

# Towards Executable Access-Control Policies Written By Managers

Michael Huth

`imperial.ac.uk/quads/`

The 13th Nordic Workshop on Secure IT Systems

**Security for the Citizens**

9-10 October 2008

Copenhagen, Denmark



## Acknowledgments

Glenn Bruns  
Bell Labs  
Naperville, Illinois

Philip Inglesant & Angela Sasse  
University College London  
London, England



## Outline

Motivation

Belnap Logic

Core Policy Language

Extending the Core Language

Policy Analysis

Field Tests (to be done)

Conclusion



# Motivation



## Example Access-Control Policy

Consider partial campus policy, due to (Halpern & Weissman):

$$p = p1 \text{ merge } p2 \text{ merge } p3$$

Its sub-policies  $p_i$  model aspects of campus policy:

- ▶  $p1$  says “faculty has permission to assign grades”
- ▶  $p2$  says “students must not assign grades”
- ▶  $p3$  says “non-faculty has permission to enroll in courses”

What does this policy mean to policy writers? Can we enforce or analyze it? And if so, how?

## Access-Control Policies for Managers

### Intuitive & easy: policies

- ▶ become more important in domains beyond “security”
- ▶ need intuitive yet expressive composition mechanisms
- ▶ need push-button technology support for validation
- ▶ need intuitive and extensible concrete syntax

### Manageable complexity: policies

- ▶ need to deal with ambiguity and levels of granularity
- ▶ need to support change management
- ▶ may be read-only, implementation guide, or model extracted from implementation



## Requirements for Policy Language

### Design

- ▶ small language core with formal semantics
- ▶ policy combinators as syntactic sugar of core

### Analysis

- ▶ efficient policy analysis (SAT, BDDs, etc)
- ▶ support for gap/conflict analysis, refinement checks, etc

### Extras

- ▶ support for policy specialization and partial evaluation
- ▶ support for clean interface with application domains



## Human Factors

### Policy writers:

- ▶ build up Mental Models (as known in psychology) of policies and of their concrete syntax and meaning
- ▶ go through a learning process based on policy writing and analysis feedback: Mental Models evolve
- ▶ ought to be at the centre of the abstract-then-refine loop of iterative policy elaboration

## Human Factors Meet Policy Tools

### Inherent friction between:

- ▶ soft psychological needs stemming from considerations of Human Factors
- ▶ hard technological needs stemming from programming language design, analysis, and implementation technology

### Incentive

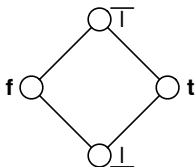
Highest return on investment in research will optimize the tradeoffs between such conflicting needs.



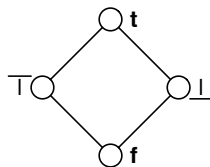
# Belnap Logic



## How Computers Should Think (Belnap 1976)



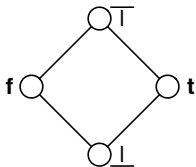
information ordering



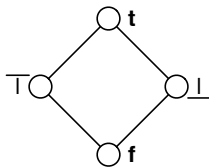
truth ordering

- ▶  $f$  (Deny),  $t$  (Grant),  $\perp$  (Undefined) or  $\top$  (Conflict)
- ▶ this four-element set forms a lattice **Four** in both
  - ▶ the information ordering  $\leq_i$  (left)
  - ▶ and the truth ordering  $\leq_t$  (right)

## Operations on Belnap Space **Four**



information ordering



truth ordering

- ▶ Negation:  $\neg f = t$ ;  $\neg t = f$ ;  $\top$  and  $\perp$  fixed
- ▶ Logical meet ( $\wedge$ ), join ( $\vee$ ) for truth ordering  $\leq_t$
- ▶ Information meet ( $\otimes$ ), join ( $\oplus$ ) for information ordering  $\leq_i$
- ▶ **contradictions are non-catastrophic**:  $x \wedge \neg x \leq_i y$  is not valid

# Core Policy Language



## Belnap Logic Inside

### Design Rationale

- ▶ policy writers/readers can't understand Belnap logic
- ▶ Belnap logic fine foundation for policy language
- ▶ thus we design core language with Belnap logic 'inside,' without policy readers/writers realizing this

### Two-level declarative programming:

- ▶ demote a policy expression into set of requests, those requests the policy grants (respectively, denies)
- ▶ promote set of requests into a policy



## PolCore: a Core Policy Language

	dec ::=	(Decision)	
	grant		
	deny		
reqs ::=	(Requests)		
ff	Falsity		
tt	Truth	pol ::=	(Policy)
reqsAtom	Atom	dec	Constant
!reqs	Neg.	pol when reqs	Promote
reqs & reqs	Conj.	pol merge pol	Merge
reqs   reqs	Disj.		
pol.dec	Demote		

## Reconsider Example Policy

### Sub-policy:

- ▶ p1 says “faculty has permission to assign grades”
- ▶ p2 says “students must not assign grades”
- ▶ p3 says “non-faculty has permission to enroll in courses”

### Formalized as `p1 merge p2 merge p3` in *PolCore* where

- ▶ `p1 = grant when faculty & grades & assign`
- ▶ `p2 = deny when student & grades & assign`
- ▶ `p3 = grant when !faculty & courses & enroll`



## What Semantics for *PolCore*?

### Quest for meaning

- ▶ What does `p1 merge p2 merge p3` mean?
  - ▶ More simply, what does `grant when faculty & grades & assign` mean?
  - ▶ Even more simply, what do `faculty`, `grades`, and `assign` mean?
- 
- ▶ University administrator: meaning of `faculty` obvious.
  - ▶ System administrator: need an implementation e.g.

```
boolean faculty(r: accessRequest) {  
    return r.role == Faculty;  
}
```



## Capture both responses abstractly

### Request model $\mathcal{M}$ :

- ▶ non-empty set  $R^{\mathcal{M}}$  of requests and, for each atom  $\text{reqsAtom}$ , a subset  $\text{reqsAtom}^{\mathcal{M}}$  of  $R^{\mathcal{M}}$
- ▶ terms  $\text{reqs}$  without sub-expressions  $\text{pol.dec}$  evaluated over  $\mathcal{M}$  as subset  $\llbracket \text{reqs} \rrbracket_{\mathcal{M}}$  of  $R^{\mathcal{M}}$ :

$$\llbracket \text{ff} \rrbracket_{\mathcal{M}} = \{ \}$$

$$\llbracket \text{reqsAtom} \rrbracket_{\mathcal{M}} = \text{reqsAtom}^{\mathcal{M}}$$

$$\llbracket r1 \ \& \ r2 \rrbracket_{\mathcal{M}} = \llbracket r1 \rrbracket_{\mathcal{M}} \cap \llbracket r2 \rrbracket_{\mathcal{M}}$$

$$\llbracket \text{tt} \rrbracket_{\mathcal{M}} = R^{\mathcal{M}}$$

$$\llbracket !\text{reqs} \rrbracket_{\mathcal{M}} = R^{\mathcal{M}} - \llbracket \text{reqs} \rrbracket_{\mathcal{M}}$$

$$\llbracket r1 \ | \ r2 \rrbracket_{\mathcal{M}} = \llbracket r1 \rrbracket_{\mathcal{M}} \cup \llbracket r2 \rrbracket_{\mathcal{M}}$$



## Meaning of `pol.dec`

### Expansion $\text{Expd}(\text{pol.dec})$ :

- ▶ element of `reqs` without sub-expressions `pol'.dec'`
- ▶  $\llbracket \text{pol.dec} \rrbracket_{\mathcal{M}} = \llbracket \text{Expd}(\text{pol.dec}) \rrbracket_{\mathcal{M}}$
- ▶  $\text{Expd}(\text{reqs})$  acts as identity over `reqs` without sub-expressions `pol'.dec'`, where  $* \in \{ \&, | \}$ :

$$\begin{array}{ll} \text{Expd}(ff) = ff & \text{Expd}(tt) = tt \\ \text{Expd}(\text{reqsAtom}) = \text{reqsAtom} & \text{Expd}(!\text{reqs}) = !\text{Expd}(\text{reqs}) \\ \text{Expd}(r1 * r2) = \text{Expd}(r1) * \text{Expd}(r2) & \end{array}$$

## Expansion for `pol.dec`

- ▶ We define  $\text{Expd}(\text{pol.dec})$  by structural induction on `pol`:

$$\text{Expd}(\text{dec.dec}) = \text{tt}$$

$$\text{Expd}(\text{dec.!dec}) = \text{ff}$$

$$\text{Expd}((\text{p when reqs}).\text{dec}) = \text{Expd}(\text{reqs}) \ \& \ \text{Expd}(\text{p.dec})$$

$$\text{Expd}((\text{p1 merge p2}).\text{dec}) = \text{Expd}(\text{p1.dec}) \ | \ \text{Expd}(\text{p2.dec})$$

where `!grant` is `deny`, and `!deny` is `grant`.

- ▶ Thus  $\llbracket \text{reqs} \rrbracket_{\mathcal{M}}$  is defined for all sets of requests `reqs`.

## Meaning of Policies

### Meaning of policy $p$ in request model $\mathcal{M}$

- ▶ total function from  $R^{\mathcal{M}}$  into Belnap space **Four**, i.e.  
 $\llbracket p \rrbracket_{\mathcal{M}}: R^{\mathcal{M}} \rightarrow \mathbf{Four}$ :

$$\llbracket p \rrbracket_{\mathcal{M}}(r) = \begin{cases} \perp & \text{if } r \notin \llbracket p.\text{grant} \rrbracket_{\mathcal{M}} \cup \llbracket p.\text{deny} \rrbracket_{\mathcal{M}} \\ \top & \text{if } r \in \llbracket p.\text{grant} \rrbracket_{\mathcal{M}} \cap \llbracket p.\text{deny} \rrbracket_{\mathcal{M}} \\ \mathbf{t} & \text{if } r \in \llbracket p.\text{grant} \rrbracket_{\mathcal{M}} \setminus \llbracket p.\text{deny} \rrbracket_{\mathcal{M}} \\ \mathbf{f} & \text{if } r \in \llbracket p.\text{deny} \rrbracket_{\mathcal{M}} \setminus \llbracket p.\text{grant} \rrbracket_{\mathcal{M}} \end{cases}$$

So  $p.\text{grant}$  means  $\geq_i \mathbf{t}$  and  $p.\text{deny}$  means  $\geq_i \mathbf{f}$ .



## Partial Evaluation of Campus Policy

Worked example  $p = p1 \text{ merge } p2 \text{ merge } p3$

- ▶  $\text{Expd}(p.\text{grant}) = (\text{faculty} \& \text{grades} \& \text{assign})$   
 $| (!\text{faculty} \& \text{courses} \& \text{enroll})$
- ▶ Query "Can students enroll in courses?" turns into  
 $[\text{student} \& \text{courses} \& \text{enroll}] \& \text{Expd}(p.\text{grant})$
- ▶ as courses are never grades, assignments never enrollments, this simplifies to  
 $\text{student} \& !\text{faculty} \& \text{courses} \& \text{enroll}$ , i.e. to  
 $!\text{faculty}$  with implicit query
- ▶ so a student can enroll in courses iff she is not faculty

# Extending the Core Language



## Adding Parameters and Methods to *PolCore*

### Language extension

Standard type system extends *PolCore* with parameters and methods of types `reqs` and `pol`.

### Policy composition through methods, e.g. Negation:

```
pol negation(P:pol) {  
  (grant when P.deny) merge (deny when P.grant)  
}
```

### Merge normal form of policy `p`

```
(grant when p.grant) merge  
(deny when p.deny)
```



## Some Policy Combinators

Predicates for gaps (`p.undef`) and conflicts (`p.incon`):

```
reqs undef(P:pol){  
  !P.grant & !P.deny }  
reqs incon(P:pol){  
  P.grant & P.deny }
```

Priority chaining (use infix `>` for that subsequently):

```
pol priorityChaining(P1:pol, P2:pol) {  
  (grant when P1.grant | (P1.undef & P2.grant))  
    merge  
  (deny when P1.deny | (P1.undef & P2.deny))  
}
```



## Exercise!

### Majority Vote

Takes three policies as input, and makes the decisions made by the majority of input policies:

```
pol MajorityOfThree(P1:pol, P2:pol, P3:pol){  
  (grant when WHAT GOES IN HERE?) merge  
  (deny when WHAT GOES IN HERE?)  
}
```

## Semantis of Method Invocations

### Expansion into core language

- ▶  $\text{methodFoo}(\vec{E}).\text{dec}$  expanded into core language  $\text{reqs}$ , gives meaning to invocation  $\text{methodFoo}(\vec{E})$  in *PolCore*
- ▶ requires extension of expansion function:

$$\text{Expd}(\text{Pvar}.\text{dec}) = \text{Pvar}.\text{dec}$$

$$\text{Expd}(\text{Rvar}) = \text{Rvar}$$

$$\text{Expd}(\text{methodPol}(\vec{E}).\text{dec}) = \text{Expd}(\text{bodyPol}[\vec{V}/\vec{E}].\text{dec})$$

$$\text{Expd}(\text{methodReqs}(\vec{E})) = \text{Expd}(\text{bodyReqs}[\vec{V}/\vec{E}])$$

## Example Semantics for Method Invocation

```
pol filter(P:pol, R:reqs) {  
  (deny when (P.grant & R)) > P  
}
```

**Expd(filter(P1,R1).dec) for dec = grant:**

$$\begin{aligned} ((\text{deny when } (P.\text{grant} \ \& \ R)) > P)[P/P1, R/R1].\text{grant} &= \\ ((\text{deny when } (P1.\text{grant} \ \& \ R1)) > P1).\text{grant} &= \\ \dots &= \text{!R1\&P1.grant.} \end{aligned}$$

# Policy Analysis



## Technical Device for Extended Policy Language

For all expressions  $reqs$ , the following are equivalent:

1.  $reqs$ , interpreted as formula  $\text{Expd}(reqs)$  of propositional logic, is valid
2.  $reqs$ , interpreted as propositional expression  $\text{Expd}(reqs)$  over unary predicates  $reqs_{\text{Atom}}$ , holds for all requests in all request models

### Insight

Proof requires synthesis of request model that captures all  $2^n$  propositional logic models for  $\text{Expd}(reqs)$ .

## Some Policy Analyses

### Gap analysis

Policy  $p$  free of gaps iff  $\text{Expd}(!p.\text{undef})$  valid as formula of propositional logic.

### Conflict analysis

Policy  $p$  free of inconsistencies iff  $\text{Expd}(!p.\text{incon})$  valid as formula of propositional logic.

### Equality

Policies  $p_1$  and  $p_2$  are equivalent iff  
 $\text{Expd}((p_1.\text{deny} \leftrightarrow p_2.\text{deny}) \& (p_1.\text{grant} \leftrightarrow p_2.\text{grant}))$   
valid as formula of propositional logic.



## More Policy Analyses

### Policy refinement

Policy  $p_2$  refines policy  $p_1$  iff

$$\text{Expd}((p_1.\text{grant} \rightarrow p_2.\text{grant}) \& (p_1.\text{deny} \rightarrow p_2.\text{deny}))$$

valid as formula of propositional logic.

### Blacklisting

Policy  $p$  blacklists set of requests  $\text{reqs}$  iff

$$\text{Expd}(\text{reqs} \rightarrow p.\text{deny} \& !p.\text{grant})$$

valid as formula of propositional logic.



## Analysis of Method Declarations

### Consider method declaration

```
pol filter(P:pol, R:reqs) {  
  (deny when (P.grant & R)) > P  
}
```

### Problem

**Static** extraction of constraints in terms of  $P$  and  $R$  that capture when invocation of `filter` is gap free, conflict free, etc?

### Solution

Expand “invocation” `filter(P,R).dec` as shown in a previous example.



## Analysis Findings as Pragmas

### Use of pragmas

Denote pre/post-conditions as in Eiffel, JML, and Spec#:

```
//@ gapfree if (P.grant & R) | !P.undef
//@ conflictfree if (P.grant & R) | !P.incon
pol filter(P:pol, R:reqs) {
  (deny when (P.grant & R)) > P
}
```

### Other pragmas

Pragmas `grants if`, `and denies if`, `and invariant` would also have use.



## Semantics of Pragmas

### As in ESC/Java

Assume pre-conditions when analyzing method body. Require pre-conditions when invoking method body.

### For `filter` with pragma `gapfree` if `P.grant`

- ▶ first discover exact condition  
 $(P.\text{grant} \ \& \ R) \mid !P.\text{undef}$  for gap freeness
- ▶ second check validity of  
 $P.\text{grant} \rightarrow ((P.\text{grant} \ \& \ R) \mid !P.\text{undef})$
- ▶ i.e. check validity of  
 $P.\text{grant} \rightarrow ((P.\text{grant} \ \& \ R) \mid (P.\text{grant} \mid P.\text{deny}))$  where  
`P.deny`, `P.grant`, and `R` are atomic propositions.



## Expressiveness of Extended *PolCore*

### Methods

Add no expressiveness to *PolCore*, just convenience.

### Predicate abstraction

For request model  $\mathcal{M}$ , set  $r_1 \equiv r_2$  iff  $r_1$  and  $r_2$  have same Boolean abstraction:

$$\forall \text{reqsAtom}: (r_1 \in \text{reqsAtom}^{\mathcal{M}} \leftrightarrow r_2 \in \text{reqsAtom}^{\mathcal{M}}).$$

### Language precise for predicate abstraction

Policy language expresses exactly those total functions  $f: R^{\mathcal{M}} \rightarrow \mathbf{Four}$  that factor through predicate abstraction:

$$\forall r_1, r_2 \in R^{\mathcal{M}}: r_1 \equiv r_2 \text{ implies } f(r_1) = f(r_2).$$



# Field Tests (to be done)



## Towards a Policy Language for Managers

### Develop tool prototype that

- ▶ presents extended policy language in Very Controlled Natural Language
- ▶ hides formal semantics from users
- ▶ implements salient policy analyses
- ▶ supports intuitions about language constructs through
  - ▶ Very Controlled Natural Language
  - ▶ feedback from policy analyses

## Experiments with Managers

- ▶ choose type of manager, e.g. clients of e-science infrastructures
- ▶ let them formulate and elaborate basic access-control policy about their domain
- ▶ determine what Mental Models they use and how that model evolves in elaboration process
- ▶ feed such insights back into policy design and analysis

## Experiments with Advanced Features

### Dynamic access

- ▶ *PolCore* seems like two-level functional programming (Nielson & Nielson 1992): levels are `pol` and `reqs`
- ▶ analysis was static: granting/denying a request won't change request model
- ▶ can our approach support dynamic notions (e.g. authorization, delegation, discretionary access control) within a user-friendly language?

### Other ideas, e.g. assume-guarantee reasoning

- ▶ we did work on static assume-guarantee reasoning and could expose users to this in experiments



# Conclusion



## What we did

- ▶ used Belnap logic to design two-level, core policy language
- ▶ interpreted policies over request models, which capture both real-world domains and implemented IT systems
- ▶ extended core language with methods and defined their meaning in core language
- ▶ methods express powerful policy combinators
- ▶ statically analyzed method declarations and policy expressions through validity checks (coNP)
- ▶ described how version of this language should be field-tested with non-experts who own assets a policy means to control

## Related work in Information Technology

### XACML

Composition algorithms of that standard can largely be written as methods in our language.

### Aspects

Technology for weaving policies into systems. Mapping policies onto aspect-oriented system: (Hankin, Nielson & Nielson 2008). How to extract policies from aspect-oriented code?

### Proof-carrying code (Lee & Necula 1996)

Policy pragmas checked with SAT solver for correctness. Correctness proof of `reqsAtom` harder, à la Lee & Necula.



Thank you for your kind attention!

