TOOLIP Semantics
&
TOOLIP - MaudeNPA

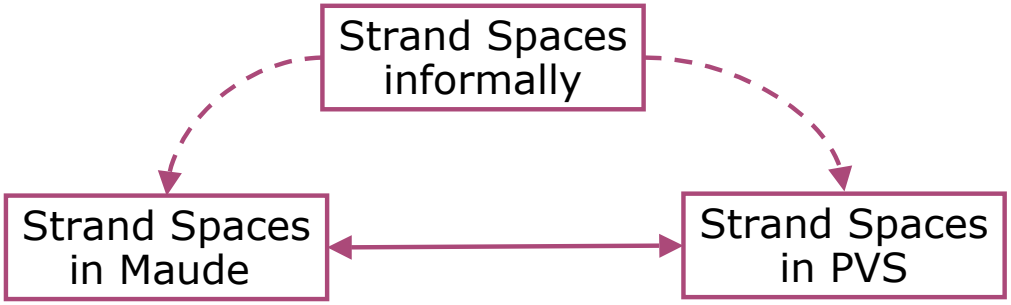
Carolyn Talcott
SRI International
Protocol Exchange, October 2007

Plan

- Vision recalled
- Toolip Grammar
- Maude formalization
  - Syntax
  - Semantics
- Examples
- Mapping Toolip to Maude NPA -- by hand

Original Vision
(due to Sylvan Pinsky)

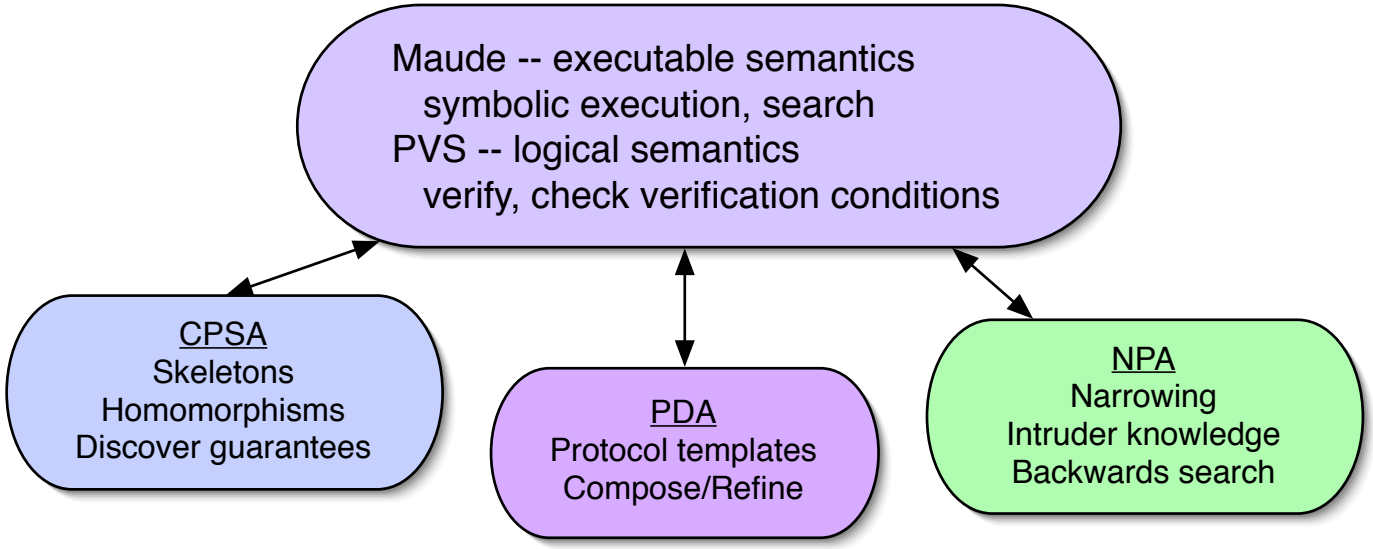
- A foundation for developing cryptographic protocol analysis algorithms
  - Specify and prototype in Maude
  - Verify in PVS



- Formal representations of strand spaces and strand space protocols in Maude and PVS
- Meaning preserving mappings between them

Emerging Vision
(from Protocol eXchange)

Formal framework for semantically sound interoperation of tools for design and analysis of cryptographic protocols (building on the strand space model)



Toolip Grammar
What are we formalizing?

Protocols and Roles

- <Protocol> :=

```
(protocol
  <Version>                    // (currently "0.1")
  <IdentDef>                    //  protocol name.
  (parameters <VariableDef>*)
  (roles  <role>+) )
```

- <role> :=

```
(role
  <IdentDef>                          // Role name
  <PrincipalVar>                 // active principal
  (inputs <VariableDef>*)
  (outputs <Exp>*)                      // [CPPL]
  <IndependentVariables>                // [Joshua]
  (localconstraints <Local-Constraint>*)
  <Code> )                      // protocol actions
```

Code and Exp

- <Code> :=

  (action <Ident> <Kind>)
  |(new <AtomExp>+)
  |(match <Exp> <Pattern> <Ident>)
  |(send <SenderRef> <ReceiverRef> <Exp> <Ident>)
  |(recv <SenderDef> <ReceiverDef> <Exp> <Ident>)
  |(seq <Code>*) | (par <Code>*)

  ...

- <Exp>:= |<AtomExp>|<TupExp>|<AppExp>|<TermExp>
- <TupExp> := (tuple <Exp>*)
- <AppExp> := (app <Fun> <Exp>+)
- <Fun> :=

  (fun(params<Exp>*)<Arity><OutType><Kind><Ident>)

AtomExp

- <AtomExp> :=

```
(principal [(controls <Principal>+)]<Kind> <Ident>)
      | (nonce <Kind> <Ident>)
      | (stkey <Kind> <Ident>)
      | (text <Kind> <Ident>)
      | (ltkey <Principal>* <Kind> <Ident>)
      | (pubkey [<Principal>] <Kind> <Ident>)
      | (privkey [<Principal>] <Kind> <Ident>)
```

Toolip Semantics in Maude Overview

Syntax Modules

- Captures the structure defined by the grammar
  - EXP --- <Exp>
    - fc("E",1,encr)[pubk("0",pc("A"))](nc("A"))
  - CODE --- <Code>
    - seq(new(nv("A")),...)
  - CONSTRAINTS / ANOTES
    - ("keys" := ks(stkc("I"))) ("ctls" := cs(ctl(pc("A"),pc("A1"))))
  - ROLE --- <Role>
    - role(rid,principal,rvars,out,ivars,notes,code)
  - PROTOCOL --- <Protocol>
    - protocol(name,vars,roles,notes)

Semantic Modules

- ENV --- binding variables
- EVENTS --- to record execution/bundle/run
  - event id --- elaboration of strand node
  - msg --- tagged with send node
  - rcv --- links receive node with send
- PLAYER --- participant in a run
  - [sid, rid, pid | n, code, codelist, env, notes]
- PCONF ~ PlayerConf RcvConf MsgConf --- execution state
- XMATCH --- defines pattern matching modulo
- TOOLIP-ALGEBRA -- axioms for the minimal crypto algebra
- XALGEBRA --- extending the basic crypto algebra
- EXE --- the rewrite rules interpreting code actions

Simulation & Analysis

- INITIAL
  - creating inital configurations for a protocol
  - uses role instances: ri(rid,principal,env,notes)
- ANALYZE --- helper functions for search patterns
- Protocol specific modules
  - <NAME> extends XMATCH --- defines the roles
  - <NAME>-INIT extends <NAME>, <INITIAL>
    - defines initial configurations, attacker roles
  - <NAME>-TEST extends <NAME>-INIT, EXE, ANALYZE
    - here you can rewrite and search

Toolip Semantics in Maude
Some details

TOOLIP-ALGEBRA

- Interprets function ops
- Extends substitution to constructors
- Interprets match code

- `xmatch(me,keys,controls,envset,exp,pat)`
  - extends each `env` in `envset` matching of `exp` to `pat`
  - `(me,keys,controls)` determines the available keys
  - `noEnv` -- the empty environment set indicates failure

Basic Pattern Matching

PatVar	Matches	Construction
nv(id)	nc(id')	Nonces
pv(id)	pc(id')	Principals
tv(id)	tc(id')	Text
ev(id)	exp	Any Expression
stkv(id)	stkc(id')	Short Term Key
ltkv(id,pcl)	ltkc(id',pcl)	Long Term Key
pubkv(id,pc)	pubkc(id',pc)	Public Key
privkv(id,pc)	privkc(id',pc)	Private Key
$[pat_1,\ldots,pat_k]$	$[exp_1,\ldots,exp_k]$	Tuples

if exp_i matches pat_i

Built in Encryption

```
op e : Key Exp -> Exp [ctor] .
op d : Key Exp ~> Exp .          **** semanic aux

ceq d(key,e(key',exp)) = exp if inv(key') == key .

eq fc("E",1,encr)[key](exp) = e(key,exp) .

eq subst(env,e(key,exp)) =
     e(subst(env,key), subst(env,exp)) .

ceq xmatch(me,keys,ctls,envs,exp,e(key,exp'))
    = xmatch(me,keys,ctls,envs,d(key',exp),exp')
    if key' := inv(key)
    /\ hasKey(me,keys,ctls,key')
    /\ d(key',exp) :: Exp .
```

Rewrite Rules (match)

```
crl[match]:
  [sid,rid,p |
   ix, match(exp,pat,tag),cdl, env, notes]
   =>
  [sid,rid,p | s ix, code, cdl, env',  notes']
  if ks(keys) := lookupd(notes,"keys",ks(none))
  /\ cs(ctls) := lookupd(notes,"ctls",cs(none))
  /\ envs := xmatch(p,keys,ctls,env,
                    subst(env,exp),subst(env,pat))
  /\ (env' ; envs') :=
      (if envs == noEnv
       then (mt ; noEnv) else envs fi)
  /\ code :=
      (if envs == noEnv then failed else noop fi)
  /\ notes' := joinNotes(notes,
         ("keys" := ks(namedKeys(env',none))),mt)
 .
```

Rewrite Rules (send)

```
rl[send]:
  [sid,rid,p |
   ix, send(sndr,rcvr,exp,tag),cdl, env, notes]
  =>
  [sid,rid,p | s ix, noop,cdl, env, notes]
  msg(subst(env,sndr),subst(env,rcvr),
      subst(env,exp),
      s(sid,rid,p,tag,ix))) .
```

Rewrite Rules (recv)

```
crl[recv]:
 [sid,rid,p |
  ix, recv(spv,rpv,zv,tag),cdl, env, notes]
 msg(sndr,rcvr,exp,eid)
   =>
 [sid,rid,p | s ix, noop,cdl, envs, notes]
 rcv(r(sid,rid,p,tag,ix),msg(sndr,rcvr,exp,eid)))
if envs :=
   xmatch(p,none,none,unbind(env,(spv, rpv, zv)),
          [sndr,rcvr,exp],[spv,rpv,zv])
  /\ envs :: Env
  /\ notes' :=
     (if exp :: Key
      then joinNotes(notes,("keys" := ks(exp)),mt)
      else notes fi) .
```

XALGEBRA
Extending the basic Algebra

Associative Pairing

```
op pr : Exp Exp -> Exp [ctor] .   **** assoc

eq fc("pr",2,other)[nil](exp0,exp1)
     = pr(exp0,exp1) .

eq subst(env,pr(exp,exp'))
     = pr(subst(env,exp),subst(env,exp')) .

eq xmatch(me,keys,ctls, envs, exp, pr(exp0,exp1))
     =  xmatchpr(me,keys,ctls, noEnv, envs,
                 exp',expl,exp0,exp1)
  if (exp',expl) := fringe(exp) .
```

Associative Pairing (xmatchpr)

```
****                          new    orig
eq xmatchpr(me,keys,ctls,envs', envs,
            expl, nil, exp0, exp1) = envs' .
****        left right  lpat  rpat

ceq xmatchpr(me,keys,ctls,envs',envs,
            (expl,exp),(exp',expl'),exp0,exp1)
 = xmatchpr(me,keys,ctls,(envs' ; envs1),envs,
            (expl,exp,exp'),expl',exp0,exp1)
if envs0 := xmatch(me,keys,ctls,envs,
                   unfringe((expl,exp)),exp0)
/\ envs1 := xmatch(me,keys,ctls,envs0,
                   unfringe((exp',expl')),exp1) .
```

Other extensions

Used to formalize examples from Cortier Survey

- Commuting encryption keys

```
e1 : e1(k1,e1(k2,exp)) = e1(k2,e1(k1,exp))
```

- Distributing encryption (homomorphism)

```
e2 : e2(k,pr(exp1,exp2))
       = pr(e2(k,exp1),e2(k,exp2))
```

Cortier Example (p16): NSpr

Cortier Protocol Specification

```
A,B : principal
Na,Nb : fresh numbers
PK, SK : principal -> key (key pair)
[_,_] : associative pairing
1. A -> B : {A,Na}PK(B)
2. B -> A : {[Na,Nb],B}PK(A)
3. A -> B : {Nb}PK(B)

Attack
i.1. I(A) -> B : {A,I}PK(B)
       --- B doesn't check I :: Nonce
i.2. B -> I(A) : {[I,Nb],B}PK(A)
ii.1. I -> A   : {I,[Nb,B]}PK(A)
       --- A doesn't check [Nb,B] :: Nonce
ii.2. A -> I   : {[[Nb,B],Na'],A}PK(I)
i.3. I(A) -> B : {Nb}PK(B)
```

Maude Protocol NSpr

```
protocol("NSpr", nil,
  role("init", pv("A"), pv("B"), nil, nil, mt,
    seq(new(nv("A")),
        send(pv("A"), pv("B"),
             e(pubkc("0", pv("B")), pr(pv("A"), nv("A"))), "i0"),
        recv(pv("X"), pv("Y"), ev("Z"), "i1"),
        match(ev("Z"), e(pubkc("0", pv("A")),
                         pr(pr(nv("A"), ev("nB")),pv("B"))), "i1z"),
        send(pv("A"),pv("B"), e(pubkc("0",pv("B")),ev("nB")), "i2")))
  role("resp", pv("B"), nil, nil, nil, mt,
    seq(recv(pv("X"), pv("Y"), ev("Z"), "r0"),
        match(ev("Z"),                      *** no nonce check
              e(pubkc("0",pv("B")),pr(pv("A"),ev("nA"))), "r0z"),
        new(nv("B")),
        send(pv("B"),pv("A"),e(pubkc("0", pv("A")),
                               pr(pr(ev("nA"),nv("B")),pv("B"))), "r1"),
        recv(pv("X"), pv("Y"), ev("Z"), "r2"),  *** nonce check
        match(ev("Z"), e(pubkc("0", pv("B")), nv("B")), "r2z"))),
  "algebra" := stv("toolDefault"))
```

Maude Attack Role

```
Attack Role: aRole =
  role("initA",pv("I"),(pv("A"), pv("B")),nil,nil,mt,
    seq(send(pv("I"), pv("B"),
             e(pubkc("0", pv("B")), pr(pv("A"),pv("I"))), "a1"),
        recv(pv("X"), pv("Y"), ev("Z"), "a0"),     ***intercept
        send(pv("I"), pv("A"), ev("Z"), "a2"),     ***forward
        recv(pv("X"), pv("Y"), ev("Z"), "a3"),
        match(ev("Z"),e(pubkc("0", pv("I")),
              pr(pr(pr(ev("nB"), pv("B")),ev("nA")), pv("A"))),"a4"),
        send(pv("I"), pv("B"), e(pubkc("0", pv("B")), ev("nB")), "a5")) )
```

Analyzing NSpr

```
nsInit =
  init(NSpr,mt,
       ri("init",pc("A"),(pv("B") := pc("B")),mt)
       ri("resp",pc("B"),mt,mt),
       0)

search [1] nsInit =>! RC:RcvConf MC:MsgConf PC:PlayerConf
  such that doneps("NSpr0",PC:PlayerConf)
        and doneps("NSpr1",PC:PlayerConf) .
```

Analyzing NSpr Attack

```
nsInitA =
  init(addRoles(NSpr,aRole),mt,
       ri("resp",pc("A"),mt,mt) ri("resp",pc("B"),mt,mt)
       ri("initA",pc("I"),
          (pv("A") := pc("A"))(pv("B") := pc("B")),mt),
       0)

search [3] nsInitA =>! RC:RcvConf MC:MsgConf PC:PlayerConf
such that doneps("NSprA0",PC:PlayerConf)   *** the attacker
      and doneps("NSprA2",PC:PlayerConf) .  *** duped responder
```

Cortier Attack

```
I->B {[A,I]}PK(B)
B->I {[[I,Nb],B]}PK(A)
I->A {[[I,Nb],B]}PK(A)
               A->I {[[[Nb,B],Na],A]}PK(I)
               I->B {Nb}PK(B)
```

No need for I to intercept

```
I->B {[A,I]}PK(B)

B->A {[[I,Nb],B]}PK(A)
```

Cortier Example (p22): TMN

Summary

Authors: M. Tatebayashi, N. Matsuzaki, and D.B. Newman (1989)

Summary: This is a symmetric key distribution protocol for digital mobile communication systems, such as cellular networks.

The only trusted key is the public key of the server.

For each session, the server verifies that the keys Ka and Kb have not been used in previous sessions.

Where does the latter get formalized? It says something about possible executions, but can't be expressed in the simple protocol formalism.

Cortier Specification

```
Protocol:
A, B, S : principal
Ka, Kb : fresh symkey
PK, SK : principal -> key (keypair)
{x}PK(S)*{y}PK(S) = {x*y}PK(S)

1. A -> S : B, {Ka}PK(S)
2. S -> B : A  --- How does S know A?
3. B -> S : A, {Kb}PK(S)
4. S -> A : B, {Kb}Ka

Attack:
C, D, S : principal
Kc, Kd : fresh symkey
i.3. B -> S : A, {Kb}PK(S)
ii.1. C -> S : D, {Kc}PK(S)
ii.2. S -> D : C
ii.3. D -> S : C, {Kd}PK(S)*{Kb}PK(S) (= {Kd*Kb}PK(S))
ii.4. S -> C : D, {Kd*Kb}Kc
```

Cortier Specification

```
ii.1' C -> S : {Kb}PK(S)
   --- fails because S checks freshness of Kb ??
```

Fixing missing sender, collapsing C,D to E, otherwise need further communication to exhibit Kb in Intruder knowledge.

```
1'. A -> S : A, B, {Ka}PK(S)
ii.1. E -> S : E,E, {Kc}PK(S)
ii.2. S -> E : E
ii.3. E -> S : E, {Kd}PK(S)*{Kb}PK(S) (= {Kd*Kb}PK(S))
ii.4. S -> E : E, {Kd*Kb}Kc

Maude> search [1] tmnConfE =>+ RC:RcvConf MC:MsgConf
 PC:PlayerConf such that doneps("TMN0",PC:PlayerConf) .

Solution 1 (state 14209)
states: 14210  rewrites: 626590 in 3109410ms
        cpu (3525060ms real) (201 rewrites/second)
....
```

Mapping Toolip to Maude NPA

NPA `Parameters'

- PROTOCOL-EXAMPLE-SYMBOLS
  - module defining the language of Messages (sort Msg)
- PROTOCOL-EXAMPLE-ALGEBRAIC
  - module giving equations satisfied by message expressions
- USER-INPUT defines
  - STRANDS-DOLEVYAO --- the attacker capabilities
  - STRANDS-PROTOCOL --- the protocol roles
  - ATTACK-STATE(n)
    - what regular player thinks and what attacker knows
  - USER-GRAMMARS --- hints for pruning the search space

Toolip NPA Symbols

```
<The Maude Toolip sort hierarchy>
sort Blob .  subsort Blob < Elt .
subsort Principal < Public .

--- msg constructors
ops pubkc privkc :  Principal -> Key [frozen] .
op stkc : Fresh -> Key [frozen] .
op ltkc : Principal Principal ->  Key [frozen] .
op nc : Fresh -> Nonce [frozen] .
op pc : String -> Principal  [frozen] .

--- Tuples  upto 6
op tup : Msg  Msg  -> Msg [frozen] .
op tup : Msg  Msg  Msg  -> Msg [frozen] .
 ....
```

Toolip NPA Algebra

```
**** Toolip encryption
  op e : Key Msg -> Blob [frozen] .

**** associative pairing
  op pr : Msg Msg -> Msg [frozen]   .

*** Bounded Associativity (for 3-depth)
  vars Xe Ye Ze : Elt .
  eq pr(Xe, pr(Ye, Ze)) = pr(pr(Xe,Ye), Ze) [nonexec] .
```

Toolip NPA (NSpr)

```
eq STRANDS-DOLEVYAO ---- threads to compose an attack
   = :: r :: [ nil | -(X), -(Y), +(pr(X, Y)), nil ] &
     :: r :: [ nil | -(pr(X, Y)), +(X), nil ] &
     :: r :: [ nil | -(pr(X, Y)), +(Y), nil ] &
     :: r :: [ nil | -(X), +(e(privkc(pc("i")),X)), nil ] &
     :: r :: [ nil | -(X), +(e(pubkc(P),X)), nil ] &
     :: r :: [ nil | +(pB), nil ]
  [nonexec] .

eq STRANDS-PROTOCOL =
     :: rA ::
     [ nil | +(e(pubkc(pB),pr(pA,nc(rA)))),
             -(e(pubkc(pA), pr(pr(nc(rA),enB),pB))),
             +(e(pubkc(pB), enB)), nil ]
     &
     :: rB ::
     [ nil | -(e(pubkc(pB),pr(pA,enA))),
             +(e(pubkc(pA),pr(pr(enA,nc(rB)),pB))),
             -(e(pubkc(pB), nc(rB))), nil ]
   [nonexec] .
```

Toolip NPA (NSpr Attack)

```
**** B has completed a responder run apparently with A,
**** and I has B's nonce

  eq ATTACK-STATE(0)
   = :: r ::
     [ nil, -(e(pubkc(pB),pr(pA,enA))),
              +(e(pubkc(pA),pr(pr(enA,nc(r)),pB))),
             -(e(pubkc(pB),nc(r))) | nil ]
     || nc(r) inI
     || nil
     || nil
  [nonexec] .
```

Next Steps

- Proving consistency of xmatch and equational axioms (in PVS)
- Automating Toolip2NPA map
- Specifying a semantic connection to CPSA
- Specifying/verifying correctness -- of connections/outcomes
- Formalizing a protocol logic in PVS?
- Engaging users?

???

Tools

Cryptographic Protocol Shape Analyzer (CPSA MITRE)

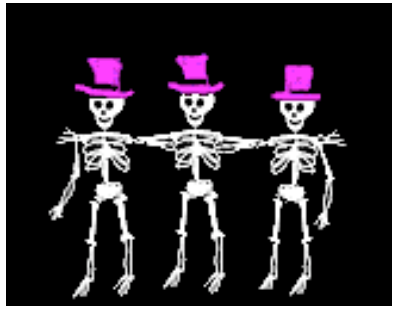
Protocol Derivation Assistant (PDA Kestrel)

NPA-Maude (NRL Protocol Analyzer in Maude
NRL, UIUC, U. Valencia)

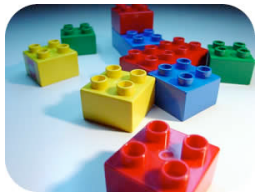
.....

CPSA

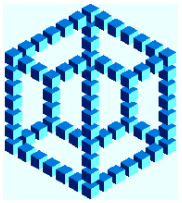
- Reasons about skeletons (regular part of bundle)
- Generate possible shapes by solving authentication tests, what else must have happened.
- Infers safe keys



PDA



- Abstract building blocks
  - basic protocol elements and associated properties
  - schema express constraints on parameters
- Derive more complex protocols and their properties by composition and refinement



- Underlying protocol composition logic

NPA-Maude

- Start with potential bad state and show it is unreachable
  - reason about what the intruder has or has not learned
  - run execution rules backwards
  - use grammars to prune search space