Towards Reasonability Properties for Access-Control Policy Languages

Michael Carl Tschantz[*]
Computer Science Department
Brown University

mtschant@cs.cmu.edu

Shriram Krishnamurthi
Computer Science Department
Brown University

sk@cs.brown.edu

ABSTRACT

The growing importance of access control has led to the definition of numerous languages for specifying policies. Since these languages are based on different foundations, language users and designers would benefit from formal means to compare them. We present a set of properties that examine the behavior of policies under enlarged requests, policy growth, and policy decomposition. They therefore suggest whether policies written in these languages are easier or harder to *reason* about under various circumstances. We then evaluate multiple policy languages, including XACML and Lithium, using these properties.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*

General Terms

Languages, Security

Keywords

Access control, policy, modularity

1. INTRODUCTION

Access-control policies should not be write-only. Because they govern both the containment and availability of critical information, they must be highly amenable to analysis by both humans and by reasoning software such as verifiers.

An access-control policy dictates a function from requests for access to decisions about whether or not to grant access. The competing requirements of expressive power and computational speed makes the design of policy languages a delicate balancing act. Contemporary policy languages have

---

[*]Current affiliation: Computer Science Department, Carnegie Mellon University.

largely followed one of two routes. Some are based on logics, restricting first-order logic (*e.g.*, Lithium [9]) or augmenting Datalog (*e.g.*, Cassandra [2]). Others are custom languages such as XACML [12] and EPAL [13], which behave roughly by rule-evaluation and do not depend on theorem-proving capabilities to determine a response to a query.

The custom language approach often produces fairly limited languages. For example, to express hierarchical role-based access-control (RBAC) [14] in XACML requires a fairly cumbersome encoding [1]. On the other hand, its more direct request evaluation strategy suggests that policies written in XACML are more transparent than policies written in languages based on first-order logic (as we motivate in Section 2).

How, then, do we distinguish different policy languages? Studies of complexity and expressive power may ensure tractable verification and the ability to capture certain policies, but do not directly classify the ease of reasoning about policies in a language. In this paper we take a step towards formalizing *reasonability* properties that make languages more amenable to reasoning. We then apply these properties to actual policy languages. Such properties are useful even when verification is computationally tractable because they provide a guide to where and how to edit a policy for a desired effect.

Concretely, our properties study three main questions: how decisions change as requests include more information, how decisions change as policies grow, and how amenable policies are to compositional reasoning. The last of these is especially important for two reasons. First, organizations increasingly have different divisions creating policy fragments that must be combined into a whole while preserving the intent of each division; second, to mirror these use cases, and to scale better as policies grow in size, it becomes important for analysis and verification tools to function modularly.

These properties codify our observations made while writing and studying policies for non-trivial systems. (We do not, however, presume to make broad statements about the impact of these properties for manual reasoning.) They are meant to be descriptive rather than prescriptive: which ones a language should satisfy depends on the context of its use. We do expect these properties to help both language designers and policy authors, the former to set goals and the latter to evaluate languages.

We first motivate the work with an example. Section 3 presents background on policy languages. Section 4 presents the heart of our formalism. Section 5 applies this framework to XACML, and Section 6 to logical approaches such as

Lithium. The remainder discusses related work and offers concluding remarks.

2. MOTIVATING EXAMPLE

Consider the following natural-language policy:[1]

1. If the subject is a faculty member, then permit that subject to assign grades.

2. If the subject is a student, then do not permit that subject to assign grades.

3. If the subject is not a faculty member, then permit that subject to enroll in courses.

We might represent this policy as follows:

$$\texttt{faculty}(s) \implies \texttt{Permit}(s, \texttt{grades}, \texttt{assign}) \qquad (\mathfrak{p}_1)$$
$$\texttt{student}(s) \implies \neg\texttt{Permit}(s, \texttt{grades}, \texttt{assign}) \qquad (\mathfrak{p}_2)$$
$$\neg\texttt{faculty}(s) \implies \texttt{Permit}(s, \texttt{courses}, \texttt{enroll}) \qquad (\mathfrak{p}_3)$$

Let the above formalization be $\mathfrak{p}$ and the first line of the policy be sub-policy $\mathfrak{p}_1$, the second $\mathfrak{p}_2$, and the third $\mathfrak{p}_3$.

Consider the following natural-language request:

A student requests to enroll in courses.

Assume that requests list the subject, resource, and action by name if possible and by variable if the name is unknown, along with any other known facts. In this representation, the request becomes:

$$(s, \texttt{courses}, \texttt{enroll}) \quad \text{with} \quad \texttt{student}(s) \qquad (\mathfrak{q}_1)$$

Should the policy grant access? At least three interpretations of the policy are possible:

1. $\mathfrak{p}$ grants access due to $\mathfrak{p}_3$. The request does not show the subject being a faculty member; thus, $\mathfrak{p}_3$ applies and $\mathfrak{p}$ produces the decision to permit access. This relies on the assumption that since the request does not show the subject being faculty, that the subject is in fact not faculty. One could drop this assumption.

2. The policy does not apply to the request. One would reason that $\mathfrak{p}_1$ and $\mathfrak{p}_2$ do not apply since they are dealing with assigning grades and not enrolling in courses. Furthermore, one could conclude that $\mathfrak{p}_3$ does not apply since the request does not prove that the subject is not faculty. To do so, the request would have been

$$(s, \texttt{courses}, \texttt{enroll}) \quad \text{with} \quad \texttt{student}(s) \wedge \neg\texttt{faculty}(s)$$

Since the policy does not apply to the request, the system should have and enact some default behavior.

3. By reasoning different than that used in the first interpretation, $\mathfrak{p}$ could still grant the request. As in the second interpretation, one could conclude that the request alone fails to establish that the subject is not a faculty member. However, if the subject were a faculty member, then the first two lines together would yield a contradiction: $\mathfrak{p}_1$ would imply that the subject could enroll in courses and $\mathfrak{p}_2$ would imply that the subject could not. Thus, student-faculty members do not exist. Since the subject of the request is clearly a student, he must not be faculty member. Thus, $\mathfrak{p}_3$ applies to grant access.

[1]This example is adapted from Halpern and Weissman [9].

In the first two interpretations the user may limit his reasoning to each sub-policy independent of one another. However, under the third interpretation (which, in fact, is the one chosen by Halpern and Weissman), the user must reason about all three sub-policies at once. Furthermore, under Interpretation 2, the user must reason about both positive and negative attributes, unlike under Interpretation 1.

These semantic differences drastically affect a reader's ability to comprehend policies. For example, Interpretation 3 requires both global analysis and demands rich reasoning power to deduce the contradiction. This paper formalizes these differences and their burdens.

3. BACKGROUND

Despite the differences between access-control policy languages, we can still identify many common elements. First we describe features common to most languages, and then we discuss in detail two areas in which many languages differ: the available decisions and policy combinators.

3.1 Common Features

A policy language must provide a way of describing the different forms of access and the environment in which they could occur. This information forms a *request*. Many languages break requests into four different parts:

Subject the person or process making the request,

Resource the object, subsystem, person, or process that the subject would like to affect (*e.g.*, a file name or a process id),

Action the command or change that the subject would like to execute on the resource, and

Environment describes any other relevant information including the time of day, location, or the previous actions of the subject.

The first three make up the form of access requested while the last gives the context in which this access would occur. Each of these parts lists attributes associated with its respective topic. In some languages, the absence or negation of an attribute might also be explicitly listed (see Section 4.2).

Languages must also provide a set of *decisions*. Such a set must include some decisions that grant access and some that prohibit access. A policy will associate with each request a decision (or in the case of nondeterministic policies, a policy will relate each request with some number of decisions).

DEFINITION 3.1. *An access-control policy language is a tuple $\mathcal{L} = (P, Q, G, N, \langle\!\langle \cdot \rangle\!\rangle)$ with*

P *a set of policies,*

Q *a family of sets of requests indexed by policies,*

G *the set of decisions that stipulate that the system should grant access (granting decisions),*

N *the set of decisions that stipulate that the system should not grant access (non-granting decisions),*

$\langle\!\langle \cdot \rangle\!\rangle$ *a function taking a policy $p \in P$ to a relation between Q_p and $G \cup N$,*

where $G \cap N = \emptyset$.

When clear from context, the above symbols will be referenced without explicitly relating them to $\mathcal{L}$, and D will represent $G \cup N$. The function $\langle\!\langle \cdot \rangle\!\rangle$ gives the meaning of policy p, and we write $q\langle\!\langle p \rangle\!\rangle d$ for $p \in P$, $q \in Q_p$, and $d \in D$, when p assigns a decision of d to the request q. If for a language $Q_p = Q_{p'}$ for all p and p' in P, then we drop the subscript on Q and treat it as a set of requests common to all policies. Given $\mathcal{L}$ define the partial order $\leq$ on D to be such that $d \leq d'$ if either $d, d' \in N$, $d, d' \in G$, or $d \in N$ and $d' \in G$.

3.2 Decisions

Policy languages must provide decisions to indicate a policy's intent to grant or not to grant a request. Some languages might just provide two decisions: *permit* for granting access and *deny* for not granting access. A policy in such a language associates various subsets of requests with one of these two decisions. For example, $\mathfrak{p}_1$ explicitly identifies a subset of permitted requests and $\mathfrak{p}_2$ gives denied requests.

However, a policy might assign some requests to neither permit nor deny (*e.g.*, $\mathfrak{q}_1$ under Interpretation 2 of $\mathfrak{p}$). To err on the side of safety, the policy language should provide for such requests a default decision that does not imply a grant of access creating a *closed* policy [10]. However, assigning them the decision of deny may limit the ability to compose policies. For example, while combining the policies of two departments, one would like to distinguish between those requests that each department really would like to prohibit and those about which they do not care [3]. The decision of *not applicable* serves this purpose.

With a decision of not applicable sufficing to prevent access, some languages elect not to include statements associating requests with deny. This leaves only statements permitting some set of requests. The uniformity of statements in such languages might make the policy easier to read and compose (see Section 3.3). However, allowing for the explicate denial of requests can quickly rule out exceptional cases and provides a means to determine when a policy does not grant access by desire rather than by default.

Some requests might not have a logical interpretation under a given policy. For example, a request of

$$(s, \texttt{grades}, \texttt{assign}) \quad \text{with} \quad \texttt{faculty}(s) \wedge \texttt{student}(s) \quad (\mathfrak{q}_2)$$

under Interpretation 3 of $\mathfrak{p}$ contradicts the policy itself. A request might even contain illogical values or require undefined computation (such as division by zero). For generality, a system might like to assign a decision to such inputs rather than excluding them from the set of requests and leaving the policy undefined on them. In such cases, a decision of *error* or some refinement of it might be appropriate.

One may view the fact that an error state is reached given a request to be a weakness in the policy. However, one may also take it to be a statement about the world in which the policy is to function: that no such requests may logically exist. Error decisions can enforce these preconditions or assumptions that the policy has made.

3.3 Policy Combinators

The policy of an organization often consists of the composition of policy fragments, or *sub-policies*, from a variety of internal units (*e.g.*, legal, accounting, and execute departments of a corporation). Thus, policy languages provide *combinators* to create a single policy from these fragments.

Under $\mathfrak{p}$, the request given above ($\mathfrak{q}_2$) is permitted by $\mathfrak{p}_1$ but denied $\mathfrak{p}_2$. The method used to combine the three sub-policies of $\mathfrak{p}$ into one policy determines how to resolve this conflict. Some languages, like the hypothetical language in which $\mathfrak{p}$ is written, might have only one policy combinator that is implicitly applied. Other languages provide multiple combinators. If a policy has sub-policies nested inside of sub-policies, the different layers may be resolved differently.

Some of the possible policy combinators are:

Permit Overrides If any of the sub-policies produces a permit, return only permit. Otherwise, if any produces a deny, return only deny. Else, return not applicable.

Deny Overrides If any of the sub-policies produces a deny, return only deny. Otherwise, if any produces a permit, return only permit. Else, return not applicable.

First Applicable Return the decision reached by the first sub-policy to reach one other than not applicable.

All Seen Return a set containing the decisions reached by all the sub-policies.

Either Permit or Deny Nondeterministically return one of the produced decisions.

Error Return a error if the sub-policies produces both permit and deny. Otherwise return permit if produced, or deny if produced. Else, return not applicable.[2]

And Conjoin the sub-policies together by logical And and return the implied decision(s).

De Capitani di Vimercati *et al.* [4] list additional combinators. The nature of the combinators available in a language can greatly impact the clarity of policies written in it.

Notice that many of the above combinators behave the same in the absence of the decision of deny. One might conclude from this observation that allowing the explicit denial of a request is an undesirable complication in a language.

To formalize the role of policy combinators, let policies be either an atomic policy or a set of sub-policies combined by some policy combinator. Let p be a policy that consists of sub-policies p_i with $1 \leq i \leq n$. Then $p = \oplus(p_1, p_2, \ldots, p_n)$ represents the composition of the sub-policies using $\oplus$.[2]

Since sub-policies are themselves policies, one may apply $\langle\!\langle \cdot \rangle\!\rangle$ to them.[3] The relationship between $\langle\!\langle \oplus(p_1, p_2, \ldots, p_n) \rangle\!\rangle$ and meaning of each sub-policy p_i affects the clarity of the policy and is studied in the next section.

4. POLICY LANGUAGE PROPERTIES

Having formalized policy languages, we are now ready to describe properties of them.

---

[2] We assume that the set of combinators in a given language $\mathcal{L} = (Q, P, G, N, \langle\!\langle \cdot \rangle\!\rangle)$ is clear from the structure of P and $\langle\!\langle \cdot \rangle\!\rangle$. If this is not the case for a language, one could explicitly add it to the definition of an policy language.

[3] Some languages may permit contextual information from enclosing policies to affect the meaning of the sub-policies. For example, a language might have a notation of variable binding. For such a language, $\langle\!\langle \cdot \rangle\!\rangle$ might be extended to take a second argument that carries such contextual information. All the following definitions could be extended, *e.g.*, monadically, to deal with such an extended $\langle\!\langle \cdot \rangle\!\rangle$.

4.1 Determinism and Totality

DEFINITION 4.1. *A language $\mathcal{L} = (P, Q, G, N, \langle\!\langle \cdot \rangle\!\rangle)$ is* **deterministic** *if*

$$\forall p \in P, \forall q \in Q_p, \forall d, d' \in D, \ q\langle\!\langle p \rangle\!\rangle d \wedge q\langle\!\langle p \rangle\!\rangle d' \implies d = d'$$

For a deterministic language, we can define a function $[\![\cdot]\!]$ which takes a policy $p \in P$ and returns a function from Q_p to D as $\lambda p \,.\, \lambda q \,.\, d \in D$ s.t. $q\langle\!\langle p \rangle\!\rangle d$. For a deterministic language, $[\![\cdot]\!]$ may be given instead of $\langle\!\langle \cdot \rangle\!\rangle$ to define the language. (We only even mention nondeterministic languages due to the existence of one: XACML with obligations.)

DEFINITION 4.2. *A language $\mathcal{L} = (P, Q, G, N, \langle\!\langle \cdot \rangle\!\rangle)$ is* **total** *if*

$$\forall p \in P, \forall q \in Q_p, \exists d \in D \text{ s.t. } q\langle\!\langle p \rangle\!\rangle d$$

The policies of total languages will always make a decision.

4.2 Safety

Under Interpretation 2, the request contained too little information to determine which of the sub-policies of $\mathfrak{p}$ applied. Interpretation 1 avoids such indecision by having requests implicitly refute the presence of any attribute not listed. These two interpretations produce different meanings for statements like $\neg\texttt{faculty}(s)$ found in $\mathfrak{p}_3$. Under Interpretation 1, $\neg\texttt{faculty}(s)$ holds if $\texttt{faculty}(s)$ is not in the request, while under the second, the request must explicitly list $\neg\texttt{faculty}(s)$ for it to hold. We call the former *implicit* and the latter *explicit*.

The explicit approach permits distinguishing between unknown information and attributes known to be absent. The explicit interpretation, however, incurs the cost of listing a possibly large set of absent attributes and can lead to indecision as shown above.

Such indecision, however, allows the system to recognize when the policy requires more information to yield a decision. In contrast, the implicit interpretation can grant undue access. If, for example, a request does not list $\texttt{faculty}(s)$ simply because the system did not determine whether s was a faculty member or not, then the system might erroneously allow s to enroll in courses. Thus, the sub-system producing requests must be sure to include all the relevant facts in each request.

For large scale systems, collecting or even determining the germane information might consume large amounts of time. For such systems, the explicate approach might prove better since requests may leave out information safely and be refined until the policy yields a decision. Furthermore, overzealous optimizations and other coding errors might result in the system producing requests that do not contain all the relevant facts.

Having a policy drive which information requests include allows for the system to collect only the information really needed to reach a decision from the policy. Under this approach, the sub-system evaluating the policy starts with a request that contains only the readily available information. If this sub-system needs additional information to reach a decision from the policy, it requests the necessary additional information. Thus, the system does not need to know what information the policy requires at the time of generating the initial request. This approach may allow for more efficient implementations.

Once a datum has been published, it cannot easily be retracted. Therefore, preventing unwanted access is usually preferable to granting it. As a result, such incomplete requests should only result in a grant of access if the complete one would have. We can formally state this safety concern:

DEFINITION 4.3. *Let $\subseteq$ be a family of partial ordering on requests of a language $\mathcal{L} = (P, Q, G, N, [\![\cdot]\!])$ indexed by the policies of $\mathcal{L}$. $\mathcal{L}$ is* **safe** *with respect to $\sqsubseteq$ iff*

$$\forall p \in P, \forall q, q' \in Q_p, \ q \sqsubseteq_p q' \implies [\![p]\!](q) \leq [\![p]\!](q').$$

Due to differences in the contents of a requests, for each language a different family of partial orderings $\sqsubseteq$ will interest users. The relation should be such that if $q \sqsubseteq_p q'$, then q' contains all the information contained in q and possibly more. Often one partial ordering may serve every policy.

For example, consider a language in which requests are sets of non-contradictory facts and the set of decisions is $\{\texttt{permit}, \texttt{deny}\}$. Then using the subset partial ordering for $\sqsubseteq_p$ (for every policy p) will make sense since it matches the intuition of information content. If the language is safe with respect to such a defined $\sqsubseteq$, then one may omit facts from a request without causing undue access.

Informally, in a safe language, undue access is impossible provided that requests tell no lies; whereas, in an unsafe language, the requests must additionally tell the whole truth. The choice between these is a function of trust in the program generating requests, comprehensiveness of analysis to generate requests, efficiency, and so on. Nevertheless, the ability to conclude, given a request that will yield access, that all requests with more information will also yield access, can potentially be a great boon to policy reasoning.

Some languages might choose to avoid the complications introduced by a policy testing for the absence of an attribute all together. In some contexts, such as certificate passing systems in which a certificate may be withheld, negated attributes may not make sense.[4] In such a context, requests would not list negated attributes and the policy would not test for the absence of an attributes at all.

4.3 Independent Composition

Consider the third interpretation of $\mathfrak{p}$. Under this interpretation, the meaning of $\mathfrak{p}$ can only be determined by looking at the interactions of the different sub-policies as a whole. Notice that any one of these sub-policies would produce a decision of not applicable in isolation, and yet together they interact to produce a permit decision. The third interpretation thus inhibits the easy use of local reasoning to reach conclusions about the policy as a whole. This increases the possibility of unintended results from combining sub-policies into a policy.

The alternative, as found in the first two interpretations, is for the sub-policies to be combined in such a way that only the result of each in isolation matters. This property is formalized as follows:

DEFINITION 4.4. *A policy combinator $\oplus$ of a language $\mathcal{L} = (P, Q, G, N, [\![\cdot]\!])$* **independently composes** *its sub-*

---

[4]One may argue that certificate passing systems may use negative certificates to achieve the checking of attribute absence. Whether this captures the notion of the absence of an attribute or just the presence of another related attribute is unclear. For example, one could conceivably hold both a positive and a negative certificate for an attribute.

policies iff

$$\forall p_1, p_2, \ldots, p_n \in P, \forall i,$$
$$1 \le i \le n \implies Q_{\oplus(p_1, p_2, \ldots, p_n)} \subseteq Q_{p_i} \quad (1)$$

and there exists a function $\boxplus : Q \to D^* \to D$ *such that*

$$\forall p_1, p_2, \ldots, p_n \in P, \forall q \in Q_{\oplus(p_1, p_2, \ldots, p_n)},$$
$$[\![\oplus(p_1, p_2, \ldots, p_n)]\!](q)$$
$$= \boxplus(q)([\![p_1]\!](q), [\![p_2]\!](q), \ldots, [\![p_n]\!](q)) \quad (2)$$

If all the combinators of $\mathcal{L}$ *independently compose, then* $\mathcal{L}$ *has the* **independent composition** *property.*

The first requirement forces a request defined for a policy to also be defined on each of its sub-policies. This is necessary for the second requirement to be well defined. The second requirement ensures that one can determine the decision of the whole policy from the request and decisions of its sub-policies on that request; no other properties of the sub-policies matter.

One might alternatively be tempted to define independent composition thus:

DEFINITION 4.5. *A policy combinator* $\oplus$ *of a language* $\mathcal{L} = (P, Q, G, N, [\![\cdot]\!])$ **semantically composes** *its sub-policies iff*

$$\exists \boxtimes : (Q \to D)^* \to (Q \to D), \forall p_1, p_2, \ldots, p_n \in P,$$
$$[\![\oplus(p_1, p_2, \ldots, p_n)]\!] = \boxtimes([\![p_1]\!], [\![p_2]\!], \ldots, [\![p_n]\!]) \quad (3)$$

If all the combinators of $\mathcal{L}$ *semantically compose, then* $\mathcal{L}$ *has the* **semantic composition** *property.*

Semantic composition ensures that all sub-policies with the same meaning in isolation will behave the same under the combinator. A language with the semantic composition property is arguably more clear than one without it, since only the isolated meaning of the sub-policy must known to reason about its use under the combinator.

THEOREM 4.6. *If a policy combinator* $\oplus$ *of an policy language* $\mathcal{L}$ *has independent composition, then it has semantic composition.*

PROOF. To prove that $\oplus$ has semantic composition, $\boxtimes : (Q \to D)^* \to (Q \to D)$ required for Equation 3 will be constructed from the $\boxplus : Q \to D^* \to D$ known to exist since $\oplus$ independently composes. Let

$$\boxtimes(f_1, f_2, \ldots, f_n) = \lambda q \,.\, \boxplus(q)(f_1(q), f_2(q), \ldots, f_n(q))$$

Then

$$\boxtimes([\![p_1]\!], [\![p_2]\!], \ldots, [\![p_n]\!])$$
$$= \lambda q \,.\, \boxplus(q)([\![p_1]\!](q), [\![p_2]\!](q), \ldots, [\![p_3]\!](q))$$
$$= \lambda q \,.\, [\![\oplus(p_1, p_2, \ldots, p_n)]\!](q)$$
$$= [\![\oplus(p_1, p_2, \ldots, p_n)]\!]$$

$\square$

THEOREM 4.7. *The semantic composition of a policy combinator does not imply that it independently composes.*

PROOF. Consider a rather odd language that has only one unary policy combinator, $\oplus$, atomic policies that are sets of

values, $G = \{\mathsf{permit}\}$, $N = \{\mathsf{deny}\}$, and requests that are sets of values. Let the semantics be

$$[\![\oplus(p_1)]\!] = \lambda q \,.\, \begin{cases} \mathsf{permit} & [\![p_1]\!](\{v'\}) = \mathsf{permit} \\ \mathsf{deny} & [\![p_1]\!](\{v'\}) = \mathsf{deny} \end{cases}$$

for some distinguished value v', and for atomic policies p, $[\![p]\!](q)$ equals permit iff $p \cap q \ne \emptyset$ and equals deny otherwise.

The language has semantic composition: for $\boxtimes$ such that

$$\boxtimes(f_1) = \begin{cases} \mathsf{permit} & f_1(\{v'\}) = \mathsf{permit} \\ \mathsf{deny} & f_1(\{v'\}) = \mathsf{deny} \end{cases}$$

clearly, $[\![\oplus(p_1)]\!] = \boxtimes([\![p_1]\!])$.

To show that the language does not have independent composition, assume that it does. Then there exists such a $\boxplus : Q \to D^* \to D$ to satisfy Equation 2. Let $p_1 = \{v\}$ and $p_2 = \{v, v'\}$ for some value v such that $v \ne v'$. Then,

$$\mathsf{deny} = [\![\oplus(\{v\})]\!](\{v\})$$
$$= \boxplus(\{v\})([\![\{v\}]\!](\{v\})) = \boxplus(\{v\})(\mathsf{permit})$$
$$= \boxplus(\{v\})([\![\{v, v'\}]\!](\{v\})) = [\![\oplus(\{v, v'\})]\!](\{v\}) = \mathsf{permit}$$

A contradiction is reached since $\mathsf{permit} \ne \mathsf{deny}$. $\square$

Only with independent composition can a policy reader with a specific request in mind know the decision of the whole policy from each of the component policies. This enables a reader to ask what-if questions like "What if Bob requests to write the log?" and determine the answer from recursively asking that question of the sub-policies. Such an ability is particularly helpful to readers interested in only a subset of the possible requests or already familiar with some of the sub-policies.

4.4 Monotonicity

As noted at the end of Section 3.3, the decision of deny complicates the policy combinators. One of the reasons for this is that, under combinators like Deny Overrides, a back-and-forth pattern can arise when considering the decision of the whole policy from the sub-policies. Consider each sub-policy in $\mathfrak{p}$ with the request $\mathfrak{q}_2$. Under a reasonable interpretation $\mathfrak{p}_1$ yields a decision of permit, $\mathfrak{p}_2$ a decision of deny, and $\mathfrak{p}_3$ not applicable. Thus, if the order of $\mathfrak{p}$ was changed to $\mathfrak{p}_3$, $\mathfrak{p}_1$, $\mathfrak{p}_2$ and we assume a Deny Overrides policy combinator, the apparent decision would go from non-granting to granting to non-granting.

Note that Permit Overrides does not exhibit this pattern since it is impossible to go from a granting decision to a non-granting one under it. Thus, the formalization of this pattern focuses on the transition from a granting to a non-granting decision.

DEFINITION 4.8. *A policy combinator* $\oplus$ *of a language* $\mathcal{L} = (P, Q, G, N, [\![\cdot]\!])$ *is* **monotonic** *iff*

$$\forall p_1, \ldots, p_n, p' \in P, \forall q \in Q',$$
$$[\![\oplus(p_1, \ldots, p_n)]\!](q) \le [\![\oplus(p_1, \ldots, p_i, p', p_{i+1}, \ldots, p_n)]\!](q)$$

where $Q' = Q_{\oplus(p_1, \ldots, p_n)} \cap Q_{\oplus(p_1, \ldots, p_i, p', p_{i+1}, \ldots, p_n)}$. *We say* $\mathcal{L}$ *is* **monotonic** *if every combinator is monotonic.*

Adding another sub-policy to a monotonic combinator cannot change the decision from granting to non-granting.

Having motivated and established these criteria, we now apply them to concrete access control languages.

5. CORE XACML

In its entirety, XACML [12] exceeds the bounds of the definitions given in Section 3. Full XACML includes obligations, which act as annotations on the decisions of permit and deny. These annotations specify actions that the system enforcing the access controls must preform before granting access or upon prohibiting access. Thus, an XACML policy may have effects beyond just granting or prohibiting access that the model presented fails to address.

Handling all of XACML is beyond the scope of this paper. For illustrative purposes, we employ a formalized subset of XACML, which we will call Core XACML (CXACML), which corresponds to the input of the tool Margrave [7, 8]. This subset is expressive enough to capture $RBAC_0$ [14].

5.1 Syntax

CXACML has two syntaxes: one for policies and one for requests. We present the policy syntax first, with the start non-terminal P. For syntactic brevity, we use a Lisp-like parenthetical syntax in place of XML notation.

```
P   ::=   (Policy C T P*) | (Rule T F)
C   ::=   FirstApp | DenyOver | PermitOver
T   ::=   ( (L*) (L*) (L*) (L*) )
L   ::=   (A+)
A   ::=   (id val)
F   ::=   Permit | Deny
```

Those policies formed by using solely the right choice of the production rule for P are called *rules*. XACML does not consider rules to be policies. However, since the semantics assigned to rules allows them to behave as policies, we will consider them policies. The elements of the syntax category T are called *targets*. The four parts of the target are the requirements placed on the subject, resource, action, and environment, respectively, for the policy to apply to a request. The non-terminals id and val are strings representing the attribute IDs and values.

EXAMPLE 5.1. *The following is a CXACML policy:*
```
(Policy FirstApp
        ((()) (((name log))) (()) (()))
        (Rule (((role dr)) (()) (()) (())) Deny)
        (Rule (((()) (()) (()) (())) Permit))
```
This policy permits all requests for access to a log except those made by doctors, which it denies. In detail, the policy is composed of two sub-policies using the combinator First Applicable and applies only to requests where the resource has the name of log. The first sub-policy denies requests where the subject has the role of dr regardless of the resource, action, or environment. The second permits all requests.

The syntax for requests is (with start non-terminal Q):

```
Q   ::=   ( (A*) (A*) (A*) (A*) )
```

They simply list the attributes possessed by the subject, resource, action, and environment in turn.

5.2 Semantics

In the following natural semantics, we will use the convention that a lower-case letter represents an element of the set or syntactic category represented by the upper-case equivalent. For example, P is the set of all policies and p is a policy. Let D be the set of all decisions ($D = \{\mathsf{permit}, \mathsf{deny}, \mathsf{na}\}$).

The core of the semantics of CXACML compares requests to targets. We will denote this relation by $q \Subset t$ for request q and target t. The natural semantics of Table 1 defines $\Subset$.

Next we define $\langle\langle \cdot \rangle\rangle$. Table 2 gives the result of evaluating rules. The following tables defines $\langle\langle \cdot \rangle\rangle$ over policies. Table 3 deals with two cases where a policy does not apply to a request. Finally, we must define the policy combinators: Permit Overrides in Table 4, Deny Overrides in Table 5, and First Applicable in Table 6.

5.3 Analysis

The syntax and semantics of CXACML defines $\mathcal{L}^{\mathsf{CXACML}} = (P, Q, G, N, \langle\langle \cdot \rangle\rangle)$. The syntax determines P and Q where the same set of requests is used for every policy (and thus, we treat Q as a set of requests). From the semantics, $G = \{\mathsf{permit}\}$, $N = \{\mathsf{na}, \mathsf{deny}\}$. CXACML allows for explicit denials and the checking of the implicit absence of attributes.

THEOREM 5.2. $\mathcal{L}^{\mathsf{CXACML}}$ *is deterministic.*

PROOF. Inspection of the inference rules for atomic policies (Table 2) shows that only one of them can hold at a time. Thus, atomic policies are deterministic.

Table 4 combined with Table 3 gives the semantics of the policy combinator Permit Overrides. The antecedents of all these inference rules are disjoint, that is, at most one them can hold for any policy and request. Thus, Permit Overrides is deterministic. The same argument holds for Deny Overrides and First Applicable using Tables 5 and 6. Thus, all the combinators are deterministic. □

Thus, one may view a CXACML policy as a function from requests to decisions with $[\![\cdot]\!]$ in place of $\langle\langle \cdot \rangle\rangle$. Further inspection establishes that $[\![\cdot]\!]$ is a total function.

For two requests $q = (s\ r\ a\ e)$ and $q' = (s'\ r'\ a'\ e')$, let $q \sqsubseteq_p q'$ (for every policy p) if $s \sqsubseteq' s'$, $r \sqsubseteq' r'$, $a \sqsubseteq' a'$, and $e \sqsubseteq' e'$ where $\sqsubseteq'$ is defined in Table 1.

THEOREM 5.3. *CXACML is not safe with respect to* $\sqsubseteq$.

PROOF. Consider the policy p shown in Example 5.1 and the requests $q = (()\ ((\mathtt{name\ log}))\ ()\ ())$ and $q' = (((\mathtt{role\ dr}))\ ((\mathtt{name\ log}))\ ()\ ())$. Clearly $q \sqsubseteq_p q'$. Yet $[\![p]\!](q) = \mathsf{permit} \not\leq \mathsf{deny} = [\![p]\!](q')$ □

THEOREM 5.4. $\mathcal{L}^{\mathsf{CXACML}}$ *has independent composition.*

PROOF. A combination algorithm c and target t together determine a policy combinator. For each pair of values for c and t, the needed function $\boxplus_c^t : Q \to D^* \to D$ exists to provide the meaning of the policy (Policy $c\ t\ p\ p^*$) and satisfy Equation 2. For Permit Overrides (when $c = \mathtt{PermitOver}$, or PO for short), the function $\boxplus_{\mathtt{PO}}^t(q)(d\ d^*)$ is equal to

na	if	$q \not\Subset t$
permit	else if	$d = \mathsf{permit} \vee \boxplus_{\mathtt{PO}}^t(q)(d^*) = \mathsf{permit}$
deny	else if	$d = \mathsf{deny} \vee \boxplus_{\mathtt{PO}}^t(q)(d^*) = \mathsf{deny}$
na	otherwise	

The function $\boxplus_{\mathtt{DO}}^t(q)(d\ d^*)$ for Deny Overrides is equal to

na	if	$q \not\Subset t$
deny	else if	$d = \mathsf{deny} \vee \boxplus_{\mathtt{DO}}^t(q)(d^*) = \mathsf{deny}$
permit	else if	$d = \mathsf{permit} \vee \boxplus_{\mathtt{DO}}^t(q)(d^*) = \mathsf{permit}$
na	otherwise	

For First Applicable, $\boxplus_{\mathtt{FA}}^t(q)(d\ d^*)$ is

$$\frac{(a_1^*)\in'(l_1^*) \qquad (a_2^*)\in'(l_2^*) \qquad (a_3^*)\in'(l_3^*) \qquad (a_4^*)\in'(l_4^*)}{((a_1^*)\ (a_2^*)\ (a_3^*)\ (a_4^*))\in((l_1^*)\ (l_2^*)\ (l_3^*)\ (l_4^*))}$$

$$\frac{\exists i\ \text{s.t.}\ l_i \sqsubseteq' (a^*)}{(a^*)\in'(l_1\ l_2 \ldots l_n)} \qquad\qquad \frac{\forall i\ \exists j\ \text{s.t.}\ a_i = a'_j}{(a_1\ a_2 \ldots a_n) \sqsubseteq' (a'_1\ a'_2 \ldots a'_m)}$$

Table 1: The Match Relationship

$$\frac{q \notin t}{q\langle\!\langle(\texttt{Rule } t\ f)\rangle\!\rangle\textsf{na}} \qquad\qquad \frac{q \in t}{q\langle\!\langle(\texttt{Rule } t\ \texttt{Permit})\rangle\!\rangle\textsf{permit}}$$

$$\frac{q \in t}{q\langle\!\langle(\texttt{Rule } t\ \texttt{Deny})\rangle\!\rangle\textsf{deny}}$$

Table 2: $\langle\!\langle\cdot\rangle\!\rangle$ on Rules

$$q\langle\!\langle(\texttt{Policy } c\ t)\rangle\!\rangle\textsf{na} \qquad\qquad \frac{q \notin t}{q\langle\!\langle(\texttt{Policy } c\ t\ p^*)\rangle\!\rangle\textsf{na}}$$

Table 3: Default na Inference Rules

$$\frac{q\in t \qquad \exists i\ \text{s.t.}\ q\langle\!\langle p_i\rangle\!\rangle\textsf{permit}}{q\langle\!\langle(\texttt{Policy PermitOver } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle\textsf{permit}}$$

$$\frac{q\in t \qquad \exists i\ \text{s.t.}\ q\langle\!\langle p_i\rangle\!\rangle\textsf{deny} \qquad \forall j, \neg(q\langle\!\langle p_j\rangle\!\rangle\textsf{permit})}{q\langle\!\langle(\texttt{Policy PermitOver } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle\textsf{deny}}$$

$$\frac{q\in t \qquad \forall i, q\langle\!\langle p_i\rangle\!\rangle\textsf{na}}{q\langle\!\langle(\texttt{Policy PermitOver } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle\textsf{na}}$$

Table 4: Inference Rules for Permit Overrides

$$\frac{q\in t \qquad \exists i\ \text{s.t.}\ q\langle\!\langle p_i\rangle\!\rangle\textsf{deny}}{q\langle\!\langle(\texttt{Policy DenyOver } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle\textsf{deny}}$$

$$\frac{q\in t \qquad \exists i\ \text{s.t.}\ q\langle\!\langle p_i\rangle\!\rangle\textsf{permit} \qquad \forall j, \neg(q\langle\!\langle p_j\rangle\!\rangle\textsf{deny})}{q\langle\!\langle(\texttt{Policy DenyOver } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle\textsf{permit}}$$

$$\frac{q\in t \qquad \forall i, q\langle\!\langle p_i\rangle\!\rangle\textsf{na}}{q\langle\!\langle(\texttt{Policy DenyOver } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle\textsf{na}}$$

Table 5: Inference Rules for Deny Overrides

$$\frac{q\in t \qquad q\langle\!\langle p_1\rangle\!\rangle\textsf{permit}}{q\langle\!\langle(\texttt{Policy FirstApp } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle\textsf{permit}}$$

$$\frac{q\in t \qquad q\langle\!\langle p_1\rangle\!\rangle\textsf{deny}}{q\langle\!\langle(\texttt{Policy FirstApp } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle\textsf{deny}}$$

$$\frac{q\in t \qquad q\langle\!\langle p_1\rangle\!\rangle\textsf{na} \qquad q\langle\!\langle(\texttt{Policy FirstApp } t\ p_2,\ldots,p_n)\rangle\!\rangle d}{q\langle\!\langle(\texttt{Policy FirstApp } t\ p_1\ p_2\ \ldots\ p_n)\rangle\!\rangle d}$$

Table 6: Inference Rules for First Applicable

na	if	$q \notin t$
d	else if	$d = \textsf{permit} \vee d = \textsf{deny}$
$\boxplus_{\texttt{FA}}^t(q)(d^*)$	otherwise	

where $\boxplus_{\texttt{PO}}^t(\circ) = \boxplus_{\texttt{DO}}^t(\circ) = \boxplus_{\texttt{FA}}^t(\circ) = \textsf{na}$ for the empty sequence $\circ$. □

THEOREM 5.5. $\mathcal{L}^{\textsf{CXACML}}$ *is not monotonic.*

PROOF. Consider the policy p in Example 5.1 and the policy p' that would be p without the first rule. Let the request q be $(((\texttt{role dr}))\ ((\texttt{name log}))\ ()\ ())$. $[\![p']\!](q) = \textsf{permit}$, but $[\![p]\!](q) = \textsf{deny}$. Thus, adding a rule to p' results in a request going from being granted to not being granted. □

6. ADAPTATIONS OF FIRST-ORDER LOGIC

Whereas XACML is an attempt to create a policy language from whole cloth, other languages are adaptations of first-order logic. Halpern and Weissman present several schemata for such languages [9]. Here we present and analyze the languages produced by two of their schemata, Lithium and $\mathcal{L}_5$.

For the ease presentation, first, we define the language schemata FOL, a more readily identifiable restriction of first-order logic. To ensure efficiency (and decidability!), the languages of $\mathcal{L}_5$ and Lithium use additional context-sensitive constraints to restrict FOL. We discuss these restrictions after giving a semantics to FOL. (The semantics of $\mathcal{L}_5$ and Lithium will be the same as that of FOL, restricted to subsets of the language.)

The schemata FOL is a restriction of many-sorted first-order logic. Each language of FOL corresponds to giving the logic a different vocabulary Φ (the parameters including quantifier symbols, predicate symbols, constant symbols, and function symbols). We assume that Φ includes the sorts S for subjects, R for resources, A for actions, and the predicate symbol Permit of the sort $S \times R \times A \to \{\textsf{T}, \textsf{F}\}$.[5] Φ may also include sorts to represent environmental data such as the current time and location.

6.1 Syntax of FOL

A *standard policy* under the vocabulary Φ is an expression with one of the following forms:
$(\forall_{y_1} x_1, \ldots, \forall_{y_m} x_m\ (\ell_1 \wedge \ldots \wedge \ell_n \to \texttt{Permit}(s, r, a)))$
$(\forall_{y_1} x_1, \ldots, \forall_{y_m} x_m\ (\ell_1 \wedge \ldots \wedge \ell_n \to \neg\texttt{Permit}(s, r, a)))$
where each x_i names a variable over the sort identified by y_i, s is a term over the sort S, r is a term over the sort R, a is a term over the sort A, and each ℓ_j is a literal over Φ that may include the variables $x_1, \ldots, x_m$.

The policies of the language FOL(Φ) are the standard policies under Φ and conjunctions of policies:

$$P ::= \textsf{StandardPolicy} \mid (\texttt{and } P^*)$$

EXAMPLE 6.1. *Let the vocabulary Φ' contain*

1. *the sorts $S = \{\texttt{amy}, \texttt{bob}, \texttt{joe}\}$,
   $R = \{\texttt{grades}, \texttt{courses}\}$, and $A = \{\texttt{assign}, \texttt{enroll}\}$;*

---

[5] Halpern and Weissman treat the Permit predicate as taking two arguments, a subject and a resource-action, instead of three.

2. the predicates $\texttt{Permit} : S \times R \times A \to \{\mathsf{T}, \mathsf{F}\}$, $\texttt{faculty} : S \to \{\mathsf{T}, \mathsf{F}\}$, and $\texttt{student} : S \to \{\mathsf{T}, \mathsf{F}\}$.

$\mathsf{FOL}(\Phi')$ *includes the following policy:*
```
(and (∀S x (faculty(x)→ Permit(x, grades, assign)))
     (∀S x (student(x)→ ¬Permit(x, grades, assign)))
     (∀S x (¬faculty(x)→ Permit(x, courses, enroll))))
```
where S *identifies the sort* S. *As the semantics will soon show, this policy has the same meaning as policy* $\mathfrak{p}$ *from Section 2 does under Interpretation 3.*

The requests of $\mathsf{FOL}(\Phi)$ have the form (s, r, a, e) where $s \in S$ is the subject making the request; $r \in R$ is the requested resource; $a \in A$ is the action the subject would like to preform on the resource; and e is a conjunction of ground literals and universal formulas of the form

$$\forall_{y_1} x_1, \dots, \forall_{y_m} x_m (\ell_1 \wedge \dots \wedge \ell_n \to \ell_{n+1})$$

where each x_i names a variable over the sort identified by y_i and each ℓ_i is a literal over Φ that may include the variables $x_l, \dots, x_m$. The expression e provides information about s, r, a, and the environment.

EXAMPLE 6.2. *The four-tuple*
```
(bob, courses, enroll,
        student(bob) ∧ faculty(amy) ∧¬ student(amy))
```
is a request of $\mathsf{FOL}(\Phi')$ *where* Φ' *is defined in Example 6.1.*

6.2 Semantics of FOL

The semantics of a policy follows from interpreting it as a formula in many-sorted first-order logic. The policy combinator $\texttt{and}$ becomes conjunction. The standard policies and e are interpreted as the corresponding logic formulas. A policy p defines a relation $\langle\!\langle p \rangle\!\rangle$ between requests and $\{\mathsf{permit}, \mathsf{deny}\}$ as follows:

$$(s, r, a, e) \langle\!\langle p \rangle\!\rangle \mathsf{permit} \quad \text{iff} \quad p \wedge e \vdash \texttt{Permit}(s, r, a)$$
$$(s, r, a, e) \langle\!\langle p \rangle\!\rangle \mathsf{deny} \quad \text{iff} \quad p \wedge e \vdash \neg\texttt{Permit}(s, r, a)$$

where $\vdash$ is interpreted as the standard "proves" relation for many-sorted first-order logic over Φ.

To define a deterministic and total version of FOL, we expand the set of decisions to $D = \{\mathsf{na}, \mathsf{permit}, \mathsf{deny}, \mathsf{error}\}$ and define $[\![p]\!]((s, r, a, e))$ to be

error if $p \wedge e \vdash \texttt{Permit}(s, r, a)$ and $p \wedge e \vdash \neg\texttt{Permit}(s, r, a)$
permit if $p \wedge e \vdash \texttt{Permit}(s, r, a)$ and $p \wedge e \nvdash \neg\texttt{Permit}(s, r, a)$
deny if $p \wedge e \nvdash \texttt{Permit}(s, r, a)$ and $p \wedge e \vdash \neg\texttt{Permit}(s, r, a)$
na if $p \wedge e \nvdash \texttt{Permit}(s, r, a)$ and $p \wedge e \nvdash \neg\texttt{Permit}(s, r, a)$

Since a policy composed of sub-policies, each composed of standard policies, is semantically equivalent to a policy composed of all the standard policies without the intermediate sub-policies, we will henceforth treat all policies as either a standard policy or a conjunction of standard policies.

6.3 Analysis of FOL

The language $\mathsf{FOL}(\Phi)$ defines the deterministic and total policy language $(P, Q, G, N, [\![\cdot]\!])$. The syntax determines P and Q where Q may be treated as a set of requests since for all policies p and p', $Q_p = Q_{p'}$. The semantics requires that $G = \{\mathsf{permit}\}$ and $N = \{\mathsf{na}, \mathsf{deny}, \mathsf{error}\}$. The languages of FOL has the policy combinator $\texttt{and}$. FOL allows for explicit denials and checking for the explicit absence of attributes.

Given two requests $q = (s, r, a, e)$ and $q' = (s', r', a', e')$, if $s \neq s'$, $r \neq r'$, or $a \neq a'$, we consider the two requests incomparable. If $s = s'$, $r = r'$, and $a = a'$, then we would like $\sqsubseteq$ to order requests according to their information content. One might conclude that $q \sqsubseteq_p q'$ if $e' \implies e$. However, suppose $e' \implies \bot$, where $\bot$ is logical contradiction. Then e' contains no information and yet it implies e. Similarly, if $p \wedge e' \implies \bot$, then e' contains no information with respect to p. Thus, we define $\sqsubseteq_p$ as follows:
Let $(s, r, a, e) \sqsubseteq_p (s', r', a', e')$ iff

1. $s = s'$, $r = r'$, and $a = a'$; and

2. $p \wedge e'$ implies $p \wedge e$ but not $\bot$, or $p \wedge e$ implies $\bot$.

THEOREM 6.3. $\mathsf{FOL}(\Phi)$ *is safe with respect to* $\sqsubseteq$ *for any vocabulary* Φ.

PROOF. Assume $\mathsf{FOL}(\Phi)$ is not safe. Then there must exist $p \in P$ and $q, q' \in Q$ such that $q \sqsubseteq_p q'$ and $[\![p]\!](q) \not\trianglelefteq [\![p]\!](q')$. Let $q = (s, r, a, e)$ and $q' = (s, r, a, e')$. Since permit is the only granting decision, $[\![p]\!](q) = \mathsf{permit}$ and thus $p \wedge e \vdash \texttt{Permit}(s, r, a)$. Since $N = \{\mathsf{na}, \mathsf{deny}, \mathsf{error}\}$, $[\![p]\!](q')$ must be either na, deny, or error.

Since $q \sqsubseteq_p q'$, two cases arise:

1. $p \wedge e'$ implies $p \wedge e$ but not $\bot$: Since $p \wedge e' \implies p \wedge e$ and $p \wedge e \vdash \texttt{Permit}(s, r, a)$, $p \wedge e' \vdash \texttt{Permit}(s, r, a)$. Thus, $[\![p]\!](q')$ is either permit or error. However, if $[\![p]\!](q') = \mathsf{error}$, then $p \wedge e' \implies \bot$, a contradiction. Furthermore, $[\![p]\!](q') = \mathsf{permit} \notin N$ is also a contradiction.

2. $p \wedge e$ implies $\bot$: In this case, $[\![p]\!](q) = \mathsf{error} \neq \mathsf{permit}$, a contradiction.

We can thus conclude that $\mathsf{FOL}(\Phi)$ must be safe. $\square$

THEOREM 6.4. $\mathsf{FOL}(\Phi)$ *does not have independent composition for some* Φ.

PROOF. Consider the policy p_{a}:
```
(and (∀S x, student(x)→ Permit(x, log, read))
     (∀S x, ¬student(x)→ Permit(x, log, read)))
```
the policy p_{b}:
```
(and (∀S x, student(x)→ Permit(x, log, edit))
     (∀S x, ¬student(x)→ Permit(x, log, edit)))
```
and request $q = (\texttt{bob}, \texttt{log}, \texttt{read}, \mathsf{T})$. On q, p_{a} produces the decision of permit while its sub-policies yield na. However, p_{b} produces the decision of na while its sub-policies also yield na on q. Thus, the required function $\boxplus_{\mathsf{and}}$ would have to satisfy

$$\mathsf{permit} = \boxplus_{\mathsf{and}}(q)(\mathsf{na}\ \mathsf{na}) = \mathsf{na}$$

A contradiction, and hence $\boxplus_{\mathsf{and}}$ cannot exist. $\square$

THEOREM 6.5. $\mathsf{FOL}(\Phi)$ *is not monotonic for some* Φ.

PROOF. Consider the policy p_{c}:
```
(and (∀S x, student(x)→ Permit(x, log, read))
     (∀S x, student(x)→ ¬Permit(x, log, read)))
```
with and without the second sub-policy, and the request $(\texttt{bob}, \texttt{log}, \texttt{read}, \texttt{student(bob)})$. In the absence of the second sub-policy, the decision is permit, whereas p_{c} produces error. $\square$

6.4 Analysis of Lithium

Halpern and Weissman restrict FOL to create the language they dub Lithium.[6] A slightly modified form follows.

Lithium relies heavily on the notion of "bipolarity". A literal ℓ of f is labeled *bipolar* in f relative to the equality statements in e if the following holds: there exists a term $\neg\ell'$ in f and variable substitutions σ and σ' such that it follows from e that $\ell\sigma = \ell'\sigma'$.

Lithium also makes use of the notion of equality-safety. (p, e_0, e_1) is *equality-safe* if

1. $e_1 \wedge p$ when written in CNF (*i.e.*, of the form $c_1 \wedge \ldots \wedge c_n$ where each c_j has the form $\forall_{y_1} x_1, \ldots, \forall_{y_m} x_m(\phi)$ where ϕ is a qualifier-free disjunction of literals) has no clause with a disjunct of the form $t = t'$, and

2. it is not the case that $f_0 \vdash t = t'$ where f_0 is the conjunction of the equality statements in e_0,

where t and t' are closed terms such that (1) they both appear in e_0; and (2) either t is a sub-term of t', or both t and t' mention function symbols.

Like FOL, Lithium is a set of languages each with a different vocabulary. Let $\mathsf{Li}(\Phi)$ be the instance of Lithium using the vocabulary Φ. $\mathsf{Li}(\Phi)$ has the same set of policies as $\mathsf{FOL}(\Phi)$. However, each policy p of $\mathsf{Li}(\Phi)$ has a different set of requests for which it is defined (a different value for Q_p). A request $(s, r, a, e_0 \wedge e_1)$ of $\mathsf{FOL}(\Phi)$ is in Q_p iff:

1. e_0 is a *basic environment* (a conjunction of ground terms),

2. e_1 is a conjunction of universally quantified formulas,

3. (p, e_0, e_1) is equality-safe, and

4. every conjunct of $e_1 \wedge p$ has at most one literal that is bipolar in $e_1 \wedge p$ relative to the equality statements in e_0.

Lithium is safe since its requests are a subset of those of FOL.

THEOREM 6.6. *Lithium does not have independent composition for some Φ.*

PROOF. Consider the policies p_a and p_b and the request given with them in the proof of Theorem 6.4. The request is in Q_{p_a}. To show this, we check that the request satisfies all four of the requirements for a request to be in Q_{p_a} given above. Since $e_0 \wedge e_1 = \mathsf{T}$, the first three requirements hold. The last requirement holds since $\mathtt{student}(x)$ and $\neg\mathtt{student}(x)$ are the only bipolars and they are each in a different conjunct.

By similar reasoning, the request is also in Q_{p_b}. Thus, the proof follows as before. □

6.5 Analysis of $\mathcal{L}_5$

In their work, Halpern and Weissman define a further restriction of FOL, which they call $\mathcal{L}_5$.

Like Lithium, $\mathcal{L}_5(\Phi)$ includes all the policies of $\mathsf{FOL}(\Phi)$ with each policy having a different set of requests for which it is defined. For a policy p of $\mathcal{L}_5(\Phi)$, Q_p consists of all the requests (s, r, a, e) of $\mathsf{FOL}(\Phi)$ such that:

[6]The name Lithium only appears in the 2006 version of their work [9].

1. e is a basic environment,

2. equality is not used in e or p,

3. for every atomic policy p' in p, all variables appearing in p' appears as an argument to $\mathtt{Permit}$ in p', and

4. there are no *bipolars* in p relative to the empty set of equality statements.

As with Lithium, $\mathcal{L}_5(\Phi)$ is safe since it is a subset of a safe language.

Halpern and Weissman have proven the following theorem (Proposition 4.2 in their updated document [9]):

THEOREM 6.7. *Let p be a compound policy and (s, r, a, e) be a request of $\mathcal{L}_5$. Then $e \wedge p \vdash \mathtt{Permit}(s, r, a)$ iff there is a sub-policy p' of p such that $e \wedge p' \vdash \mathtt{Permit}(s, r, a)$.*

Using the same approach as given in their proof, one can generalize this proof to include statements of the form $e \wedge p \vdash \neg\mathtt{Permit}(s, r, a)$ also.

THEOREM 6.8. $\mathcal{L}_5(\Phi)$ *has independent composition for all* Φ.

PROOF. Allowing p_i to range over the sub-policies of p, the above result yields:

$$\llbracket p \rrbracket(q) = \begin{cases} \mathsf{error} & \exists i, j, \llbracket p_i \rrbracket(q) = \mathsf{permit} \wedge \llbracket p_j \rrbracket(q) = \mathsf{deny} \\ \mathsf{permit} & \text{else if } \exists i, \llbracket p_i \rrbracket(q) = \mathsf{permit} \\ \mathsf{deny} & \text{else if } \exists i, \llbracket p_i \rrbracket(q) = \mathsf{deny} \\ \mathsf{deny} & \text{otherwise} \end{cases}$$

From this, it is easy to construct an appropriate value for $\boxplus_{\mathsf{and}}(q)(d_1, d_2, \ldots, d_n)$:

error	if	$\exists i, j, d_i = \mathsf{permit} \wedge d_j = \mathsf{deny}$
permit	else if	$\exists i, d_i = \mathsf{permit}$
deny	else if	$\exists i, d_i = \mathsf{deny}$
deny	otherwise	

Notice that $\boxplus_{\mathsf{and}}$ does not use the value of q: it merely composes the results from its sub-policies. □

A policy author concerned solely with expressive power would select Lithium over $\mathcal{L}_5$. However, the choice becomes more complicated when concerned about the ability to reason about policies, because only $\mathcal{L}_5$ features independent composition. We hope that elucidating this trade-off with a combination of proof and illustrative examples, as we have done above, will help authors choose better between the policy languages they use, even when the languages are within the same family.

7. RELATED WORK

De Capitani di Vimercati *et al.* discuss explicit denial and how it introduces the need for policy combinators that reduce the clarity of the language [4]. The authors list various policy combinators that are possible, many of which are more complex than those we present. The paper includes discussion of a few policy languages, including XACML and a language grounded in first-order logic. The paper does not, however, attempt to systemically compare them.

The work of Mark Evered and Serge Bögeholz concerns the quality of a policy language [5]. After conducting a case study of the access-control requirements of a health

information system, they proposed a list of criteria for policy languages. They state that languages should be concise, clear, aspect-oriented (*i.e.*, separate from the application code), fundamental (*i.e.*, integrated with the middleware, not an *ad hoc* addition), positive (*i.e.*, lists what is allowed, not what is prohibited), supportive of needs-to-know, and efficient. Although they compare four languages based on these criteria, they do not formalize the criteria.

Some authors have considered formal treatments of programming language expressiveness [6, 11]. Felleisen's is the closest in spirit to ours. His framework examines the ability to translate the features of one language in the other with only local transformations. That work does not, however, directly address reasoning.

8. DISCUSSION

This paper presents our analysis framework and its findings. Some differences between languages lie in the realms of decision sets, policy combinators, and checking for the absence of attributes, but these are clear from the language definitions. Our framework highlights the following more subtle, *semantic* differences:

Independence Core XACML and $\mathcal{L}_5$ feature independent composition of polices into compound policies, and thus allow for reasoning about a policy by reasoning about the sub-policies separately. Lithium, in contrast, does not exhibit this property, and therefore potentially requires reasoning about a policy all at once.

Safety $\mathcal{L}_5$ and Lithium provide safety for the most natural definition of the "contains more information" ordering. Core XACML, in contrast, does not, which implies that information missing from a Core XACML request could result in unintended access being granted.

These differences are not orthogonal. Clearly, the combinators selected determine whether the language will have independent composition. Furthermore, implicit checking of attributes will result in the loss of safety.

These properties may guide policy language designers. For example, suppose a designer wishes to create a safe variant of XACML. One way to achieve this would be to eliminate rules that deny access and thus the decision of deny. (We provide further details in an extended version of this work [15].)

As noted in Section 5, the comparison framework must be generalized to treat language with more exotic constructs like obligations. More importantly, we need to perform user studies to determine whether, and how well, our properties correlate with policy comprehension by humans. Lastly, this framework should be coupled with one for measuring the expressive power of a policy language before fair judgment may be passed on languages.

9. ACKNOWLEDGMENTS

10. REFERENCES

[1] A. Anderson. Core and hierarchical role based access control (RBAC) profile of XACML, version 2.0. Technical report, OASIS, Sept. 2004.

[2] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations Workshop*, pages 139–154, 2004.

[3] E. Bertino, P. Samarati, and S. Jajodia. Authorizations in relational database management systems. In *ACM Conference on Computer and Communications Security*, pages 130–139, 1993.

[4] S. De Capitani di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control. In *Databases in Networked Information Systems: 4th International Workshop*, volume 3433 of *Lecture Notes in Computer Science*. Springer-Verlag, Mar. 2005.

[5] M. Evered and S. Bögeholz. A case study in access control requirements for a health information system. In *Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 53–61, 2004.

[6] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[7] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *International Conference on Software Engineering*, pages 196–205, May 2005.

[8] M. M. Greenberg, C. Marks, L. A. Meyerovich, and M. C. Tschantz. The soundness and completeness of Margrave with respect to a subset of XACML. Technical Report CS-05-05, Department of Computer Science, Brown University, Apr. 2005.

[9] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *IEEE Computer Security Foundations Workshop*, pages 187–201, 2003. Updated 2006 version available at http://www.citebase.org/cgi-bin/citations?id=oai:arXiv.org:cs/0601034.

[10] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *ACM SIGMOD International Conference on Management of Data*, pages 474–485, 1997.

[11] J. C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 212:141–163, 1993.

[12] OASIS. eXtensible Access Control Markup Language (XACML) version 2.0. OASIS Standard, Feb. 2006.

[13] C. Powers and M. Schunter. Enterprise privacy authorization language (EPAL 1.2). W3C Member Submission, Nov. 2003.

[14] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[15] M. C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages with extended XACML analysis. Technical Report CS-06-04, Department of Computer Science, Brown University, Apr. 2006.