Writing and analyzing system policies in SUPPL

Robert Dockins (Galois, Inc.)
Andrew Tolmach (LRI Univ. Paris-Sud)
Alix Trieu (ENS Rennes)

SOUND project
Portland State University
SOUND Connection Service now extends to applications and users to protect communications from end point-to-endpoint. Users and applications authenticate using the SOUND Identity Service. Policies combined with sensor observations form SOUND decisions.

Applications connect to SOUND by way of the SOUND Connector, a system-level component of the SOUND Connection Service. Assertions are used to validate users to the application server.

SOUND supports two classes of applications:
1. SOUND-aware apps that are written for SOUND
2. Legacy apps that run within the SOUND Wrapper

SOUND Identity Service empowers SOUND to trace misbehavior back to all associated identities, including users, applications, and systems.

SOUND Data Service enables sensors to monitor data channels and attribute misbehavior to its origin.

Application servers request new direct connections to SOUND-protected data repositories. Data connections are bound to the identities associated with the data request for misbehavior attribution.

(SAFE and) SOUND
Configuring an Active Firewall

VPN-style network access on a per-connection level.
Configuring an Active Firewall

VPN-style network access on a per-connection level.

• Who may access network resources?
• Who is to be considered malicious and how is their access limited?
• Who gets what quality of service?
Configuring an Active Firewall

VPN-style network access on a per-connection level.

- Complex, changing configuration criteria
- Multiple administrators must cooperate to set policy
  - Set of administration domains is dynamic
  - Different administrators have different goals
- Multiple, complex platforms for implementing policy decisions
Existing tools are poor...

- Primitive user-driven configuration methods
  - Mysterious command-line incantations
  - Inflexible graphical control panels
  - Highly limited configuration languages
- Usually limited to static configuration
  - Cannot react to past behavior
- Must often fall back to ad-hoc coding
  - Fragile, hard to validate
Domain-Specific Languages to the rescue!

- Design a domain-specific **policy language**
  - specification
  - generating implementations
  - testing
  - **static analysis**
- (But not just for security policies...)
- cf. Ponder, PDL, ASL, FML, JRules,...
SUPPL

- Simple Unified Policy Programming Language
- Event-Condition-Action (ECA) policies
- Full-featured programming language
- Supports combining of policies
- Supports static conflict detection
- (Domain-neutral)
- (Modest performance goals)
Event-Condition-Action Policies

Policy Evaluator
(Policy Decision Point)

Governed Application
(Policy Enforcement Point)

Events
Primitives
Actions

Rest of the world

Data Tables
SUPPL Language Elements

- Like Prolog, but with strong type and mode disciplines
- Precise interface with surrounding environment
- “Locally stateless” style; data tables for handling stateful policies
Connection service: interface

primitive type principal.

data ip4address ::= ip(number, number, number, number).

event connectRequest
{ who : principal
, src : ip4address
, dest : ip4address }

event hostBehavesBadly (ip4address).

action allow.
action deny(string). % include reason for denial
Connection service: predicates and handlers

predicate \texttt{connectionAuthorized}(\texttt{principal} \ \texttt{in}, \ \texttt{ip4address} \ \texttt{in}).

c\texttt{onnectionAuthorized}(\texttt{PRINC}, \ \texttt{DEST}) :-

\hspace{1cm} \texttt{DEST} = \texttt{ip}(192,168,0,1).

c\texttt{onnectionAuthorized}(\texttt{PRINC}, \ \texttt{DEST}) :-

\hspace{1cm} \texttt{domain}(\texttt{DEST},\texttt{DOM}), \ \texttt{administrator}(\texttt{PRINC},\texttt{DOM}).

primitive predicate \texttt{administrator}(\texttt{principal} \ \texttt{in}, \ \texttt{number} \ \texttt{in}).

predicate \texttt{domain}(\texttt{ip4address} \ \texttt{in}, \ \texttt{number} \ \texttt{out}).
\texttt{domain}(\texttt{ip}(\texttt{A,B,C,D}), \ \texttt{N}) :- \ \texttt{A} = \ \texttt{N}.

handle \texttt{connectRequest} \ \{\texttt{who}:?P, \ \texttt{src}:?SADDR, \ \texttt{dest}:?DADDR\} =>

\hspace{1cm} \texttt{query}

\hspace{2cm} | \ \texttt{connectionAuthorized}(P,DADDR) => \texttt{allow};

\hspace{2cm} | \ _ => \texttt{deny("connection not authorized");}

\hspace{1cm} \texttt{end};

\hspace{1cm} \texttt{end}.
Firewall connection service: data tables

table blacklisted (ip4address,string)
key (in,out) lifetime 60000. % 60000 ms = 1 min

handle hostBehavesBadly(?ADDR) =>
    queue insert (?ADDR,"is badguy") into blacklisted;
end.

handle connectRequest {who:?P, src:?SADDR, dest:?DADDR} =>
    query |
        | connectionAuthorized(?P,?DADDR) =>
            query |
                | blacklisted(?SADDR,?WHY) => deny(WHY);
                | _ => allow;
            end;
        | _ => deny("connection not authorized");
    end;
end.
Types, modes, terms, clauses, queries

\[ t ::= \]
\[ \langle \text{id} \rangle \quad \text{named type} \]
\[ X, Y, Z, \ldots \quad \text{type variables} \]
\[ \text{list}(t) \quad \text{list} \]
\[ \text{set}(t) \quad \text{finite set} \]
\[ \text{map}(t_1, t_2) \quad \text{finite map} \]
\[ t_1 \ast \ldots \ast t_n \quad \text{tuple type} \]
\[ \text{number} \quad \text{numeric type} \]
\[ \text{string} \quad \text{string type} \]

\[ o ::= \]
\[ \text{in} \quad \text{out} \quad \text{ignore} \]

\[ c ::= \]
\[ m_1 = m_2 \mid m_1 <> m_2 \quad \text{(dis)equality comparisons} \]
\[ m_1 <= m_2 \mid m_1 < m_2 \quad \text{comparisons} \]
\[ m_1 >= m_2 \mid m_1 > m_2 \]
\[ \langle \text{id} \rangle(m_1, \ldots, m_n) \quad \text{predicate} \]
\[ c_1 | c_2 \quad \text{disjunction} \]
\[ c_1, c_2 \quad \text{conjunction} \]
\[ \text{not } c \quad \text{logical negation} \]
\[ c_1 \rightarrow c_2 \quad \text{implication} \]
\[ \text{findall}(m, g, X) \quad \text{find all} \]

\[ m ::= \]
\"literal", \ldots \quad \text{strings} \]
\[ 10, 3.14, 2.9e8, \ldots \quad \text{numbers} \]
\[ X, Y, Z, \ldots \quad \text{variables} \]
\[ ?X, ?Y, ?Z, \ldots \quad \text{variable bindings} \]
\[ _{} \quad \text{anonymous variable} \]
\[ \langle \text{id} \rangle(m_1, \ldots, m_n) \quad \text{function call} \]
\[ m_1 + m_2 \mid m_1 - m_2 \quad \text{numeric operations} \]
\[ m_1 * m_2 \mid m_1 / m_2 \]
\[ \sim m \quad \text{numeric negation} \]
\[ \text{[ ]} \quad \text{empty list} \]
\[ \text{[m_1, \ldots, m_n]} \quad \text{concrete list} \]
\[ \text{(m_1, \ldots, m_n)} \quad \text{list cons cell} \]
\[ \text{tuple} \]

\[ g ::= \]
\[ \langle \text{id} \rangle(m_1, \ldots, m_n) \quad \text{Goal} \]

\[ q ::= \]
\[ m_1 = m_2 \mid m_1 <> m_2 \quad \text{(dis)equality} \]
\[ m_1 <= m_2 \mid m_1 < m_2 \quad \text{comparisons} \]
\[ m_1 >= m_2 \mid m_1 > m_2 \]
\[ \langle \text{id} \rangle(m_1, \ldots, m_n) \quad \text{predicate query} \]
\[ q_1, q_2 \quad \text{conjunction query} \]
\[ \text{not } c \quad \text{negation query} \]
Handler bodies and declarations

\[ b ::= \]
\[ \begin{align*}
& b_1 \ b_2, \\
& \langle \text{id} \rangle (m_1, \ldots, m_n); \\
& \text{queue insert}(m_1, \ldots, m_n) \ 	ext{into} \langle \text{id} \rangle; \\
& \text{queue delete}(m_1, \ldots, m_n) \ 	ext{from} \langle \text{id} \rangle; \\
& \text{skip}; \\
& \text{query} q_1 \Rightarrow b_1; \ \ldots \ \mid q_n \Rightarrow b_n; \\
& \text{foreach} q \Rightarrow b \ 	ext{end}; \\
\end{align*} \]

\[ d ::= \]
\[ \begin{align*}
& \text{primitive type} \langle \text{id} \rangle. \\
& \text{type} \langle \text{id} \rangle := t. \\
& \text{data} \langle \text{id} \rangle ::= \\
& \quad \langle \text{id} \rangle (t_1, \ldots, t_n) \\
& \quad \ldots \\
& \quad \langle \text{id} \rangle (t_1, \ldots, t_n). \\
& \text{event} \langle \text{id} \rangle (t_1, \ldots, t_n). \\
& \text{action} \langle \text{id} \rangle (t_1, \ldots, t_n). \\
& \text{procedure} \langle \text{id} \rangle (t_1, \ldots, t_n). \\
& \text{primitive function} \langle \text{id} \rangle (t_1, \ldots, t_n) \\
& \quad \text{yields} \ t. \\
& \text{predicate} \langle \text{id} \rangle (t_1 o_1?, \ldots, t_n o_n?). \\
& \text{primitive predicate} \langle \text{id} \rangle (t_1 o_1?, \ldots, t_n o_n?). \\
& \text{mode} \langle \text{id} \rangle (o_1, \ldots, o_n). \\
& \text{table} \langle \text{id} \rangle (t_1, \ldots, t_n) \\
& \quad \text{key} (o_1, \ldots, o_n) \\
& \quad \text{(lifetime (int))?}. \\
& \text{index} \langle \text{id} \rangle (o_1, \ldots, o_n). \\
& \text{axiom} \ c. \\
& \text{handle} \langle \text{id} \rangle (?X_1, \ldots, ?X_n) \\
& \quad \Rightarrow b \ 	ext{end}. \\
& \text{define procedure} \langle \text{id} \rangle (?X_1, \ldots, ?X_n) \\
& \quad \Rightarrow b \ 	ext{end}. \\
& \text{axiom} \ c. \\
& \text{lemma} \ c. \\
\end{align*} \]
Type system highlights

• Built-in types for strings, numbers, lists, finite sets and finite maps
• User-definable type abbreviations and ML-style algebraic types
• Predicates, functions, events, etc. require type declarations
• Types are inferred in rule and handler bodies
• Hindley-Milner polymorphism for predicates and functions
Mode system highlights

- Modes ensure that arguments are always fully instantiated ground data (no variables).
- In mode: the argument must be instantiated before the call
- Out mode: argument will be instantiated after call
- Predicate mode(s) must be declared
Comparison to Prolog

- Very similar syntax for rules
- Standard operational semantics based on SLDNF resolution
- Turing-complete via recursive predicates
- SUPPL is strongly typed and moded
- SUPPL rules are side-effect free
- SUPPL has no nonlogical “cut” operator
- SUPPL has call-by-value semantics for evaluable functions and numeric operators (no “is” needed)
Prototype Implementation

Input policy file

Policy compiler

Compiled prolog

Rest of application

Interacts with

Custom library of primitives

Called from

Policy driver code

Called from

Loaded via

tuProlog.jar
Static conflict detection

- Policies may contain *conflicts*
  - One event triggers two inconsistent actions
  - e.g. both allowing and denying a request
- Some systems attempt dynamic conflict *resolution*
  - Based on priorities, or other meta-policies
  - Inherently fragile and hard to understand
- SUPPL approach: detect and fix conflicts statically
Example conflict

% Existing handler
handle connectRequest {who:?P, src:?SADDR, dest:?DADDR} =>
    query
    | connectionAuthorized(P,DADDR) => allow;
    | _ => deny("connection not authorized");
end;
end.

% Suppose new administrator adds this:

primitive predicate superuser(principal in).

handle connectRequest {who:?P, src:?SADDR, dest:?DADDR} =>
    query
    | superuser(P) => allow;
end;
end.
Example conflict

conflict allow, deny(_).

handle connectRequest {who:?P, src:?SADDR, dest:?DADDR} =>
   query
   | connectionAuthorized(P,DADDR) => allow;
   | _ => deny("connection not authorized");
end;
end.

primitive predicate superuser(principal in).

handle connectRequest {who:?P, src:?SADDR, dest:?DADDR} =>
   query
   | superuser(P) => allow;
end;
end.
When can conflict occur?

conflict allow, deny(_).

handle connectRequest {who:?P, src:?SADDR, dest:?DADDR} =>
  query
   | connectionAuthorized(P,DADDR) => allow;
   | _ => deny("connection not authorized");
  end;
end.

primitive predicate superuser(principal in).

handle connectRequest {who:?P, src:?SADDR, dest:?DADDR} =>
  query
   | superuser(P) => allow;
  end;
end.
Conflict detection: verification formulae

- Render conflict condition as first-order logic formula...
- ...in context of axioms characterizing (non-primitive) predicates and table keys
- If formula is satisfiable, the conflict might actually occur ⇒ so report to user
- If formula is unsatisfiable, the conflict is definitely impossible ⇒ so keep quiet
Encoding control flow

conflict \textit{allow, deny}(_).

\begin{verbatim}
handle connectRequest \{who:?P, src:?SADDR, dest:?DADDR\} =>
    query
    | connectionAuthorized(P,DADDR) => allow;
    | _ => deny("connection not authorized");
end;
end.
\end{verbatim}

primitive predicate \textit{superuser}(principal in).

\begin{verbatim}
handle connectRequest \{who:?P, src:?SADDR, dest:?DADDR\} =>
    query
    | superuser(P) => allow;
end;
end.
\end{verbatim}
conflict \texttt{allow, deny(\_)}. \\

\texttt{handle connectRequest \{who:?P, src:?SADDR, dest:?DADDR\} =>} \\
query \\
| \texttt{connectionAuthorized(P,DADDR) => allow;} \\
| \_ => \texttt{deny("connection not authorized");} \\
end; \\
end. \\

\texttt{primitive predicate superuser(\_).} \\

\texttt{handle connectRequest \{who:?P, src:?SADDR, dest:?DADDR\} =>} \\
query \\
| \texttt{superuser(P) => allow;} \\
end; \\
end.
Encoding predicates

predicate connectionAuthorized(principal in, ip4address in).

connectionAuthorized(PRINC, DEST) :-
    DEST = ip(192,168,0,1).

connectionAuthorized(PRINC, DEST) :-
    domain(DEST,DOM), administrator(PRINC,DOM).

primitive predicate administrator(principal in, number in).

predicate domain(ip4address in, number out).

domain(ip(A, B, C, D), N) :- A = N.

table blacklisted (ip4address,string)
    key (in,out) lifetime 60000. % 60000 ms = 1 min
Encoding predicates

predicate connectionAuthorized(principal in, ip4address in).

connectionAuthorized(PRINC, DEST) :-
    DEST = ip(192,168,0,1).
connectionAuthorized(PRINC, DEST) :-
    domain(DEST,DOM), administrator(PRINC,DOM).

primitive predicate administrator(principal in, number in).

predicate domain(ip4address in, number out).

domain(ip(A,B,C,D),N) :-
    A = N.

table blacklisted(ip4address,string)
key (in,out) lifetime 60000. % 60000 ms = 1 min

collection d :=
    d = ip(192,168,0,1) ∨
    (∃x.domain d x ∧ administrator p x)
Encoding predicates

domain d c := ∃ c1,c2,c3,c4. d = ip(c1,c2,c3,c4) ∧ c = c4

connectionAuthorized(PRINC, DEST) :- domain(DEST,DOM), administrator(PRINC,DOM).

administrator(principal in, number in).

domain(ip4address in, number out).

domain(ip(A, B, C, D), N) :- A = N.

table blacklisted (ip4address,string)
  key (in,out) lifetime 60000. % 60000 ms = 1 min
Encoding predicates

\[ \text{connectionAuthorized}(\text{PRINC}, \text{DEST}) :\text{-} \]
\[ \text{DEST} = \text{ip}(192, 168, 0, 1). \]
\[ \text{connectionAuthorized}(\text{PRINC}, \text{DEST}) :\text{-} \]
\[ \text{domain}(\text{DEST}, \text{DOM}), \text{administrator}(\text{PRINC}, \text{DOM}). \]

\[ \text{primitive predicate administrator}(\text{principal in, number in}). \]

\[ \text{predicate domain}(\text{ip4address in, number out}). \]
\[ \text{domain}(\text{ip}(A, B, C, D), N) :\text{-} A = N. \]

\[ \text{table blacklisted } (\text{ip4address,string}) \]
\[ \text{key (in,out) lifetime 60000. } \% 60000 \text{ ms} = 1 \text{ min} \]
Encoding predicates

```prolog
predicate connectionAuthorized(principal in, ip4address in).

connectionAuthorized(DEST = ip(19), domain(DEST), administrator).

primitive predicate administrator(principal in, number in).

predicate domain(ip4address in, number out).

domain(ip(A, B, C, D), N) :- A = N.

table blacklisted (ip4address, string)
  key (in, out) lifetime 60000. % 60000 ms = 1 min
```

blacklisted a s: uninterpreted
forall k d1 d2,
  blacklisted k d1 → blacklisted k d2 → d1 = d2
Fixing conflicts

• One way to fix the conflict is to change the SUPPL policy by adding the following clause:

```
connection_authorized(PRINC,DEST) :- superuser(PRINC)
```
Fixing conflicts

- One way to fix the conflict is to change the SUPPL policy by adding the following clause:

\[
\text{connection\_authorized}(\text{PRINC}, \text{DEST}) :- \\
\text{superuser}(\text{PRINC})
\]

Conflict formula is now unsatisfiable
Generating conflict formulae

• Find all control flow paths through the program, from event to action

• For each pair of paths that might be executed in response to a single event, generate a formula representing a potential conflict

• Being careful about quantifiers

• Have formalized this process for the core event handling language and proven it sound
Checking formulae

• Give the formulae to a standard SMT solver

• Currently done via Why3
  • Just a formula, not a Why3 program
  • Allows flexible choice of backends
  • Handles polymorphism, some built-in types

• CVC4, Alt-Ergo seem to be best provers for us
Why3 Example

theory Background
  type principal
  type ip4address = IP real real real real

  predicate administrator principal real
  predicate superuser principal

  predicate domain (a1:ip4address) (a2:real) =
    exists (x1 x2 x3 x4 : real) . a1 = IP x1 x2 x3 x4 /
    x1 = a2

  predicate connectionAuthorized (a1:principal) (a2:ip4address) =
    (exists x1. domain a2 x1 /
      administrator a1 x1)
    /
    a1 = IP 192.0 168.0 0.0 1.0
end

theory Conflict0
  use import Background

  function p : principal
  function dst : ip4address

  Axiom conflict: superuser p /
  not (connectionAuthorized p dst)
  Goal impossible: false
end
Feedback to user

• Currently via highlighted html pretty-printed from SUPPL source

• [demo]

• Currently working to improve specificity of feedback

• Would like to get models back from provers to instantiate variables in conflict
False positives

• If analysis reports no conflicts, there are none

• Converse definitely untrue!

• False positives due to
  • Uninterpreted predicates, tables
  • Recursive predicates
  • Prover incompleteness or timeout

• User can eliminate by passing hints to prover
Vocabulary of prover hints

• **axiom**: assertion about predicates or tables

• **lemma**: assertion about predicates or tables
  • which we ask prover to confirms

• **presume**: assertion about events

• **forbid**: assertion about actions

• All very dangerous!
  • but we do ask prover to find inconsistencies
SUPPL: Summary

• A domain-neutral language for programming ECA policies.

• Unusual combination of pure logic programming + imperative wrapper + data tables

• Designed to support static error and conflict checking
SUPPL: Status

• Prototype implementation compiles and runs simple policies for
  • an active proxying firewall
  • the SOUND research network platform
• Language details in APLAS14 paper
• Have initial implementation (and proofs of correctness) of conflict analysis [JFLA submission]
• Software is released under open-source license
web.cecs.pdx.edu/~rdockins/suppl

Thanks!

Questions?