh))), +(n(b, #1:Fresh)), nil] &
:: nil :: [nil | -(pk(i, a ; n(a, #0:Fresh))), +(a ; n(a, #0:Fresh)), nil] &
:: nil :: [nil | -(n(b, #1:Fresh)), +(pk(b, n(b, #1:Fresh))), nil] &
:: nil :: [nil | -(a ; n(a, #0:Fresh)), +(pk(b, a ; n(a, #0:Fresh))), nil] &
:: #1:Fresh :: [nil | -(pk(b, a ; n(a, #0:Fresh))), +(pk(a, n(a, #0:Fresh) ; n( b, #1:Fresh))),
   -(pk(b, n(b, #1:Fresh))), nil] &
:: #0:Fresh :: [nil | +(pk(i, a ; n(a, #0:Fresh))), -(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))),
   +(pk(i, n(b, #1:Fresh))), nil])
||
pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh)) !inI, pk(b, n(b, #1:Fresh)) !inI, pk(b,
    a ; n(a, #0:Fresh)) !inI, pk(i, n(b, #1:Fresh)) !inI, pk(i, a ; n(a,
    #0:Fresh)) !inI, n(b, #1:Fresh) !inI, (a ; n(a, #0:Fresh)) !inI
||
+(pk(i, a ; n(a, #0:Fresh))), -(pk(i, a ; n(a, #0:Fresh))), +(a ; n(a,
    #0:Fresh)), -(a ; n(a, #0:Fresh)), +(pk(b, a ; n(a, #0:Fresh))), -(pk(b, a
    ; n(a, #0:Fresh))), +(pk(a, n(a, #0:Fresh) ; n(b, #1:Fresh))), -(pk(a, n(a,
    #0:Fresh) ; n(b, #1:Fresh))), +(pk(i, n(b, #1:Fresh))), -(pk(i, n(b,
    #1:Fresh))), +(n(b, #1:Fresh)), -(n(b, #1:Fresh)), +(pk(b, n(b,
    #1:Fresh))), -(pk(b, n(b, #1:Fresh)))
||
nil
```

Another Way of Specifying Goal State

- Next, we specify the attack state by saying that a completes, but b does not

```
eq ATTACK-STATE(1)
= :: r ::
   [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
   || empty
   || nil
   || nil
   butNeverFoundAny
   :: r' ::  [nil | +(pk(b,a ; N)), -(pk(a, N ; n(b,r))), +(pk(b,n(b,r))), nil ]
             & S:StrandSet
   || K:IntruderKnowledge
   || M:SMsgList
   || G:GhostList
 [nonexec] .
```

Using Attack States to Prune Search

- If we keep on looking for the attack state, we find out the search does not terminate
- If we inspect the output, we see that the strand `[nil | -(N1 ; N2), +(pk(B, N1 ; N2)), nil]` keeps appearing in an infinte loop
- Putting it the the `butNeverFoundAny` section eliminates the infinite loop
- Note: soundness is not guaranteed
- Next version of MaudeNPA should reduce need for this without compromising soundness

```
 eq ATTACK-STATE(1)
 = :: r ::
   [ nil, -(pk(b,a ; N)), +(pk(a, N ; n(b,r))), -(pk(b,n(b,r))) | nil ]
   || empty
   || nil
   || nil
   butNeverFoundAny
   :: r' ::  [nil | +(pk(b,a ; N)), -(pk(a, N ; n(b,r))), +(pk(b,n(b,r))), nil ]
            & :: nil ::  [nil | -(N1 ; N2), +(pk(B, N1 ; N2)), nil] & S:StrandSet
   || K:IntruderKnowledge || M:SMsgList  || G:GhostList
[nonexec] .
```

Exercises

- Try running the NSPK protocol (using the red run command) until you get an initial state (14 iterations). Then do red initials to see what the attack looks like
- Try running the protocol with and without the butNeverFoundAny clause.
- Try replacing `-(N1 ; N2), +(pk(B, N1 ; N2))` with -(X ; Y), +(pk(B, X; Y)) in the butNeverFoundAny clause, and see what happens.
- Try specifying Lowe-NSPK and see what happens. Does the second butNeverFoundAny clause still help? How would you change it so it does?

An AC Protocol: Diffie-Hellman

```
A --> B: A ; B ; exp(g,N_A)
B --> A: A ; B ; exp(g,N_A)
A --> B: enc(exp(exp(g,N_B),N_A),secret(A,B))
```

Properties of Interest

- `e(K,d(K,M) -> M d(K,e(K,M) => M`
- `exp(exp (Z, X1), X2) exp(Z,X1 <+> X2)`
- `<+>` is AC

The AC properties will be handled differently from the rest.

Sort Information

```
fmod PROTOCOL-EXAMPLE-SYMBOLS is
  --- Importing sorts Msg, Fresh, Public
  protecting DEFINITION-PROTOCOL-RULES .


  --- Sort Information
  sorts Name Nonce NeNonceSet Gen Exp Key GenvExp Enc Secret .
  subsort Gen Exp < GenvExp .
  subsort Name NeNonceSet GenvExp Enc Secret Key < Msg .
  subsort Exp < Key .
  subsort Name < Public . --- This is quite relevant and necessary
  subsort Gen < Public . --- This is quite relevant and necessary
```

Operations

```
--- Secret
  op sec : Name Fresh -> Secret [frozen] .
  --- Nonce operator
  op n : Name Fresh -> Nonce [frozen] .
--- Intruder
  ops a b i : -> Name .
  --- Encryption
  op e : Key Msg -> Enc [frozen] .
  op d : Key Msg -> Enc [frozen] .
  --- Exp
  op exp : GenvExp NeNonceSet -> Exp [frozen] .
--- Gen
  op g : -> Gen .
  --- NeNonceSet
  subsort Nonce < NeNonceSet .
  op _<+>_ : NeNonceSet NeNonceSet -> NeNonceSet [frozen assoc comm] .
  --- Concatenation
  op _;_ : Msg Msg -> Msg [frozen gather (e E)] .

endfm
```

Algebraic Properties (Not Counting AC)

```
fmod PROTOCOL-EXAMPLE-ALGEBRAIC is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .

  ----------------------------------------------------------
  --- Overwrite this module with the algebraic properties
  --- of your protocol
  ----------------------------------------------------------

  eq exp(exp(W:Gen,Y:NeNonceSet),Z:NeNonceSet)
   = exp(W:Gen, Y:NeNonceSet <+> Z:NeNonceSet) .
  eq e(K:Key,d(K:Key,M:Msg)) = M:Msg .
  eq d(K:Key,e(K:Key,M:Msg)) = M:Msg .

endfm
```

- Note: AC is defined in the operations section.

Variable Declarations

```
fmod PROTOCOL-SPECIFICATION is
  protecting PROTOCOL-EXAMPLE-SYMBOLS .
  protecting DEFINITION-PROTOCOL-RULES .
  protecting DEFINITION-CONSTRAINTS-INPUT .

  ----------------------------------------------------------
  --- Overwrite this module with the strands
  --- of your protocol
  ----------------------------------------------------------

  vars NS1 NS2 NS3 NS : NeNonceSet .
  var NA NB N : Nonce .
  var GE : GenvExp .
  var G : Gen .
  vars A B : Name .
  vars r r' r1 r2 r3 : Fresh .
  var Ke : Key .
  vars XE YE : Exp .
  vars M M1 M2 : Msg .
  var Sr : Secret .
```

Intruder Strands

```
eq STRANDS-DOLEVYAO =
   :: nil :: [ nil | -(M1 ; M2), +(M1), nil ] &
   :: nil :: [ nil | -(M1 ; M2), +(M2), nil ] &
   :: nil :: [ nil | -(M1), -(M2), +(M1 ; M2), nil ] &
   :: nil :: [ nil | -(Ke), -(M), +(e(Ke,M)), nil ] &
   :: nil :: [ nil | -(Ke), -(M), +(d(Ke,M)), nil ] &
   :: nil :: [ nil | -(NS1), -(NS2), +(NS1 <+> NS2), nil ] &
   :: nil :: [ nil | -(GE), -(NS), +(exp(GE,NS)), nil ] &
   :: r ::   [ nil | +(n(i,r)), nil ] &
   :: nil :: [ nil | +(g), nil ] &
   :: nil :: [ nil | +(A), nil ]
[nonexec] .
```

- By the way we using typing on the inputs to `exp`, we control the size of the search space

Protocol Strands

```
eq STRANDS-PROTOCOL =
     :: r,r' ::
     [nil | +(A ; B ; exp(g,n(A,r))),
            -(A ; B ; XE),
            +(e(exp(XE,n(A,r)),sec(A,r'))), nil] &
     :: r ::
     [nil | -(A ; B ; XE),
            +(A ; B ; exp(g,n(B,r))),
            -(e(exp(XE,n(B,r)),Sr)), nil]
  [nonexec] .
```

Extra Grammars

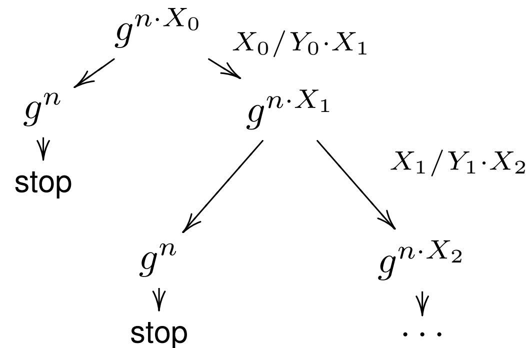
- We are still experimenting with the best initial grammars for AC grammars, so we can't generate them automatically
- We specify initial grammars for `<+>` in a section called `EXTRA-GRAMMARS`
- Maude-NPA will generate grammar from this initial grammar as well as from those it generates automatically
- Note that initial grammar does not need to be a seed term. It can be any legal grammar.

```
eq EXTRA-GRAMMARS
   = (grl empty => (NS <+> n(a,r)) inL . ;
      grl empty => n(a,r) inL . ;
      grl empty => (NS <+> n(b,r)) inL . ;
      grl empty => n(b,r) inL .
      ! S2 )
  [nonexec] .
```

Attack State

```
eq ATTACK-STATE(0)
  = :: r ::
    [nil, -(a ; b ; XE),
          +(a ; b ; exp(g,n(b,r))),
          -(e(exp(XE,n(b,r)),sec(a,r'))) | nil]
    || empty
    || nil
    || nil
    butNeverFoundAny
    *** Pattern for authentication
    (:: R:FreshSet ::
    [nil | +(a ; b ; XE),
           -(a ; b ; exp(g,n(b,r))),
           +(e(YE,sec(a,r'))), nil]
     & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
```

Infinite Behavior Not Captured by Grammars



- Different from rewrite-rule based grammar behavior, because infinite behavior results from substitution
- Root term grows larger instead of leaf terms

NeverFoundAny Clause Ruiing Out that Infinite Behavior

```
*** Pattern to avoid infinite search space
         (:: nil ::
     [ nil | -(exp(GE,NS1 <+> NS2)), -(NS3), +(exp(GE,NS1 <+> NS2 <+> NS3)), nil ]
     & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
```

- Again, soundness no longer guaranteed
- Working to remove necessity of this by use of comparison with previous states (folding)

Some Cats and Dogs

- There are a few states that Maude-NPA ultimately proves unreachable, but keep cropping up again and again
- We put them in butNeverFoundAny to reduce state explosion

```
*** Pattern to avoid unreachable states
  (:: nil ::
   [nil | -(exp(#1:Exp, N1:Nonce)),
          -(sec(A:Name, #2:Fresh)),
          +(e(exp(#1:Exp, N2:Nonce), sec(A:Name, #2:Fresh))), nil]
   & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
  *** Pattern to avoid unreachable states
  (:: nil ::
   [nil | -(exp(#1:Exp, N1:Nonce)), -(e(exp(#1:Exp, N1:Nonce), S:Secret)),
          +(S:Secret), nil]
   & S:StrandSet || K:IntruderKnowledge || M:SMsgList || G:GhostList)
  *** Pattern to avoid unreachable states
  (S:StrandSet
   || (#4:Gen != #0:Gen), K:IntruderKnowledge || M:SMsgList || G:GhostList)
```

More Exercises

- Try searching on the DH protocol until you get an intial state. (Note: this takes a while, about iterations
- Try it without the state-space controlling butNeverFoundAny clauses. What happens?

Protocol Specification Exercise (1)

- Specify and query the following protocol, asking if B can complete and execution without a corresponding execution by A and vice versa (two different attack states)

  1. $A \to B : N_A$
  2. $B \to A : N_B, sk(B, N_A; N_B)$
  3. $A \to B : sk(A, N_B; N_A)$

- Do not give sk any algebraic properties

Protocol Specification Exercise (2)

- Specify and query the following protocol as in (1)
  1. $A \to B : exp(g, N_A)$
  2. $B \to A : exp(g, N_B), sk(B, exp(g, N_A); exp(g, N_B))$
  3. $A \to B : sk(A, exp(g, N_B); exp(g, N_A))$
- Use the algebraic properties of exponentiation we used in the DH protocol specification.

Protocol Specification Exercise (2)

- Specify and query the following protocol, as in (1)
  1. $A \to B : exp(g, N_A)$
  2. $B \to A : exp(g, N_B); sk(B, exp(g, N_A); exp(g, N_B))$
  3. $A \to B : sk(A, exp(g, N_B); exp(g, N_A))$
- Use the algebraic properties of exponentiation we used in the DH protocol specification.

Protocol Specification Exercise (3): Station to Station Protocol

- Specify and query the following protocol, asking if B can complete and execution without a corresponding execution by A and vice versa
  1. $A \to B : exp(g, N_A)$
  2. $B \to A : exp(g, N_B); e(exp(N_B <+> N_A), sk(B, exp(g, N_A); exp(g, N_B)))$
  3. $A \to B : e(exp(N_A <+> N_B)sk(A, exp(g, N_B); exp(g, N_A)))$
- Use the algebraic properties of exponentiation we used in the DH protocol specification.

Protocol Specification Exercise(4): STS Busted

- Specify the previous three protocols, but with B appending his name to the first message, and A appending her name to the third message
- Try querying them again. What happens now?