A Simple and Expressive Semantic Framework for Policy Composition in Access Control

Glenn Bruns
Bell Labs
grb@bell-labs.com

Daniel S. Dantas
Princeton University
ddantas@cs.princeton.edu

Michael Huth
Imperial College London
mrh@doc.ic.ac.uk

ABSTRACT

In defining large, complex access control policies, one would like to compose sub-policies, perhaps authored by different organizations, into a single global policy. Existing policy composition approaches tend to be ad-hoc, and do not explain whether too many or too few policy combinators have been defined. We define an access control policy as a *four-valued* predicate that maps accesses to either *grant*, *deny*, *conflict*, or *unspecified*. These correspond to the four elements of the Belnap bilattice. Functions on this bilattice are then extended to policies to serve as policy combinators. We argue that this approach provides a simple and natural semantic framework for policy composition, with a minimal but functionally complete set of policy combinators. We define derived, higher-level operators that are convenient for the specification of access control policies, and enable the decoupling of conflict resolution from policy composition. Finally, we propose a basic query language and show that it can reduce important analyses (e.g. conflict analysis) to checks of policy refinement.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Control*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Languages, Theory, Security

Keywords

Access-control policy languages, Bilattices, Multi-valued logic

1. INTRODUCTION

An access control policy is essentially a predicate that determines whether an access – a requested operation on some object by some subject – should be permitted or not. In practice, however, there are many complications. The problem we focus on here is how a complex access control policy can be concisely specified. Since an access control policy is merely a predicate, one could list all the accesses that are permitted. However, there are many subjects and objects, and the sets of subjects and objects often change.

One feature of access control policy languages that addresses this problem is the ability to refer to *collections* of subjects and objects. The best-known example of this approach is *role-based access control* [9, 10]. More generally, classes of subjects and objects can be set in a hierarchy, allowing more general or more specific policy statements. For example, an access policy for a hospital may refer to roles such as "physician" and "cardiologist". We might expect cardiologists to inherit the permissions given to physicians.

Another useful policy language feature is high-level *policy combinators*, so that policies authored by sub-organizations can be combined easily into a single organization-wide policy. For example, a group of libraries may want to have a common access control policy, and to do this they might want to merge their individual policies. A natural concept is to permit in the combined policy any access permitted in some sub-policies. Or, in some circumstances, it may be better to permit in the combined policy only what is permitted in every sub-policy.

However, real problems arise when attempting to define policy combinators on policies that are simply predicates on accesses. The first problem is that it is impossible to indicate within a policy the difference between denying some access, and simply not caring about it. For example, a policy may regard an access as outside its scope, or may not have a position as to whether the access should be denied or permitted. This matter is explained clearly by Halpern and Weissman (see [16], Section 6). They give as an example two libraries that want to merge their policies. Suppose the policy of the first library states that users may access the coatroom, but the policy of the second library does not state that users may access the coatroom. If the second library's policy was intended to deny access to the coatroom, then the two policies are in conflict. However, if the second library's policy simply wanted to be silent on the subject of coatrooms, perhaps because the second library has no coatroom, then the policies are not in conflict.

FMSE'07, November 2, 2007, Fairfax, Virginia, USA

A solution to this problem is to allow a policy to independently state positive and negative permissions. Thus, the second library, if desired, could explicitly state that users may not access the coatroom. Many languages for access control policies support this feature (e.g., [17, 22, 8]). With this feature present, a policy is naturally interpreted as a mapping from accesses to one of three values: *true* (or *grant*), *false* (or *deny*), and $\perp$ (or *unspecified*). In the end, a reference monitor responsible for enforcing a policy will have to make a grant or deny decision on every access, but for the purpose of defining policies, the value *unspecified* is useful. By not denying an access by default when no explicit rule for granting is found, it is then possible to check whether a policy has *gaps* – accesses on which it is not defined.

Problems in policy composition still occur, however, even once both positive and negative permissions are treated separately. There is the possibility that two policies will *conflict* on an access: one will grant the access and the other will deny it. A common solution to this problem is to define all policy composition operators so that conflicts are immediately resolved, so that the composed policy is guaranteed to have no conflicts. For example, in logical specifications of access control policies this is appealing because conflicts may represent logical contradictions that will cause the policy specification to become entirely inconsistent, even for accesses not related to the particular cause of the conflict.

However, there are good reasons for not insisting that potential conflicts be eliminated when policies are composed. First, for analysis purposes it can be helpful for conflicts to become explicit so that their causes can be discovered and resolved in an intelligent manner. By "automatically" resolving every conflict at the moment of composition, it may be that a poor choice of resolution is made. On the other hand, it may be that a uniform method is desired for resolving conflicts. In this case, it would make sense to have a single point at which conflicts are resolved, and not have to define every policy composition operator so that it resolves conflicts in the same way. In short, a policy language should allow the issues of policy composition and conflict resolution to be treated separately.

Our approach is to treat a policy as a four-valued predicate. That is, a policy maps every requested access to either *grant*, *deny*, *unspecified*, or *conflict*. The *unspecified* response to an access allows a policy to indicate that an access is outside the policy's scope, or that a policy does not take a stance towards the access. The *conflict* response naturally arises when composing two policies that take opposite stances towards an access.

These four values we propose for policy responses are exactly the four truth values of Belnap's four-valued logic [5]. This logic has been studied extensively [11, 14, 3, 2], and it is no coincidence that a main application has been to the semantics of logic programs, where handling negation under a closed-world assumption has been a difficulty [12]. Belnap's logic is based on supplying two orderings on the truth values; one that captures degree of truth and another that captures degree of information. These ideas also apply very naturally to policies, where we might ask whether a policy grants accesses more than another policy, and whether a policy is defined on more accesses than another policy.

Our approach to policy composition is to derive policy operators from the operators of Belnap's logic. Thus, we interpret expressions of this logic not as truth values, but as policies. Similarly, we derive relations on policies from the two (truth and knowledge) orderings of Belnap's logic.

Our framework for policy composition and analysis is comprised of three layers of language:

1. A core language for policies and their composition, essentially a pointwise extension of logical operators of the Belnap space to accesses.
2. Syntactic sugar over the core language, capturing important idioms of policy composition.
3. Finally, a propositional query language in which policy analyses can be expressed.

We strongly believe that this framework, by explicitly supporting *unspecified* and *conflict* values of policies, and by clearly and separately capturing the two notions of truth and knowledge, greatly improves on existing semantic approaches to the treatment of policies and their composition. Additional benefits are gained by building on an established logic with well-understood properties.

Organization. In Section 2 we briefly list the basic elements of access control. In Section 3 we motivate the Belnap bilattice and give its formal definition. In Section 4 we describe a four-valued semantics for policies and show how a language for policy composition can be built on the operators of the Belnap bilattice. In Section 5 we list some problems in defining and composing access control policies, and show how the problems are solved using our approach to policy composition. In Section 6 we define truth and information orderings on policies, and then define a query language in which analysis questions on policies can be framed. In Section 7 we specialize and extend our language for the case in which accesses are based on roles, in the sense of role-based access control. In the last sections of the paper we discuss related work and make concluding remarks.

2. ACCESS CONTROL

A basic access control mechanism is a predicate on accesses, where an access is commonly treated as a triple of the form $(subject, operation, object)$, meaning that *subject* (e.g. some user, role, or program) is requesting to perform the *operation* on *object* (e.g. a file on a computer system).

In this paper, the precise structure of accesses is not important, so we simply imagine a domain A of accesses. In fact, it is important not to commit to a specific access control model to highlight the wide range of applicability of our semantic framework. We follow the standard approach of treating a basic access control mechanism semantically as a predicate of type $A \times Context \to \mathbf{Bool}$ where $\mathbf{Bool} = \{true, false\}$. Here $Context$ represents some additional information an access control mechanism might need in making decisions. We give some examples for the use of such a context for the purpose of illustration:

- Accesses may be permitted or denied according to the time of the access. In this case the elements of domain $Context$ may be time stamps.
- An access (s, op, o) may be permitted only if it carries a certificate saying that subject s inherits rights pertaining to operation op on object o from subject s'. In this case $Context$ contains delegation certificates.
- In *history-based* access control, an access may be permitted or denied based on properties of the record of past accesses. In this case the elements of $Context$ are sequences of past access decisions which may have to manage information flow [4].

For the purposes of this paper, we may think of $A \times Context$ abstractly as A, so we will no longer use the richer type. Also, we do not consider here the structure of principals, as in [1], nor do we look at issues of authentication, trust models, identity management, or higher-order policies. Some of these issues are orthogonal to our approach and contributions; others will be expanded on in future work.

3. BELNAP'S FOUR-VALUED LOGIC

We use the Belnap bilattice as the basis for policy composition. Rather than starting with a definition of the Belnap bilattice, we show how this structure arises naturally from issues in access control.

First consider the issue of composing access control policies. Imagine two access control policies, each of which reports a boolean representing whether an access should be permitted or not (or "abstains" if the policy is unspecified on that access). We can form four possible sets by collecting the verdicts: $\{true\}$, $\{true, false\}$, $\{false\}$, and $\{\}$. For example, the set $\{true, false\}$ arises when one policy permits the access and the other denies it. The set $\{\}$ arises when the access is outside the domain of both policies.

We can order these sets in two different ways. If we order by the degree to which the access is permitted, we have that $\{true\}$ is greatest and $\{false\}$ is least. Fig. 1 left) depicts a Hasse diagram containing the four sets. We can also order by the amount of information we have obtained. In this ordering $\{true, false\}$ is greatest and $\{\}$ is least. We write $\leq_t$ for the truth ordering and $\leq_k$ for the information (i.e. knowledge) ordering. The two orderings are shown along the two axes in Fig. 1.

A negation operation on these sets, written $\neg$, can be derived by applying standard logical negation to each element of the set and collecting the results. For example, $\neg\{true\} = \{false\}$ and $\neg\{true, false\} = \{true, false\}$.

Next consider the issue of access control policies containing both "permit rules" and "deny rules". A permit rule returns *true* if the access is permitted and *false* if the access is not permitted (but not necessarily denied). Deny rules work symmetrically. Now suppose we have an access for which both a permit rule and a deny rule are applicable. We can summarize by building a pair (a, b) out of the results from the two rules; e.g. $(true, false)$ arises when the permit rule says "permit" and the deny rule does not say "deny".

These pairs can similarly be ordered in two different ways. If we order by the degree to which the access is permitted, then $(true, false)$ is greatest and $(false, true)$ is least – see Fig. 1 right). If we order by the amount of information, we have that $(true, true)$ is greatest and $(false, false)$ is least.

In the two cases we have formed isomorphic ordered sets of four values. Indeed, two orderings are present in each case. Thus, when combining the outcome of multiple policies, or even when combining the effects of multiple rules within a single policy, it can be natural to regard a policy as producing one of four possible results for an access.

This structure of four elements with two orderings was developed by Belnap as the basis for a four-valued logic [5]. It is an example of the more general notion of *bilattice*, defined by Ginsburg [15]. A bilattice consists of a set of elements with two orderings and a negation operator, such that both orderings form lattices and the negation operator interacts with the two orderings in a particular way.

Definition 1 *A* bilattice *is a structure $(A, \leq_t, \leq_k, \neg)$, where A is a non-empty set, and $\leq_t$ and $\leq_k$ are partial orders on A such that $(A, \leq_t)$ and $(A, \leq_k)$ are complete lattices, $\neg$ maps from A to A, and these conditions must hold:*

- $x \leq_t y \Rightarrow \neg y \leq_t \neg x$,
- $x \leq_k y \Rightarrow \neg x \leq_k \neg y$, *and*
- $\neg\neg x = x$.

(The form of this definition of bilattice comes from [14].) The first two conditions say that negation inverts truth, but does not affect knowledge.

The Belnap bilattice $(\mathbf{Four}, \leq_t, \leq_k, \neg)$ is the simplest nontrivial bilattice, where $\mathbf{Four} = \{true, false, \bot, \top\}$. It is shown in Fig. 1 bottom). We sometimes refer to **Four** as the "Belnap space". We write $\wedge$ and $\vee$ for the meet and join operations of the lattice formed by the truth ordering $\leq_t$, and we write $\otimes$ and $\oplus$ for the meet and join operations of the lattice formed by the information ordering $\leq_k$. It is common to also define a negation operator relative to the information ordering. It is written $-$ and called "conflation". As expected it inverts knowledge, but does not affect truth:

$$x \leq_k y \Rightarrow -y \leq_k -x \qquad \text{and} \qquad x \leq_k y \Rightarrow -x \leq_t -y$$

The Belnap bilattice can be used as the basis for a four-valued logic. The values *true* and *false* capture the standard logical notions of truth and falsity. The value $\bot$ means "no information", and the value $\top$ means "conflicting information" or "too much information".

The meet and join operators of **Four** can be understood as logical operators. Think of conjunction and disjunction in ordinary two-valued logic as the meet and join operators in a lattice of only two values, with *true* as the top element and *false* as the bottom element. Then it is clear that one can generalize two-valued to many valued logics by using a lattice of truth values. In particular, one can form a four-valued logic from the Belnap bilattice, interpreting conjunction as $\wedge$, disjunction as $\vee$, and negation as $\neg$. One can then develop other logical concepts, like logical consequence, in this four-valued setting (see [5, 3]).

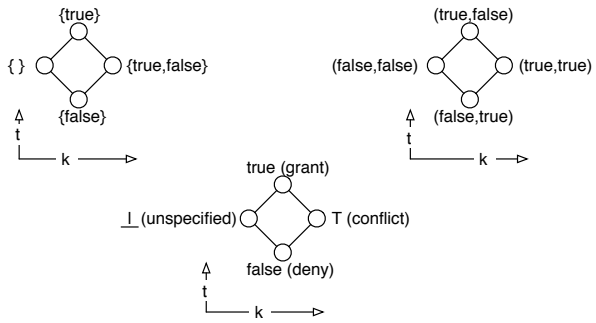

Figure 1: The Belnap bilattice: obtained left) by a subset construction; right) by a lattice product construction; and bottom) in its standard presentation (with synonyms for access control decisions in parentheses)

Also, one can define an implication operator $\supset$ by $a \supset b = b$ if $a \in \{true, \top\}$, and $a \supset b = true$ otherwise. This operator is one possible generalization of classical implication.

The Belnap lattice has been used in Artificial Intelligence, along with other non-monotonic logics, to capture human reasoning processes [15]. It also has been successfully applied to logic programming, e.g. it helped there in the presence of negation or multiple truth values [12].

We briefly note some algebraic properties of the Belnap bilattice. First, for each lattice taken separately we have that the meet and join operations are commutative and associative, and we have absorption (e.g., in the truth ordering, $a \vee (a \wedge b) = a$ and $a \wedge (a \vee b) = a$), and idempotency (e.g., in the truth ordering, $a \wedge a = a$ and $a \vee a = a$). Also, each meet and join operation is monotonic in its respective ordering. Because the Belnap bilattice is distributive, we have for each ordering that meet distributes over join and vice versa. The Belnap bilattice is an *interlaced* bilattice: $\wedge$ and $\vee$ are monotonic with respect to $\leq_k$, and $\otimes$ and $\oplus$ are monotonic with respect to $\leq_t$.

4. POLICIES

The idea of this paper is to understand an access control policy not as a simple predicate, but as a four-valued one of the form $A \rightarrow$ **Four**. We use the synonyms g ("grant", for *true*), d ("deny", for *false*), u ("unspecified" for $\bot$), and c ("conflict", for $\top$).

The intuition behind the g and d values should be clear. When a policy returns u for an access, it might be that the policy author considers the access outside the scope of the policy, or it might be that the policy author has no opinion as to how the access is treated. When a policy returns c, the idea is that there is conflict within the policy as to whether the access should be granted or denied.

We obtain policy operators as the pointwise extension of operators on the Belnap space. Letting p and q be policies, and $\circ$: **Four** $\times$ **Four** $\rightarrow$ **Four** be a binary operator on the Belnap space, the derived policy operator is then:

$$(p \circ q)(a) \stackrel{\text{def}}{=} p(a) \circ q(a)$$

The type of $p \circ q$, like p and q, is $A \rightarrow$ **Four**. Operators with more or less than two operators can be handled similarly. In this way the Belnap operators can serve as policy operators. The algebraic properties of the operators when interpreted on **Four** are retained when the operators are interpreted on policies. For example, we have $p \otimes q = q \otimes p$.

p, q	::=	*Policy*
x if ap		Basic policy
$\neg p$		Logical negation
$p \wedge q$		Logical meet
$p \supset q$		Implication

Figure 2: Syntax of the Policy Language.

4.1 A Basic Policy Language

The Belnap operators provide a language of policy composition, but for a complete policy language we also need a way to write basic policies. A basic policy expression shows, for various sets of accesses, whether the accesses within the sets are to be denied or permitted. Policy languages often contain rich sub-languages for defining basic predicates on accesses, including use of the access context. We do not want to define yet another such language here. We will therefore assume a sub-language of boolean access predicates, and allow policies to be defined in terms of these access predicates.

Fig. 2 shows the abstract syntax of our policy language, where x ranges over $\{g, d\}$, and ap is an expression in the sub-language of boolean access predicates, of type $A \rightarrow$ **Bool**. Next we consider the meaning of policy expressions. For the sub-language of predicate expressions, we assume only a semantic function $[\![.]\!]$ that maps each such expression to an access predicate. Then, a policy expression maps an access to an element of the Belnap bilattice. We overload the symbol $[\![.]\!]$ to denote also the meaning of a policy expression, defined as follows:

$$[\![x \text{ if } ap]\!](a) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } [\![ap]\!](a) = true \\ u & \text{otherwise} \end{cases}$$

$$[\![\neg p]\!](a) \stackrel{\text{def}}{=} \neg [\![p]\!](a)$$

$$[\![p \wedge q]\!](a) \stackrel{\text{def}}{=} [\![p]\!](a) \wedge [\![q]\!](a)$$

$$[\![p \supset q]\!](a) \stackrel{\text{def}}{=} [\![p]\!](a) \supset [\![q]\!](a)$$

The expression x if ap produces a policy from an access predicate. For example, expression g if ap evaluates to g whenever ap evaluates to *true* on an access, and evaluates to u whenever ap evaluates to *false* on an access. In this way a predicate defined to express when an access is granted does not, at the same time, define when an access is denied.

Since all other logical operators of the Belnap space can be expressed through the set of operators $\{d, \neg, \wedge, \supset\}$ [3], it is justified to refer to other policy operators such as $\oplus$ and $\otimes$ subsequently.

A simple example policy is

$$\begin{aligned} & g \text{ if } (Librarian, write, CardCatalog) \qquad (1) \\ \oplus \quad & d \text{ if } (Reader, write, CardCatalog) \end{aligned}$$

The policy says that librarians may modify card catalogs but library users must not write card catalogs. The first clause (1) evaluates to u for all accesses except those in which a librarian requests a modification of a card catalog. If the class *Librarian* is disjoint from the class *Reader*, then the composition through $\oplus$ will result in a policy that can only return a value of g, d, or u for any access.

We can make two statements about the completeness of our language. The first says that, if the access predicate sublanguage is expressive enough, then every policy can be written as a policy expression.

Proposition 1 *Let $pol : A \to$* **Four** *be a policy, and suppose that for every access a in A, there exists an access predicate expression ap_a such that $[\![ap_a]\!](b) = true$ iff $a = b$. Then there exists a policy expression p such that, for every access a, $pol(a) = [\![p]\!](a)$.*

Proof. The proof constructs a policy expression p as follows:

$$\sum_{a \in A} \left(\sum_{pol(a) \in \{d,c\}} d \text{ if } ap_a \quad \oplus \sum_{pol(a) \in \{g,c\}} g \text{ if } ap_a \right)$$

Here $\sum$ is the indexed form of $\oplus$, with the empty sum defined to be u. We now show that $pol(a) = [\![p]\!](a)$, for every a. Consider the summands produced for the top-level sum, each of which has the form $p_1 \oplus p_2$. We now do case analysis on the interpretation of a summand relative to the access a.

Consider first the summand produced for an access a' distinct from a. In this case $ap_{a'}$ is interpreted as false on a, so both p_1 and p_2 are interpreted as u, and hence $p_1 \oplus p_2$ is interpreted as $u \oplus u = u$.

Next consider the summand produced for a. We now do a case analysis on the possible values of $pol(a)$.

- Case $pol(a) = d$: In this case $pol(a) \in \{d, c\}$ and $pol(a) \notin \{g, c\}$, so the summand is equivalent to the policy expression $(d \text{ if } ap_a) \oplus u$, which is equivalent to d if ap_a, which is interpreted as d on a.
- Case $pol(a) = c$: In this case $pol(a) \in \{d, c\}$ and $pol(a) \in \{g, c\}$, so the summand for access a is equivalent to $(d \text{ if } ap_a) \oplus (g \text{ if } ap_a)$, which is interpreted as c on a.
- The other cases are similar. □

The second statement concerns the ability of the language to express policy compositions. We shall explain this fact loosely, rather than introducing the additional concepts and notation needed for a precise statement. Consider the set of policy operations that can be defined through the pointwise extension of functions on the Belnap space. Assume that the language of access predicates contains an expression that always evaluates to the constant *true*. Then all such policy operations can be represented in our policy language, simply because all functions on the Belnap space, of any arity, can be represented with operators $\neg$, $\wedge$, $\supset$, and constants g and d – a direct consequence of the functional completeness result of [3] – and because g and d can be obtained from g if *true* and d if *true* (respectively).

To summarize, our policy language is complete in the sense of being able to express any policy (relative to a sufficiently expressive language of access predicates), and also complete in the sense that it can express all policy operators that are the pointwise extensions of functions on the Belnap space.

However, there are some useful policy composition operations that are not pointwise extensions of functions on the Belnap space. One example, the "majority rule" operator, grants access if the majority of relevant sub-policies grant access, even if some sub-policies deny access. This cannot be expressed in our language because

$$g \oplus u = g \oplus g$$

Our language does not allow one to distinguish between two policies granting access, and one policy granting access while another is unspecified on that access.

A second useful composition not handled by our language is the "most specific rule wins" operation. Suppose a hospital policy states that physicians cannot perform surgery to install a heart stent, but also states that a cardiologist can perform surgery to implant a heart stent. On the surface there is a conflict, because a cardiologist is a kind of physician, so both rules apply to cardiologists. However, a common way to compose these two simple policies is to allow a cardiologist to perform the surgery, letting the more specific rule take priority. Our language knows nothing about subject hierarchies, so cannot make such distinctions. In Section 7 we look at an extension of our language that allows roles to be considered in composition.

4.2 High-Level Policy Operators

Our basic policy language can be inconvenient for practical use. For example, suppose we want a version of policy p that is restricted to accesses defined by the access predicate ap. This can be written as $p \otimes ((g \text{ if } ap) \oplus (d \text{ if } ap))$. One can define a new operator p if ap to capture this idea succinctly. We therefore define these additional policy op-

erations, where x ranges over values in **Four**.

$$
\begin{array}{rcl}
(p+q)(a) & \stackrel{\text{def}}{=} & p(a) \oplus q(a) \\
(x)(a) & \stackrel{\text{def}}{=} & x \\
(p[x \mapsto q])(a) & \stackrel{\text{def}}{=} & \begin{cases} p(a) & \text{if } p(a) \neq x \\ q(a) & \text{otherwise} \end{cases} \\
(p \text{ if } ap)(a) & \stackrel{\text{def}}{=} & \begin{cases} p(a) & \text{if } ap(a) = true \\ u & \text{otherwise} \end{cases} \\
(p : q)(a) & \stackrel{\text{def}}{=} & \begin{cases} q(a) & \text{if } p(a) \in \{g, c\} \\ u & \text{otherwise} \end{cases} \\
(p > q)(a) & \stackrel{\text{def}}{=} & p[u \mapsto q] \\
(p\downarrow)(a) & \stackrel{\text{def}}{=} & p[c \mapsto d][u \mapsto d] \\
(p\uparrow)(a) & \stackrel{\text{def}}{=} & p[c \mapsto g][u \mapsto g]
\end{array}
$$

The operator $+$ is simply a synonym for $\oplus$, an important operator for policy composition. Given an access a, policy $p + q$ returns the value $p(a)$ if p and q agree on a, or if p returns a value greater in $\leq_k$ than q (and symmetrically). Policy $p+q$ returns c if, for example, one of p and q returns d and the other returns g.

The operator x is a policy that always returns Belnap constant x. The operator $[x \mapsto q]$ allows for the selective "repair" or "overwriting" of one policy by another. The operator p **if** ap is a generalization of the basic operator x **if** ap. The operator $p : q$, due to Fitting [13] and called *guard* connective, is a kind of four-valued conditional expression.

The operator $>$ is a policy priority operator. In expression $p > q$, policy p is given priority; policy q is only used to cover "gaps" in p. With the priority operator one can easily capture case statements. Informally, a statement of the form

$$
\begin{array}{lll}
\textbf{case } \{ & ap_1 & : d; \\
& ap_2 & : g; \\
& \textbf{default} & : p \qquad \}
\end{array}
$$

can be written

$$(d \text{ if } ap_1) > (g \text{ if } ap_2) > p$$

The operator $\downarrow$ turns a four-valued policy into a two-valued one by treating c and u conservatively as d. Ultimately, the top-level policy driving an access control mechanism should be two-valued. In a considered approach, one might identify and eliminate any gaps or conflicts in the policy. As a stopgap one might use this operator. By using a four-valued approach we allow conflicts to be detected and reconciled appropriately. In some cases these conflicts might reflect bugs in the policy. In a two-valued approach that removes conflict at each composition, such errors would remain invisible.

The operator $\uparrow$ is the optimistic dual of $\downarrow$ that treats c and u as g. In other words, if p makes no statement, or conflicting statements, about an access a, then $p(a) = g$.

We have defined these policy operators directly for clarity. However, they are all pointwise extensions to functions in the Belnap space and therefore could have been defined in terms of the operations in our basic policy language.

Proposition 2 *Let p and q be policy expressions. We can define our high-level policy operators through the operators of our core policy language as follows:*

$$
\begin{array}{rcl}
[\![g]\!] & = & [\![g \text{ if } true]\!] \\
[\![u]\!] & = & [\![g \text{ if } false]\!] \\
[\![d]\!] & = & [\![\neg g]\!] \\
[\![c]\!] & = & [\![g \oplus d]\!] \\
[\![p[d \mapsto q]]\!] & = & p \vee (\neg(p \vee -p) \wedge q) \\
[\![p[g \mapsto q]]\!] & = & p \wedge (\neg(p \wedge -p) \vee q) \\
[\![p[u \mapsto q]]\!] & = & p \oplus (-(p \oplus \neg p) \otimes q) \\
[\![p[c \mapsto q]]\!] & = & p \oplus (-(p \otimes \neg p) \otimes q) \\
[\![p \text{ if } ap]\!] & = & [\![p \otimes ((g \text{ if } ap) \oplus (d \text{ if } ap))]\!] \\
[\![p : q]\!] & = & [\![(p \supset q) \otimes \neg(p \supset \neg q)]\!]
\end{array}
$$

For the encoding of the constant policies we require that the underlying language of access predicates includes the truth constants *true* and *false*. The operator $p[x \mapsto q]$ can be regarded as an indexed set of binary operators, one for each value of x, as shown above. The encoding of $p : q$ is due to Arieli and Avron [3]. Fitting gives his own encoding in [13].

We conclude this section by discussing some algebraic properties of these high-level operators. We write $p = q$ as a shorthand for $[\![p]\!] = [\![q]\!]$. Some algebraic identities are

$$
\begin{array}{rcl}
p + q & = & q + p \\
p > (q > r) & = & (p > q) > r \\
(p \text{ if } ap) + (q \text{ if } ap) & = & (p + q) \text{ if } ap \\
p\uparrow\uparrow & = & p\uparrow \\
p\downarrow\uparrow & = & p\downarrow \\
p\downarrow\downarrow & = & p\downarrow \\
p\uparrow\downarrow & = & p\uparrow
\end{array}
$$

More identities for p **if** ap can be formulated whenever more structure of access predicates is being exposed syntactically. The identities for $\downarrow$ and $\uparrow$ mean that one arrow cannot interfere with the previous application of another arrow as the image values will then be in $\mathbf{Two} = \{g, d\}$.

5. EXAMPLES

In this section we supply examples to support our thesis that working with four semantic values for policy composition in access control is beneficial.

Top-level policy wrappers. Consider two policies p and q (of type $A \to \mathbf{Four}$). We can "wrap" each of these policies, e.g., with $\downarrow$, so that $p\downarrow$ and $q\downarrow$ are free of gaps and conflicts. The composition $p\downarrow + q\downarrow$ of these wrapped versions may introduce conflicts that are solely caused by the wrappers: if $p(a) = g$ and $q(a) = u$, then $p\downarrow(a) = g$ and $q\downarrow(a) = d$ and so $(p\downarrow + q\downarrow)(a) = c$. This conflict would require yet another wrapper, e.g., $(p\downarrow + q\downarrow)\downarrow(a) = d$. But the intended value of $(p+q)(a)$ is g and not d, for g is the ruling of policy p on a and policy q does not specify any recommendations for a. A dual example highlights the same problem with

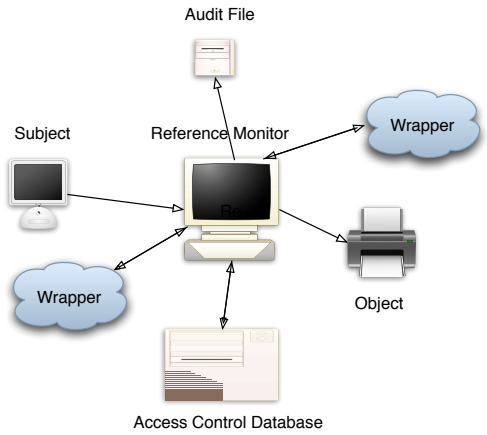

Figure 3: Implementation of wrappers as services that a reference monitor can use to resolve conflicts or fill gaps. The choice of wrapper may well be context-dependent.

the wrapper $\uparrow$, and mixed combinations of wrappers will have corresponding problems. This shows that it is best to apply wrappers only at the top level, and to remove those wrappers again prior to further policy composition. In the above example, this says that we use $p\downarrow$ and $q\downarrow$ if we want to use p and q on their own, but want to use $(p \circ q)\downarrow$ whenever we mean to compose p and q. Fig. 3 shows the use of such wrappers in a prominent architecture for access control.

Extending access domains. Consider a two-valued policy $p\colon A \to \mathbf{Two}$, where $\mathbf{Two} = \{g, d\}$, and another two-valued policy $q\colon A' \to \mathbf{Two}$ on an access domain A' that is a strict superset of A. For example, A' may include roles not found in A. For any $a \in A' \setminus A$, policy p is unspecified at a, but we have to cast p to type $A' \to \mathbf{Two}$ if we wish to compose it with q. Setting $p(a) = g$ is undesirable, because then a two-valued conjunction of p and q would lead to conflict if q denies the access. Setting $p(a) = d$ is undesirable as well, in the case that q would grant the access. This example shows the utility of having the value u as a means of expressing partial functions. This is very familiar in set theory, where the space of partial functions $X \rightharpoonup Y$ is isomorphic to the space of total function $X \to Y_u$, where $u \notin Y$. It should be clear that, if **Two** is extended with u, the semantic value u cannot also serve the role of c as this would confuse the specification of the absence of information and that of information overload.

We can take this example further by assuming, implicitly, that policies are well defined on supertypes by setting them to be constantly u on the extended domain of accesses. For example, we could write $p > (q + r)$ for policies p, q, and r with types $A \to \mathbf{Four}$, $A' \to \mathbf{Four}$, and $A'' \to \mathbf{Four}$ (respectively) where $A \subset A' \subset A''$. Implicit type casting is a familiar and well accepted mechanism in a wide range of programming languages.

Exceptions. Our next example uses the conditional p if ap. Recall that this is the policy that behaves like p whenever ap is true, and returns u otherwise. Now consider the roles Cardiologist and Physician. Intuitively, cardiologists should be permitted to engage in any action that a physician is allowed to engage in. After all, a cardiologist is a kind of physician. On the other hand, as noted in [10], there may be tasks that a physician frequently performs but that are rather alien to cardiologists, who therefore should not be permitted to perform them. These exceptions to permissions will break the normal flow of permissions associated with the specialization of roles, but one can express such exceptions through the idiom

$$(d \text{ if } ap_{exc}) > p \tag{2}$$

where p is the original policy for Cardiologist, including the permissions inherited from Physician, and ap_{exc} specifies the set of accesses to which these exceptions should apply.

A posteriori **strengthening.** A slight variation of the idiom in (2) is of use in the *a posteriori* strengthening of an access predicate. As a simple example, suppose we have policies p and q defined as follows:

$$p \stackrel{\text{def}}{=} g \text{ if } ap_1 + d \text{ if } ap_2$$
$$q \stackrel{\text{def}}{=} d \text{ if } ap_3$$

In the composite policy $p+q$ we may find conflict because the access sets represented by ap_1 and ap_3 may overlap. It may be that for accesses satisfying ap_2 we want policy q to have priority. This is achievable by redefining p so that the grant clause has the form g if $ap_1 \ \& \ \neg ap_3$ (assuming the language of access predicates has boolean connectives). However, the same result can be accomplished by using, instead of $p + q$, the expression $(p \text{ if } \neg ap_3) + q$. This approach might be preferable if p were very large, or difficult to modify.

Absolute rights and absolute prohibitions. Bonatti *et al.* [6] have an operator that allows one to define a policy having the permissions of one policy less the permissions of another. They mention that this is important in supporting "explicit prohibitions". Our view is that there is no reason to not equally accept the need for prescribing "explicit rights". In our framework, we can specify access rights for a set of accesses ap_1 by the policy g if ap_1. Similarly, we can specify access prohibitions for a set of accesses ap_2 by d if ap_2. Now, given any policy p – which may have been the result

of repeated policy compositions – we can enforce these rights and prohibitions as *absolute ones for p* by using the idiom

$$[(g \text{ if } ap_1) + (d \text{ if } ap_2)] > p$$

6. RELATIONS ON POLICIES

As part of a mathematical framework for access control policies, it is helpful to have relations on policies that reflect their relative permissiveness. For example, in [8], a permissiveness relation is defined (named *contains*) in which one policy contains another if it permits more accesses and denies fewer accesses.

A truth ordering on policies can be derived directly from the truth ordering on the Belnap space. The derived truth order reflects that one policy is "more lenient" than another.

$$p_1 \leq_t p_2 \stackrel{\text{def}}{=} \forall a \in A\colon p_1(a) \leq_t p_2(a)$$

For example, if p_1 grants an access, then so too must p_2. If p_1 is unspecified for some access, then if p_2 is specified it must grant the access.

Less obvious is the value of an ordering on policies that reflects their definedness. However, such an ordering is helpful in working with policies and again can be derived from the information ordering of **Four**.

$$p_1 \leq_k p_2 \stackrel{\text{def}}{=} \forall a \in A\colon p_1(a) \leq_k p_2(a)$$

For example, if p_1 is defined on every access, then so is p_2. Also, note that if p_1 grants some access, then p_2 must either grant that access or report conflict.

These relations can be used in various ways when working with access control policies. For example, they can be used to show that policies lie in a particular relation by construction. To illustrate, the following algebraic relations hold:

$$\begin{array}{rcl} p & \leq_k & p+q \\ q & \leq_k & p+q \\ p \wedge q & \leq_t & p \\ p \wedge q & \leq_t & q \\ p & \leq_k & p > q \\ p\downarrow & \leq_t & p \\ p & \leq_t & p\uparrow \end{array}$$

So, e.g., a policy p can be refined by $p > q$, which is guaranteed to be greater than or equal to p in the information ordering. Also, policy p is at least as permissive as $p \wedge q$.

The truth and information orderings can also be used for the analysis of arbitrary policy expressions. A basic analysis question is whether two policies are ordered according to $\leq_k$ or $\leq_t$. Going further, we can make these refinement checks the basic ingredients of a query language in which a wide variety of analysis questions can be framed. We now define a query language as a propositional logic in which the atomic propositions have the form $p \leq_t q$ and $p \leq_k q$, where p and q are any policy expressions. Fig. 4 shows the abstract syntax of the language.

$$\begin{array}{lll} \phi, \psi & ::= & \textit{Query} \\ p \leq_t q & & \text{Truth ordering} \\ p \leq_k q & & \text{Information ordering} \\ \neg\phi & & \text{Negation} \\ \phi \wedge \psi & & \text{Conjunction} \end{array}$$

Figure 4: Syntax of the Query Language.

Example 1 *We list some important example queries:*

- *policy q is more defined and more permissive than p:* $(p \leq_k q) \wedge (p \leq_t q)$
- *policy q is more defined but less permissive than p:* $(p \leq_k q) \wedge (q \leq_t p)$
- *policies p and q are semantically the same:* $p = q$ *defined as* $(p \leq_t q) \wedge (q \leq_t p)$
- *policy p has neither gaps nor conflicts:* $p = p\downarrow$
- *policy p has no gaps:* $p[c \mapsto d] = p\downarrow$
- *policy p has no conflicts:* $p[u \mapsto d] = p\downarrow$

The example above illustrates that many important analyses of policy composition reduce efficiently to checks of policy refinement. It also shows that the expressive power of such queries stems to a large degree from the expressive power of our policy composition operators. As pointed out already, certain refinement relations will simply hold by construction.

Although a detailed treatment of policy refinement checks is the subject of future work, we hasten to point out that static analyses can be used to decide or compute constraints for; e.g., in gap analysis, for $p+q$ to have any gaps, we need that p and q have a gap at some same access. Such flow laws are familiar in program analysis and SAT solvers.

7. ROLE-BASED ACCESS CONTROL

Mandatory and discretionary access control models and systems are often too restrictive and inefficient to be used in public-sector or commercial organizations, or their federations. Role-based access control (RBAC) [9, 10] models and commercial products based on RBAC are being promoted as fulfilling those needs of flexibility and efficiency.

In this section we discuss how RBAC strengthens our case for the use of four values and further illustrates the utility of our query language for policy analysis. In considering RBAC we specialize our policy language in two ways.

Firstly, A is assumed to be a finite set of type

$$A = \mathit{Roles} \times \mathit{Operations} \times \mathit{Objects}$$

so that (r, op, o) is the access (request) of some role r – where r means all users or subjects assigned to that role, either statically or in a current session – to perform operation op on object o. We leave the domains of roles, objects and operations unspecified but will instantiate them freely in examples. We also note that this paper does not provide

a mapping between roles and their users, subjects or user-initiated processes. Although such a mapping is crucial for the core RBAC, it is not important for the aspects of RBAC-based policies we wish to discuss in this paper.

Secondly, we assume a general role hierarchy [10], a partial order $\prec \subseteq Roles \times Roles$ that allows us to model a form of inheritance. The intuition behind $r \prec r'$ is that greater elements in the ordering correspond to more general roles. For example, $Cardiologist \prec Physician$.

In RBAC we have that more specialized roles ought to inherit permissions from less specialized roles. For example, everything that a physician is permitted to do should also be permitted for a surgeon, expressed in $Surgeon \prec Physician$. If two-valued access control is done, then we should dually expect that less specialized roles inherit restrictions from more specialized roles. However, this does not always work. Suppose a physician is permitted to prescribe cough medicine. Then by permission inheritance a surgeon can also prescribe cough medicine. But since surgeons are specialists it may be appropriate to stipulate that they be denied the right to prescribe cough medicines. In that case, a two-valued treatment of permission inheritance would force that physicians are also not permitted to prescribe cough medicine.

This problem goes away if we use four-valued policies; then more specialized roles inherit both permissions and restrictions from more general roles. Both requirements can be stated simply by saying that the result for a more specialized role should be greater in the information ordering than the result for a less specialized role. This intuitively makes sense because in any policy we are saying more about more specialized roles.

We now illustrate the utility of our query language for analyzing policies that are based on hierarchical RBAC. In doing so, we extend our policy language with a clause

$$p ::= \ldots \mid p \setminus r$$

where r ranges over $Roles$ and the semantics of $p \setminus r$, for a given access (r', op, o) is simply the semantics of p for the access (r, op, o). That is to say, $p \setminus r$ takes an access as argument, replaces its role with that of r, and then applies the original policy p to the modified access. Given this extension, the query

$$\bigwedge_{r \prec r'} (p \setminus r') \leq_k (p \setminus r)$$

evaluates to true if, and only if, policy p respects the inheritance structure of the underlying hierarchical RBAC model.

8. RELATED WORK

Halpern and Weissman [16] use a stylized form of standard first-order logic to capture policies. A policy ϕ is a formula, and then the decision on whether to grant an access is made by checking the validity of the formula $\phi \rightarrow Permitted(t, t')$, where t and t' are terms representing a subject and action, respectively. A decision on whether to deny an access is made by checking the validity of $\phi \rightarrow \neg Permitted(t, t')$. Because this is a first-order framework, it is essential that the policy be consistent, and so it is impossible that an access could be both granted and denied. On the other hand, an access could be neither granted nor denied in a consistent policy. In short, this framework allows a kind of three-valued attitude towards accesses, but these values cannot be used in composition, as the three values result from a validity check that can only occur at the "top-level" of policy processing.

In [6], Bonatti *et al.* interpret a policy as a set of accesses, and then define a collection of composition operators that manipulate the access sets. For example, $P\&Q$ stands for the intersection of the access sets described by P and Q, and $P+Q$ their union. The approach has the virtue of simplicity, but does not allow for the distinction between an access that should be denied and an access that is outside the scope of a policy. Some ability to model negative permissions is achieved by composition subtraction: $p - q$ means the set of accesses of p less the set of accesses of q. In this way denials by q are given higher priority than grants by p, so conflicts cannot arise. However, one cannot symmetrically state that grants of q should take priority over denials of p. It should be possible to state absolute access rights as well as absolute prohibitions.

In the SPL policy language [22], one can specify accesses that are granted, accesses that are denied, and accesses that are neither granted nor denied. For composition, a three-valued logic is used. However, the propositional operators of this logic do not appear to come from a standard three-valued propositional logic [18]. Furthermore, although the logic appears useful for expressing certain kinds of composition, the conjunction and disjunction operators are not monotonic with respect to any ordering of the three truth values, making the intuition behind the operators difficult. Also, being three-valued, the operators always treat the combination of grant and deny (e.g., conjoining a grant and a deny, or disjoining them) as either a grant or a deny, thus, causing conflicts to be eliminated through composition.

9. DISCUSSION

We emphasize that our approach treats access predicates as black-box primitives. This high level of abstraction ensures a flexible and wide-ranging applicability of our semantic framework for policy composition in access control. It also allows an elegant casting of access predicates into four-valued policies (e.g. through x if ap) and, conversely, allows for several systematic ways of collapsing four-valued policies into access predicates (e.g. through $p\downarrow$).

Also related to the practical use of our work is the idea of combining an access control matrix for permissions with a separate one for denials. From these, a four-valued policy result could be obtained according to the bilattice depicted in Fig. 1 right).

Semantically, there are at least two ways in which the approach of this paper could be enriched. In one, the Belnap bilattice is retained but we allow composition operators that are not necessarily pointwise extensions of the Belnap operators. For example, this would allow one to define a majority rule operator maj_k. Let $p_1, \ldots, p_n$ be policies of type $A \rightarrow$ **Four** and $0 < k < n$. Then $\mathsf{maj}_k(p_1, \ldots, p_n) \colon A \rightarrow$ **Four** maps a given access a to c if at least k policies p_i map a

into $\{g, c\}$ and at least k (not necessarily the same) policies p_i map a to $\{d, c\}$. Otherwise, it is mapped to g if at least k policies map a into $\{g, c\}$; and a dual definition applies to d. In fact, if we identity **Four** with the powerset of $\{d, g\}$ in the obvious manner, then maj_k is simply counting grants and denials and uses the threshold k to determine the grants and denials for the composite policy.

Another way to enrich our semantic framework would be to use a bilattice more complex than the Belnap space. A balance would have to be drawn between the perceived increased benefit of such a generalization and the perceived overhead such complexity would bring to those who write and compose policies. For example, there is a seven-valued bilattice used for default reasoning [15]. Default reasoning is already being studied in the context of composing policies that negotiate parameters for security protocols [19].

Our approach to composition, including the application of unary operators, can be couched within the framework of abstract interpretation [7]. To illustrate the unary case, projection and closure operators are particular examples of Galois insertions, and therefore of abstract interpretations. One such example is the map

$$p \mapsto p\uparrow \; : \; (A \to \mathbf{Four}, \leq_t) \to (A \to \mathbf{Four}, \leq_t)$$

This is a closure operator as it is monotone with respect to the truth ordering, idempotent ($p\uparrow\uparrow = p\uparrow$), and inflationary ($p \leq_t p\uparrow$). Dually, the map $p \mapsto p\downarrow$ is a projection operator for the truth ordering.

There is a fairly large body of work on policy conflict analysis (e.g. [20]) and the management of inconsistencies in distributed systems (e.g. [21]). In [21] it is argued for a need to make inconsistencies (i.e. conflicts) explicit and to manage them through monitoring, diagnosing, and resolution. The approach taken in this paper is consistent with such a view.

To summarize our contributions, we have defined a language for policy composition, and shown that it neatly handles common problems in policy composition. We have further defined policy refinement relations, and built a query language on top of such refinement checks that is suitable for policy analysis. One of the key findings in this work was to realize the benefits of separating policy composition from conflict resolution or analysis but, at the same time, to support such resolution and analysis through composition operators and policy refinement checks.

Acknowledgements

Glenn Bruns and Daniel S. Dantas were both supported in part by US National Science Foundation grant 0244901. This work has, in part, be performed in collaboration with the project "Aspects of Security for Citizens", funded by the Danish Strategic Research Council.

10. REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.

[2] O. Arieli and A. Avron. The Logical Role of the Four-Valued Bilattice. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS 98)*, pages 118–126. IEEE Press, 1998.

[3] O. Arieli and A. Avron. The value of the four values. *Artificial Intelligence*, 102(1):97–141, 1998.

[4] A. Banerjee and D. A. Naumann. History-based access control and secure information flow. In *Proc. of the workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS)*, *LNCS 3362*, pages 27–48. Springer-Verlag, 2005.

[5] N. D. Belnap. A useful four-valued logic. In J. M. Dunn and G. Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 8–37. D. Reidel, Dordrecht, 1977.

[6] P. Bonatti, S. de Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security*, 5(1):1–35, 2002.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, New York, 1977.

[8] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In U. Furbach and N. Shankar, editors, *Automated Reasoning – Third Int'l Joint Conference on Automated Reasoning (IJCAR)*, *LNAI 4130*. Springer-Verlag, 2006.

[9] D. Ferraiolo and D. R. Kuhn. Role-Based Access Control. In *Proc. of the NIST-NSA National (USA) Computer Security Conference*, pages 554–563, 1992.

[10] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control (Second Edition)*. Artech House, Inc., Norwood, MA, USA, 2003.

[11] M. Fitting. Bilattices in logic programming. In *20th Int'l Symposium on Multiple-Valued Logic, Charlotte*, pages 238–247. IEEE CS Press, Los Alamitos, 1990.

[12] M. Fitting. Bilattices and the semantics of logic programming. *Journal of Logic Programming*, 11(1&2):91–116, 1991.

[13] M. Fitting. Kleene's three valued logics and their children. *Fundam. Inf.*, 20(1-3):113–131, 1994.

[14] M. Fitting. Bilattices are nice things. In T. Bolander, V. Hendricks, and S. A. Pedersen, editors, *Self-Reference*, pages 53–77. Center for the Study of Language and Information, 2006.

[15] M. Ginsberg. Multivalued logics: a uniform approach to reasoning in AI. *Computational Intelligence*, 4:256–316, 1988.

[16] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proceedings of the Computer Security Foundations Workshop (CSFW'03)*, 2003.

[17] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[18] S. C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, 1952.

[19] A. J. Lee, J. P. Boyer, L. E. Olson, and C. A. Gunter. Defeasible security policy composition for web services. In *Proc. of workshop in Formal Methods in Security Engineering*, pages 45–54. ACM Press, 2006.

[20] J. Moffett and M. Sloman. Policy conflict analysis in distributed systems management. *Journal of Organizational Computing*, 4(1):1–22, 1994.

[21] B. Nuseibeh and S. Easterbrook. The process of inconsistency management: a framework for understanding. In *Proc. of workshop on Database and Expert Systems Applications*, pages 364–368. IEEE Computer Society, 1999.

[22] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: An access control language for security policies and complex constraints. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2001.