TOOLIP
Semantics & Interoperation

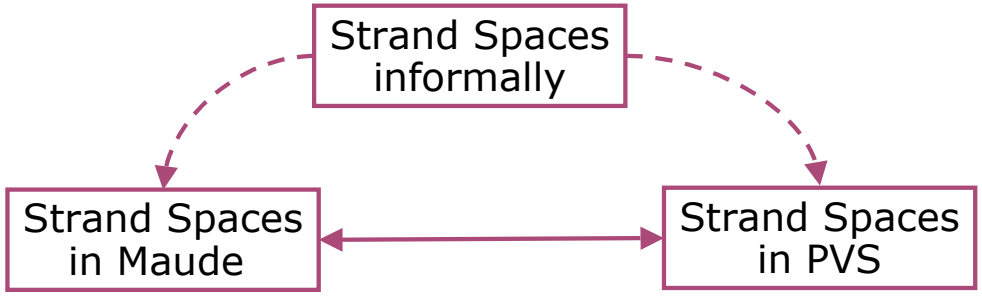
Carolyn Talcott
SRI International
Protocol Exchange, October 2008

Plan

- Vision recalled
- Semantic relations
  - Algebra at the core
- Interoperation
  - Context
  - Toolip <-> CPSA via maudeToolip
  - Interoperation by email
- Next?

Original Vision
(due to Sylvan Pinsky)

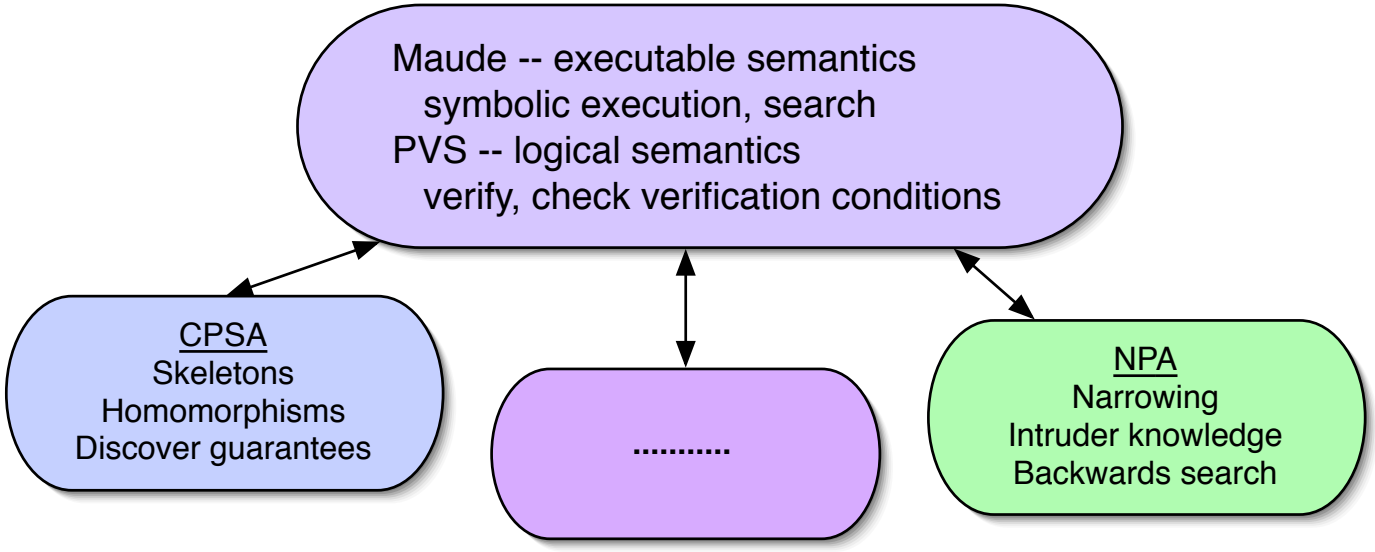
- A foundation for developing cryptographic protocol analysis algorithms
  - Specify and prototype in Maude
  - Verify in PVS



- 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)



Semantics Relations

- Semantic Core
- Toolip <-> maudeToolip
- maudeToolip <-> CPSA
- [maudeToolip <-> maudeNPA -- TBD]

Semantic Core

- For tools to interoperate meaningfully, they should be talking about the same set of terms and have common notion of execution that is what they are reasoning about.
- How they describe the terms or executions could be rather different. However, the properties studied should be about the shared notion of execution.
- The Toolip semantics core is a term algebra: an order sorted algebra with a distiguish sort Term.
- Executions are event diagrams
  - located event partial orders / bundles
  - Events are send/recv of terms

Term Algebra

- To avoid pathological cases, require
  - coherence, so connected components (of sort partial order) have an upperbound
  - regularity, so every term has a unique least sort.
- Other constraints?
  - matching / unification decidable
- Propose Maude functional modules as standard presentation language (order-sorted fragment)
- Are there any required sorts beyond Term?
  - Atom, Key, Principal/Name, Nonce ?

Toolip <=> maude Toolip

- Syntax: sexp2protocol(protocolSexp)
- Threat: irole
- Semantics
  - rinst ri(rid,p,env)
  - init(protocol, iroles, rinst) -- initial configuration
  - executions are EventConfs -- generated by rewrite rules
    - EventId: s(uid,rid,p,tag,ix) r(uid,rid,p,tag,ix)
    - SendEvent: msg(fromP,toP,exp,s(uid,rid,fromP,tag,ix))
    - RecvEvent: rcv(r(uid',rid',p',tag',ix'),
      msg(fromP,toP,exp,s(uid,rid,fromP,tag,ix))
    - s(uid,rid,fromP,tag,ix) < r(uid',rid',p',tag',ix')

maudeToolip <=> maudeNPA

- protocol + dolev-yao |= [maudeNPA]

  attackState ==> Strands + msgs

- init(protocol, irole(iStrands), rinst(Strands)) -> Pconf
  - events(Pconf) ~ msgs
  - Strands < Pconf

toolipMaude <=> CPSA

- protocol + preskel -CPSA-> shape
- init(protocol, irole, rinst) -> Pconf
  - regular(Pconf) ~ shape

toolipMaude
CPSA <=> NPA

- Given Toolip protocol
- Toolip skeleton
- => CPSA (protocol, skeleton)
- => CPSA shape
- => Toolip skeleton
- => NPA (strands, attack-state)
- => NPA solution
- => Toolip execution

Interoperation

- toolip protocol + cpsa Q (skeleton) -> maudeToolip -> CPSA -> shapes -> maudeToolip skeletons ...
- toolip protocol + npa Q (Strands and IK) -> maudeToolip -> maudeNPA -> NPA solutions -> maudeToolip executions
- toolip protocol + cpsa Q (skeleton) -> maudeToolip -> CPSA -> shapes -> maudeToolip skeletons -> maudeNPA -> NPA solutions -> maudeToolip executions

The Toolip CPSA Connection

Toolip Grammar
and
Maude representations

Protocols and Roles

- <Protocol> :=

```
(protocol <Version>
  <Ident>                    //  protocol name.
  (parameters <Variable>*)
  (roles  <role>+) [<Annotations>] )
```

- <role> :=

```
(role
  <Ident>                    // Role name
  <PrincipalVar>             // active principal
  (inputs <Variable>*)
  (outputs <Exp>*)           // [CPPL]
  <IndependentVariables>     //[Joshua] DROPPED
  (localconstraints <Local-Constraint>*)
  <Code>  )
```

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>)

- <Variable> -- an atom with <Kind> := var

CPSA restrictions

- no use of par
- every receive is of the form recv(X,Y,Z) followed by match(Z,exp) where X,Y are not used elsewhere (dummies)
- each binding occurrence of a variable has a unique name (no shadowing --- this is just a convenience)
- no symbols of kind const for now
  - different consts must be interpreted by distinct values which I don't think can be enforced by CPSA,
- non-originating keys (constructed from ids and principals) must have id "0" (or some other fixed constant). Also the principal list for ltk must be length 2.
  - Don't know how to give the same principal multiple public or private keys in CPSA.
- The only ops are tup, and generic enc. No nonce sort ?

maudeToolip Data Structures

```
protocol : String VarList RoleSet Anotes -> Protocol
protocol(id,pars,roles,anotes)

role : String PrincipalV VarList ExpList Anotes Code -> Role
role(rid,pvar,vars,exps,ivars,notes,code)

skeleton : String StrandList OccOrderSet AtomSet AtomSet
           [SexpL] -> Skeleton
skeleton(pid,strands,ordering,uniq,non,sexpl)

strand : String String Principal Nat Env -> Strand
strand(uid,rid,p,ht,env)

o : String String Principal Nat -> Occ --- event
o(uid,rid,p,ix)

oo : Occ Occ -> OccOrder                --- before
oo(o(uid,rid,p,ix) ,o(uid',rid',p',ix'))
```

Toolip 2 CPSA Protocols

Protocol mapping

```
<Protocol> :=
   (protocol <Version> <Ident> (parameters <Variable>*)
             (roles  <role>+) [<Annotations>]    )

protocol : String VarList RoleSet Anotes -> Protocol
sexp2protocol([a("protocol") a(version) a(id)
          [a("parameters") evsl] [a("roles") rsl] nsl ])
     = protocol(id,pars,roles,notes)
    if pars  :=  sexpl2varl(evsl)
    /\ roles :=  sexpl2roles(rsl)
    /\ notes :=  sexpl2notes(nsl) .

PROTOCOL   ::= (defprotocol ID ALG ROLE+ PROT-ALIST)
t2c-protocol : Protocol -> Sexp
t2c-protocol(protocol(id,pars,roles,anotes)) =
    mkcProtocolS(id,svStr(lookupd(anotes,"algebra",stv("basic"))),
                 t2c-roles(roles) ) .
```

Role mapping

```
<role> := (role <Ident> <PrincipalVar> (inputs <Variable>*)
             (outputs <Exp>*) (localconstraints ... ) <Code>)
sexp2role([a("role") a(id) pexp
              [a("inputs") evsl] [a("outputs") exl]
              [a("localconstraints") lcsl] cexp  ])
   = role(id,pvar,(pvl,pars),expl, notes, code)
   if {(pvar,pvl),ctls} := sexp2px(pexp)
   /\ pars := sexpl2varl(evsl)
   /\ expl := sexpl2expl(exl)
   /\ notes := sexpl2cstrs(lcsl) ; "cs" := cs(ctls)
   /\ code := sexp2code(cexp) .

ROLE ::= (defrole ID (vars VAR*) (trace DIRECTED+)
                 [(non-orig TERM*)]  [(uniq-orig TERM*)])
t2c-role(role(rid,pvar,vl,expl,notes,seq(codel)))
   = t2c-roleX(codel,rid,(pvar,vl),nil,nil,anotes2non(notes)) .
t2c-roleX(nil, rid, vl, trace, uniSL, nonSL)
   = [a("defrole") a(rid) [a("vars") vars2decls(vl)]
[a("trace") trace] [a("uniq-orig") uniSL] [a("non-orig") nonSL]] .
```

Code mapping

```
VAR         ::= (ID SORT)
DIRECTED    ::= (send TERM) | (recv TERM)

t2c-roleX((new(vl), codel), rid, vl0,trace,uniSL,nonSL)
    = t2c-roleX(codel, rid, addElts(vl0,vl),
                trace, uniSL expl2termSL(vl), nonSL) .

t2c-roleX((send(var0,var1,exp,tag),codel),rid,vl,trace,uniSL,nonSL)
   = t2c-roleX(codel, rid, vl, trace [a("send") exp2termS(exp)],
               uniSL, nonSL) .

t2c-roleX((recv(var0,var1,zvar,tag0), match(zvar,exp,tag1), codel),
                rid, vl, trace, uniSL, nonSL)
   = t2c-roleX(codel, rid, patvars(exp,vl),
               trace [a("recv") exp2termS(exp)], uniSL, nonSL) .
```

Exp mapping

```
SORT :: = text | name | skey | akey
TERM ::= ID | (pubk ID) | (privk ID) | (invk ID) | (ltk ID ID)
        |  (cat TERM+) | (cat STRING TERM TERM+) | (enc TERM TERM)

exp2termS(nv(id)) = a("n-" + id) .
exp2termS(ltkc("0", (pv(id1), pv(id2))))
    = [a("ltk") exp2termS(pv(id1)) exp2termS(pv(id2))]
exp2termS([expl]) =   [a("cat") expl2termSL(expl)]

var2decl(pv(id)) = [a("p-" + id) a("name")]
var2decl(nv(id)) =  [a("n-" + id) a("text")]
var2decl(tv(id)) = [a("t-" + id) a("text")]
```

Skeleton mapping

```
skeleton : String StrandList OccOrderSet AtomSet AtomSet SexpL
             -> Skeleton
skeleton(pid,strands,ordering,uni,non,sexpl)

PRESKELETON ::=
  (defpreskeleton ID (vars VAR*)  STRAND+
      [(non-orig TERM*)] [(uniq-orig TERM*)]
      [(precedes (NODE NODE)*] ... )
STRAND ::= (defstrand ID INT MAPLET*) | (deflistener TERM)
MAPLET ::= (TERM TERM)
NODE   ::= (INT INT)
```

CPSA Skeleton 2 Toolip

Skeleton mapping

```
cpsaskel2skeleton(protocol(id,pvars,roles,anotes),
                  [a("defpreskeleton") a(id)
                  [a("vars") decls]
                  sexpl ])
  = skeleton(id,strands,ordering,uniq,non,dsexpl)
    if vmap := decls2vmap(decls)
    /\ strands := cpsa2strands(id,roles,vmap,0,
                               get-tagsL(sexpl,"defstrand"))
    /\ ordering := cpsa2ordering(strands,get-tagL(sexpl,"precedes"))
    /\ uniq := cpsa2atoms(vmap,get-tagL(sexpl,"uniq-orig"))
    /\ non := cpsa2atoms(vmap,get-tagL(sexpl,"non-orig"))
    /\ dsexpl := (
          get-tagS(sexpl,"label")
          get-tagS(sexpl,"parent")
          get-tagS(sexpl,"operation")
          get-tagS(sexpl,"unrealized")
          get-tagS(sexpl,"comment")
        ) .
```

Strand mapping

```
*** vmap : id to cpsa skel vars,
decl2vmap([a(id) a("text")]) = id := exv(tv(id)) .
decl2vmap([a(id) a("name")]) = id := exv(pv(id)) .


cpsa2strand(pid,roles,vmap,ix,
            [a("defstrand") a(rid) a(istr) sexpl ])
  = strand(pid + string(ix,10),rid,p,rat(istr,10),env)
if env := maplets2env(vmap,sexpl)
/\ p := getPlayer(rid,roles,env) .

maplet2env(vmap, [a(id) sexp])
   = (id2var(id) := cterm2exp(vmap,sexp))
id2var(id) =
    (if prefix == "p-" then pv(rest)
     else (if prefix == "n-" then nv(rest)
     ...  ))
   if prefix := substr(id,0,2)
   /\ rest := substr(id,2,length(id)) .
```

Term mapping

```
cterm2exp(vmap,a(id)) = evalExp(lookupd(vmap,id,exv(ev(id)))) .

cterm2exp(vmap,[a("ltk") a(id0) a(id1)]) =
        ltkv("0",
             (evalExp(lookupd(vmap,id0,exv(ev(id0)))),
              evalExp(lookupd(vmap,id1,exv(ev(id1)))))) .

cterm2exp(vmap,[a("cat") sexpl]) = [cterml2expl(vmap, sexpl)] .
```

Interoperation by email

Ottway Reese – Scenario

```
A->B [M A B {NA M A B}KAS]      X = {NA M A B}KAS
B->S [M A B X {M A B NB}KBS]
S->B [M  {NA K}KAS {NB K}KBS]
B->A Y                          (Y = {NA K}KAS )

toolip-or  = "(protocol  "0.1"   or  (parameters) ...)"
toolip-orS = sexp2protocol(string2Sexp(toolip-or))

or-rStrand = strand("or1","resp",pv("B"),4,
         ((pv("A"):=pv("a"))(pv("B"):=pv("b"))(pv("S"):=pv("s"))
          (tv("M"):=tv("m"))(ev("X"):=ev("x"))(ev("Y"):=ev("y"))
          (nv("NB"):=nv("nb"))(stkv("K"):=stkv("k"))))
or-rSkel =
  skeleton("or",or-rStrand,none,nv("nb"),ltkc("0",(pv("b"),pv("s"))))

cpsa-orStr = sexp2str(t2c-protocol(orP),"")
cpsa-orRSkelStr = sexp2str(t2c-skeleton(or-rSkel))
cpsa-or-skelx16 -- found by cspa given orPStr and orRSkelStr
```

Ottway Reese Shape in Maude

```
cpsaskel2skeleton(orP,cpsa-or-skelx16) = skeleton("or",
strand("or0", "init", pv("p-a"), 2,
    (nv("NA"):=tv("n-na"))(tv("M"):=tv("t-m"))(stkv("K"):=stkv("s-k"))
    (pv("A"):=pv("p-a"))(pv("B"):=pv("p-b"))(pv("S"):= pv("p-s")))
strand("or1", "resp", pv("p-b"), 4,
  (ev("X"):=ev("e-x"))(ev("Y"):=ev("e-y"))(nv("NB"):=tv("n-nb"))
  (tv("M"):=tv("t-m"))(stkv("K"):=stkv("s-k"))(pv("A"):=pv("p-a"))
  (pv("B"):=pv("p-b"))(pv("S"):=pv("p-s")))
strand("or2", "serv", pv("p-s"), 2,
  (nv("NA"):=tv("n-na"))(nv("NB"):=tv("n-nb"))(tv("M"):=tv("t-m"))
  (stkv("K"):=stkv("s-k"))(pv("A"):=pv("p-a"))(pv("B"):=pv("p-b"))
  (pv("S"):=pv("p-s"))),
oo(o("or0", "init", pv("p-a"), 0), o("or1", "resp", pv("p-b"), 0))
oo(o("or1", "resp", pv("p-b"), 1), o("or2", "serv", pv("p-s"), 0))
oo(o("or1", "resp", pv("p-b"), 3), o("or0", "init", pv("p-a"), 1))
oo(o("or2", "serv", pv("p-s"), 1), o("or1", "resp", pv("p-b"), 2)),
tv("n-na") tv("n-nb") stkv("s-k"),   --- uniq
ltkv("0",pv("p-a"),pv("p-s")) ltkv("0",pv("p-b"),pv("p-s")), --- non
[a("label") a("16")] [a("unrealized")] [a("comment") a("shape")])
```

Next Steps

- Formalzing Algebras and mappings
- Automating Toolip2NPA map
- Formalizing the semantic relations
  - Toolip <-> CPSA
  - Toolip <-> NPA
- NPA Gui -- in progress

???