Imperial College London

Sponsors of Tomorrow.™

Policy-Based Numerical Aggregation of Trust Evidence

Michael Huth and Jim Huan-Pu Kuo
Imperial College London

Outline of presentation

- (initial) project aim and objectives
- completed implementation work
- numeric aggregation and its verification
- how we see working within the Trust Evidence program

Project Aim

- understand how software annotations can generate trust evidence
- show that such evidence leads to policies that can effectively guard rail executions
- do this for both qualitative and quantitative evidence
- study programmers' intent as one source of evidence

Trust evidence

- increasingly, evidence for trust is numeric
- e.g. reputations of web sites
- e.g. age of software

- e.g. statistical information about past behavior of subjects
- e.g. probability of one machine connecting to another

Example

- numeric evidence may be needed even in seemingly qualitative settings

- e.g. you just downloaded the MacTex 2012 package and it does not have a matching hash

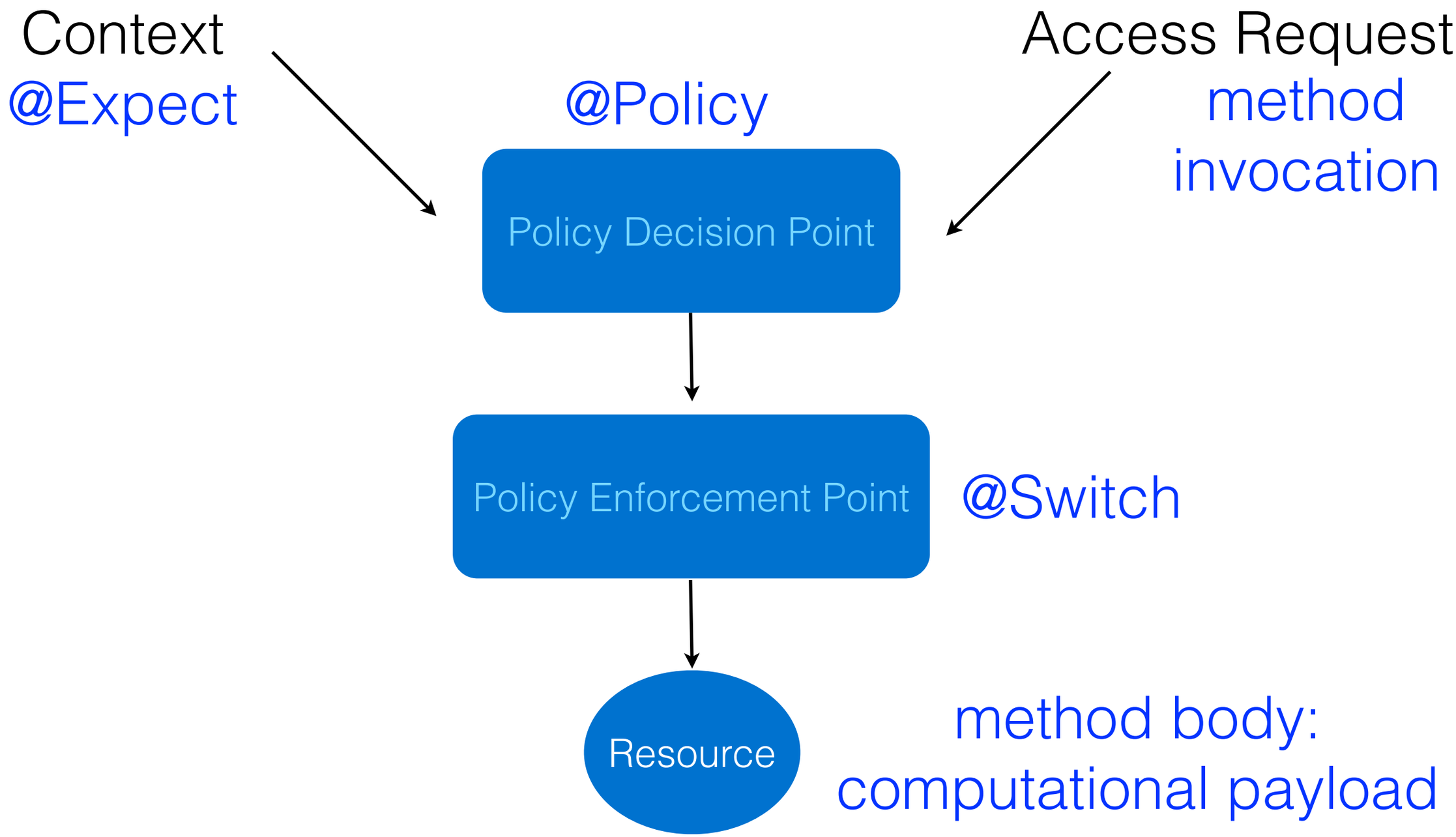
- how likely is it that this downloaded software is maliciously corrupted?

- conditional on which browser you use (e.g. Chrome does things to your file!)

Test bed of our idea

- explored guard railing method executions at callee sites for Scala code
- used a three-layered architecture for annotation-based computation of evidence for method executions
- **Layer 1:** expectations of behavior, from which trust evidence is computed (@Expect)
- **Layer 2:** policies that make access decision based on trust evidence and other factors (@Policy)
- **Layer 3:** policy enforcement (@Switch)

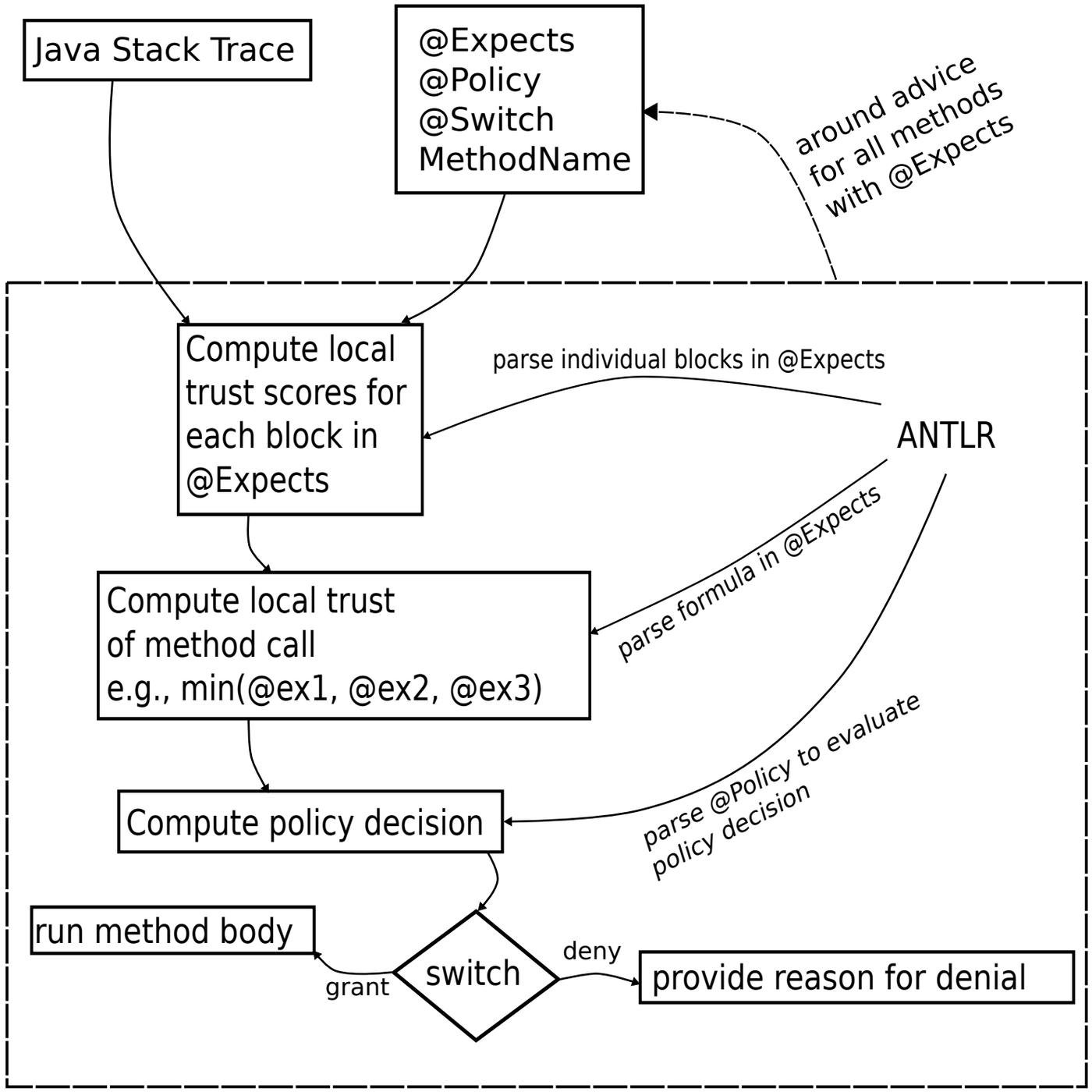
Layered annotations



Make evidence compositional

```
@Expect[ex1,max] default 0.1 {    // declares optimistic trust evidence
  (calledBy foo0) hasTE 0.9;
  (calledBy foo1) hasTE 0.3;
  (calledBy foo2) hasTE 0.8;
}
@Expect[ex2,+] default 0 {     // declares accumulative trust evidence
  (isSanitized(x)) hasTE 0.7;
  (calledBy foo3)  hasTE 0.2;
}
// conservative composition of optimistic and accumulative evidence
localTrust = min(@ex1, @ex2);
```

Data flow


Java Stack Trace
@Expects
@Policy
@Switch
MethodName
around advice
for all methods
with @Expects
Compute local
trust scores for
each block in
@Expects
parse individual blocks in @Expects
ANTLR
parse formula in @Expects
Compute local trust
of method call
e.g., min(@ex1, @ex2, @ex3)
parse @Policy to evaluate
policy decision
Compute policy decision
run method body
switch
deny
grant
provide reason for denial

Numeric aggregation and its verification

- want to use (non)-numeric evidence to produce recommendations or obligations

- which then should inform access-control decisions

- and at all sorts of hardware and software interfaces

- how can we verify such aggregation and what does verification mean here?

@Expect blocks as numeric policies “pol”

op ::= + | min | max

rule ::= if (q) score

pol ::= op(rule*) default score

- policy (pol) returns a score:
- first, collect all scores of rules in policy whose predicate (q) is true
- second, apply op to all these scores to get result; if no q is true, return default score

Policy composition

op ::= + | min | max
rule ::= if (q) score
pol ::= op(rule*) default score
pSet ::= pol | max(pSet,pSet) | min(pSet,pSet)
cond ::= th < pSet | pSet <= th

- policy isA policy sets
- combine policy sets (pSet) with min/max
- cond: compares values of pSet with thresholds

Trust expressed as “cond”

p1 = max((if (loc(u) = ext_trusted) 0.6)
if (loc(u) = www) 0.3)
if (loc(u) = inside) 0.8)) default 0

p2 = +((if affiliate(u) 0.1)
(if partner(u) 0.2)
(if (noEdit(o)) 0.3)) default 0

pSet = min(p1,p2)

cond = (0.5 < pSet)

Logical synthesis

- For each condition cond (“th < pSet” or “pSet <= th”): synthesize propositional formula **Φ**[cond] over predicates q

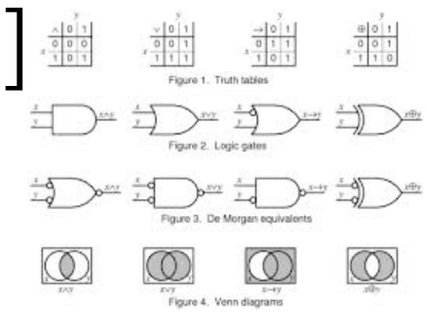


Figure 1. Truth tables

Figure 2. Logic gates

Figure 3. De Morgan equivalents

Figure 4. Venn diagrams

- **Φ**[cond] is true in model iff cond holds in model
- Synthesized **Φ**[cond] is basis for verification tasks
- predicates q themselves typically quantifier-free formulas of first-order logic

Formula for “th < pol”

op monotone, e.g. +

M1 set of all minimal index sets I whose score sum > th

$$\phi[th < pol] \stackrel{\text{def}}{=} \phi^{df}[th < pol] \vee \phi^{ndf}_{op}[th < pol]$$

$$\phi^{df}[th < pol] \stackrel{\text{def}}{=} (th < s) \wedge \bigwedge_{i=1}^{n} \neg q_i$$

$$\phi^{ndf}_{op}[th < pol] \stackrel{\text{def}}{=} \bigvee_{X \in \mathcal{M}_1} \bigwedge_{i \in X} q_i$$

+ composition

- let pol be +(rule*) default score
- sort predicates qi in pol by ascending order of scores
- pol may then we written as vector of scores, e.g. the vector [0.1, 0.2,0.2,0.3,0.5]
- want to find minimal index sets {i,...,j} of predicates whose score sum is greater than 0.5 (“0.5 < pol”), say
- here: M1 = {{4,5}, {3,5}, {2,5}, {2,3,4}, {1,3,4}, {1,2,4}}

Computing all minimal index sets: M1

call enumOne({},0,n+1,+) for n rules where ti = s0 + ... + si

$$
\begin{array}{l}
enumOne(X, sum, index, op)\ \{ \\
\quad if\ (th < sum)\ \{\ output\ X;\ \} \\
\quad else\ \{ \\
\quad\quad j = index - 1; \\
\quad\quad while\ ((0 < j) \wedge (th < op(sum, t_j))\ \{ \\
\quad\quad\quad enumOne(X \cup \{j\}, op(sum, s_j), j, op); \\
\quad\quad\quad j = j - 1; \\
\quad\quad \} \\
\quad \} \\
\}
\end{array}
$$

Example trace

th = 0.5, s = [0.1,0.2,0.2, 0.3, 0.5], t = [0.1,0.3,0.5,0.8,1.3]

```
enumOne({},0,6,+)
  enumOne({5},0.5,5,+)
    enumOne({5,4},0.8,4,+) --> {5,4}
    enumOne({5,3},0.7,3,+) --> {5,3}
  enumOne({5,2},0.7,2,+) --> {5,2}
  enumOne({5,1},0.6,1,+) --> {5,1}
enumOne({4},0.3,4,+)
  enumOne({4,3},0.5,3,+)
    enumOne({4,3,2},0.7,2,+) --> {4,3,2}
    enumOne({4,3,1},0.6,1,+) --> {4,3,1}
  enumOne({4,2},0.5,2,+)
    enumOne({4,2,1},0.6,1,+) --> {4,2,1}
```

Verification tasks

- Is **Φ**[cond] always true/false? Vacuity checking

- How do **Φ**[th < pSet] and **Φ**[th’ < pSet] differ? Sensitivity analysis

- Is cond safe, i.e. incomplete requests will grant access only if complete one would?

These and other tasks reduce to satisfiability checks of **Φ**[cond] over logic of predicates q

Ongoing work

- prototype verification tool using SMT solver Z3 (Microsoft)
- Note: a SAT solver would ignore logical dependencies of predicates q
- SMT solving needs to refine numerical spuriousness in satisfiability witnesses
- for this, we use Scala API (ETH tool) to interact with Z3
- will explore tradeoff between explicit synthesis of F[cond] (space intense) and synthesis of abstract F[cond] with refinement loop

Why we like working with Intel

- Gift/Donation funding asks us to publish results and source code in open-access mode: no lengthy negotiation of IP

- seed funding of blue-sky ideas

- monthly meetings with other PIs and yearly physical meetings: learned a lot

- Intel people are encouraging and supportive, e.g. “buddy scheme” works very well

Issues in working with Intel

- Academic institutions really want overheads for funded research
- Imperial students get the IP of their work, may conflict with Intel funding model
- teleconference times not family friendly for EU-based PIs ;-)
- UK Research Councils won't/cannot top up industry selected and awarded seed projects

This project makes us collaborate with ACEs!

- Jason Crampton, Information Security at Royal Holloway (an ACE)

- Iacovos Kirlappos and Angela Sasse, Information Security at UCL (an ACE)

- Charles Morrisset, Centre for Cybercrime and Computer Security at Newcastle (an ACE)

- Francesca Toni, Artificial Intelligence at Imperial College London (an ACE)

- hopefully with many more soon...

Thank You

Questions?