On Completeness in Languages for Attribute-Based Access Control

Jason Crampton
Royal Holloway University of London
Egham, TW20 0EX, United Kingdom
jason.crampton@rhul.ac.uk

Conrad Williams
Royal Holloway University of London
Egham, TW20 0EX, United Kingdom
conrad.williams.2010@live.rhul.ac.uk

ABSTRACT

Attribute-based access control (ABAC) has attracted considerable interest in recent years, resulting in an extensive literature on the subject, including the standardized XML-based language XACML. ABAC policies written in languages like XACML have a tree-like structure in which leaf nodes are associated with authorization decisions and non-leaf nodes are associated with decision-combining algorithms. In this paper, we consider the expressive power of the rule- and policy-combining algorithms defined by the XACML standard. In particular, we identify unexpected dependencies between the combining algorithms and demonstrate that there exist useful combining algorithms that cannot be expressed by any combination of XACML combining algorithms. We briefly discuss the decision operators defined in the PTaCL language, an abstract language for defining ABAC policies, and the advantages of replacing the XACML combining algorithms with the PTaCL operators. Following this, we review results in the literature on multi-valued logic and introduce the notion of canonically complete policy languages. We discuss important practical advantages of canonically complete policy languages, primarily in simplifying policy specification and providing efficiently enforceable policies. Finally, we propose a new policy authorization language PTaCL(E) which is canonically complete and show it is capable of expressing any arbitrary policy in a normal form and discuss the advantages of using PTaCL(E) over existing policy languages such as XACML and PTaCL.

CCS Concepts

•Security and privacy → Access control; Authorization; *Security requirements;* **•Software and its engineering → Specialized application languages;**

Keywords

XACML, PTaCL, decision operators, combining algorithms, functional completeness, canonical completeness

SACMAT'16, June 05-08, 2016, Shanghai, China

DOI: http://dx.doi.org/10.1145/2914642.2914654

1. INTRODUCTION

One of the fundamental security services in computer systems is *access control*, which is a mechanism for constraining the interaction between (authenticated) users and protected resources. Generally, access control is implemented by an authorization service, which includes an *authorization decision function* for deciding whether a user request to access a resource (an "access request") should be permitted or not. In its simplest form an authorization decision function either returns an allow or deny decision.

A common method for specifying access control models and systems is through the use of *authorization policies*, where a user request for a resource is evaluated against a policy that defines which requests are authorized. Due to the increasing rise of collaboration between industry partners, there have been many proposed languages for the specification of authorization policies for "open" systems. This has prompted a move away from traditional means of using user identities for making authorization decisions. Instead, authorization decisions are made based on user and resource attributes, allowing greater flexibility as relationships between specific users and resources no longer need to be specified. The most widely used language of this type is XACML [10, 13]. However, XACML suffers from poorly defined and counterintuitive semantics [8, 11], and is inconsistent in its articulation of policy evaluation. PTaCL is a more formal language for specifying authorization policies [5], providing a concise syntax for policy targets and precise semantics for policy evaluation.

Typically, an authorization policy is defined by a target, a set of child policies and a decision-combining algorithm. The focus of this paper is on the way in which decisions (and hence authorization policies) are combined through the use of these decision-combining algorithms. The XACML standard specifies twelve rule- and policy-combining algorithms which are used to combine authorization decisions. In principle, customized combining algorithms can be written and deployed to supplement the twelve standardized algorithms. In contrast, PTaCL specifies only three policy operators for combining decisions, and all other policy operators can be represented in terms of these three operators. This is a powerful tool for policy authors as it allows them to design and implement new operators on an ad hoc basis. However, one drawback of PTaCL (and XACML and other similar languages) is that policies are defined by combining subpolicies. In other words, policy specification is performed in a bottom-up manner.

We believe there will be, perhaps many, situations where the policy writer knows what decision should be returned for each authorization request, but is unable to construct the desired policy using the operators provided by the policy language. As a simple example, suppose we have policies defined by the tables in Figure 1. Here we are assuming there are two sub-policies P_1 and P_2, whose targets partition the set of all authorization requests. The rows represent values that are returned by evaluating P_1, while the columns represent values that are returned by evaluating P_2. The values 0, 1 and $\perp$ represent the decisions "deny", "allow" and "not-applicable", respectively. Thus, if P_1 and P_2 evaluate to 0 and 1, respectively, the final result should be 0 if the policies are combined using $\oplus_1$ and $\perp$ if combined using $\oplus_2$.

$\oplus_1$	0	1	$\perp$
0	0	0	0
1	0	1	$\perp$
$\perp$	0	$\perp$	$\perp$

$\oplus_2$	0	1	$\perp$
0	0	$\perp$	0
1	$\perp$	1	$\perp$
$\perp$	0	$\perp$	$\perp$

Figure 1: Two combining operators for policies

PTaCL is known to be functionally complete, which means it is possible, in principle, to construct the policies $P_1 \oplus_1 P_2$ and $P_1 \oplus_2 P_2$ by combining copies of P_1 and P_2 using the PTaCL operators. It is also possible in XACML to define custom policy-combining algorithms to directly construct $\oplus_1$ and $\oplus_2$. However, it would be useful, both in theory and in terms of implementation, to develop an authorization language which is functionally complete, like PTaCL, and which enables a policy writer to write down any desired policy directly using the operators of the language. In propositional logic, for example, one can use the truth table for an arbitrary formula to write down a logically equivalent formula in disjunctive normal form. In short, in this paper we wish to develop a policy authorization language that has a "normal form" in which *any* desired policy can be expressed.

In order to do this, we apply concepts introduced by Jobe [6] in the study of multi-valued logics to the development of a policy language for attribute-based access control. We construct a new authorization language based on the operators in Jobe's 3-valued logic and show how arbitrary policies can easily be expressed in a normal form as a combination of Jobe's operators. In summary, the main contributions of this paper are:

- the identification of redundant XACML combining algorithms;
- a thorough investigation into how expressive the XACML combining algorithms are;
- an overview of how PTaCL operators can be used to construct arbitrary policy operators; and
- the specification of a new policy authorization language, PTaCL(E), based on Jobe's operators, which provides the means to express an arbitrary policy in a normal form.

The first part of this paper provides a comprehensive justification for the development of a new set of policy combining operators, while the second part introduces new operators chosen to address the shortcomings of existing sets of operators. More specifically, in the following section we provide a brief introduction to tree-structured languages for attribute-based access control policies, including XACML and PTaCL, and notions of completeness for authorization languages. In Section 3, we show there is significant duplication and redundancy in the XACML rule- and policy-combining algorithms, in particular, we show only two XACML rule-combining algorithms are required to express all of the XACML rule-combining algorithms. We then conduct a detailed investigation into the expressiveness of these algorithms, demonstrating that XACML is not functionally complete. We briefly discuss the advantages of replacing the XACML combining algorithms with the PTaCL operators. Then, in Section 4, we review concepts of completeness and normal forms for 3-valued logics. In Section 5, we apply Jobe's results on multi-valued logics to develop a new language PTaCL(E) for specifying attribute-based access control policies. We introduce a new policy authorization language and a normal form for expressing policies in this language, and discuss some of the advantages of using this language. We conclude the paper with a summary of our contributions and suggest ideas for future work.

2. BACKGROUND AND RELATED WORK

In the context of attribute-based access control (ABAC), we assume there exists a set of attributes, each of which can take a range of values. An authorization request is specified in terms of attribute name-value pairs. Given a set of requests, an ABAC policy specifies whether each request is authorized or not.

Much of the research on ABAC policies assumes that policies are constructed from sub-policies. One sub-policy might, for example, specify that some subset of requests is allowed, while another sub-policy specifies that some subset of requests is denied. Defining policies in this way inevitably means that the sub-policies may "clash", so research in this area has focused on ways of resolving the conflicts that may arise when combining policies.

There are two broad approaches, which we may label as "policy algebras" [2, 14, 11, 12] and "tree-structured languages" [5, 13]. A policy algebra defines the semantics of a policy in terms of the sets of requests it allows and denies. Then sub-policies are combined by defining policy operators that are defined in terms of set operations (such as intersection, union and set difference) on the sets of allowed and denied requests. In contrast, a tree-structured language defines what decision to return for each sub-policy and then combines the decisions arising from the evaluation of sub-policies using decision-combining algorithms.

Of course, there are strong parallels between the two approaches, and it is often possible to define exact correspondences between policy operators and decision-combining algorithms. Nevertheless, the popularity and widespread use of XACML has led to more research on tree-structured languages in recent years (in comparison to policy algebras) [4, 5, 8, 13].

2.1 Tree-structured languages

Informally, we say a language is *tree-structured* if a policy is specified by a decision-combining algorithm and a set of child policies. A request is evaluated with respect to a policy by first computing a decision for each of the child policies and then combining those decisions using the decision-combining algorithm.

More formally, we assume the existence of a set of requests, defined in terms of attributes. Each policy specifies a target defining, in terms of attribute values, the set of requests to which a policy applies. A target t is evaluated with respect to a request q. We write $[\![t]\!](q) \in \{0_\mathsf{t}, 1_\mathsf{t}\}$ to indicate the result of evaluating target t with respect to request q, where

$$[\![t]\!](q) = \begin{cases} 1_\mathsf{t} & \text{if the target is applicable,} \\ 0_\mathsf{t} & \text{otherwise.} \end{cases}$$

We do not discuss here how target applicability is determined; the reader is referred to the literature for further details [5, 13].

We define a set of (authorization) decisions $D = \{0_\mathsf{a}, 1_\mathsf{a}, \bot_\mathsf{a}\}$, representing "allow", "deny" and "not-applicable", respectively. Then an *atomic* policy has the form (t, d), where t is a target and d is a decision. We define the evaluation of an atomic policy (t, d) as

$$[\![(t, d)]\!](q) = \begin{cases} d & \text{if } [\![t]\!](q) = 1_\mathsf{t}, \\ \bot_\mathsf{a} & \text{otherwise.} \end{cases}$$

A policy may be represented as a triple $(t, A, \overline{p})$, where t is a target, A is a decision-combining algorithm and $\overline{p} = \langle p_1, \ldots, p_k \rangle$ is a tuple of policies. Then we define

$$[\![(t, A, \overline{p})]\!](q) = \begin{cases} A([\![p_1]\!](q), \ldots, [\![p_k]\!](q)) & \text{if } [\![t]\!](q) = 1_\mathsf{t}, \\ \bot_\mathsf{a} & \text{otherwise.} \end{cases}$$

Henceforth, we will confine our attention to authorization decisions, rather than target evaluation. Hence, we will simplify the notation and use $\{0, 1, \bot\}$ (that is, without the subscript) to denote the set of authorization decisions.

In general, a decision-combining algorithm may take an arbitrary number of inputs. It is convenient, in terms of formal exposition, to assume that a decision-combining algorithm is implemented using binary *decision operators*. (Thus, we would apply a binary decision operator $k - 1$ times to evaluate a call to a decision-combining algorithm with k inputs.)

Hence, we may visualize a policy as a binary tree, in which the atomic policies are leaves and non-leaf nodes are target-operator pairs. The policy P, for example

$$\Big(t_6, A_2, \Big(\big(t_5, A_1, ((t_1, d_1), (t_2, d_2), (t_3, d_3))\big), (t_4, d_4)\Big)\Big)$$

may be represented by the binary tree depicted in Figure 2.

Then policy evaluation, from an algorithmic perspective, consists of assigning decisions to the leaf nodes, by determining whether the targets are applicable or not, and then combining the decisions using the decision operators, until a decision is obtained at the root node.

We say two policies p and p' are *equivalent*, denoted by $p \equiv p'$, if $[\![p]\!](q) = [\![p']\!](q)$ for all requests q. To simplify the notation, we will, henceforth, omit q when it is obvious from context. We will also make use of the following terminology [4] when describing decision operators.

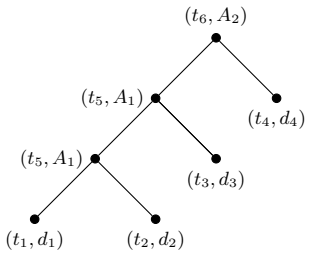

Figure 2: Policy tree

DEFINITION 1. *Let $\oplus : D \times D \to D$ be a decision operator. Then*

- $\oplus$ *is* commutative *if $d \oplus d' = d' \oplus d$ for all $d \in D$;*
- $\oplus$ *is* idempotent *if $d \oplus d = d$ for all $d \in D$, and* quasi-idempotent *if $d \oplus d = d$ for all $d \in \{0, 1\}$;*
- $\oplus$ *is* conclusive *if $d \oplus d' \in \{0, 1\}$ for all $d, d' \in D$, and* quasi-conclusive *if $d \oplus d' \in \{0, 1\}$ for all $d, d' \in \{0, 1\}$;*
- $\oplus$ *is a* $\cup$-operator *if $d \oplus \bot = d = \bot \oplus d$ for all $d \in D$;*
- $\oplus$ *is a* $\cap$-operator *if $d \oplus \bot = \bot = \bot \oplus d$ for all $d \in D$;*
- $\oplus$ *is* well-behaved *if it is either a $\cup$- or an $\cap$-operator.*

Any binary operator on the decision set $\{0, 1, \bot\}$ can be represented as a 3×3 array, as shown in Figure 3a. These decision tables correspond to the XACML rule-combining algorithms deny-overrides (do), permit-overrides (po), deny-unless-permit (dup), permit-unless-deny (pud), and first-applicable (fa) operators.

2.1.1 XACML

XACML [13] is the most commonly used authorization language for implementing attribute-based access control in the real world. An XACML rule corresponds to an atomic policy (as defined in the previous section), where the "effect" of the rule is the decision.[1] Similarly, an XACML *policy* may be represented as a triple $(t, A, \overline{r})$, where t is a target, A is a rule-combining algorithm and $\overline{r} = \langle r_1, \ldots, r_k \rangle$ is a tuple of rules. Thus an XACML policy is a non-leaf node in which all the child policies are leaf nodes (that is, rules). XACML also defines *policy sets*, which are triples of the form $(t, A, \overline{p})$, where $\overline{p} = \langle p_1, \ldots, p_k \rangle$ is a tuple of policies, t is a target, and A is a policy-combining algorithm. XACML policies and policy sets are evaluated in exactly the same way: (recursively) evaluate the "child" policies and then combine the decisions.

XACML 3.0 defines 11 rule-combining algorithms [13, Appendix C]. XACML defines ordered and unordered versions of most of the algorithms and also provides backward compatibility with previous versions of XACML. The decision tables for the ordered, unordered and legacy versions of do,

[1] In this paper we do not consider XACML "conditions", mostly because this notion is inadequately constrained in XACML 3.0. Indeed, a condition can be any boolean expression, including arbitrarily complex functions. In particular, the notion of condition makes the notion of target redundant, because any target can be expressed as a condition.

do	0	1	⊥
0	0	0	0
1	0	1	1
⊥	0	1	⊥

po	0	1	⊥
0	0	1	0
1	1	1	1
⊥	0	1	⊥

dup	0	1	⊥
0	0	1	0
1	1	1	1
⊥	0	1	0

pud	0	1	⊥
0	0	0	0
1	0	1	1
⊥	0	1	1

fa	0	1	⊥
0	0	0	0
1	1	1	1
⊥	0	1	⊥

(a) Decision tables

Operator	Idempotent	∪-operator	Commutative	Conclusive	Quasi-conclusive
do, po	Yes	Yes	Yes	No	Yes
dup, pud	No	No	Yes	Yes	Yes
fa	Yes	Yes	No	No	Yes

(b) Properties

Figure 3: XACML rule-combining algorithms do, po, dup, pud and fa

po, dup and pud are identical for the decision set $\{0, 1, \bot\}$.[2] The one algorithm that is non-commutative (that is, the order of rule evaluation matters) is first-applicable (fa). Henceforth, we will treat the XACML rule-combining algorithms as decision-combining binary operators, as defined in Figure 3a. All five of the XACML decision operators in Figure 3a are ∪-operators (and thus well-behaved). Other properties of the XACML decision operators are summarized in Figure 3b.

All of the XACML rule-combining algorithms have associated policy-combining algorithms which have identical properties and produce the same decision tables[3]; the only difference is that they take decisions returned by the evaluation of policies (as opposed to rules) as input. Thus, the features and properties shown in Figure 3 also hold true for the XACML policy-combining algorithms. There is one additional policy-combining algorithm – only-one-applicable – which does not have an associated rule-combining algorithm. We do not consider this algorithm as it introduces a fourth, indeterminate, decision, and does not appear to be very useful.

2.1.2 PTaCL

PTaCL [5] is a tree-structured policy language intended to provide a generic framework for specifying target-based policy languages. Like XACML, it is defined (without indeterminacy) over a three-valued decision set, which makes it useful for comparisons with and analysis of XACML.

PTaCL defines two unary operators and one binary operator, whose decision tables are shown in Figure 4. The unary operators ¬ and ∼ simply modify the ⊥ decision: the former switches the values of 0 and 1, leaving ⊥ unchanged; the latter transforms ⊥ to 0, leaving 0 and 1 unchanged. These operators are used to implement policy negation and deny-by-default policies, respectively. The binary operator $\wedge_{\mathrm{p}}$ corresponds to strong conjunction in the Kleene three-valued logic [7]. It returns 0 if at least one of the operands is 0, 1 if both operands are 1, and ⊥ otherwise.

[2]There are some minor differences in the way in which the indeterminate value is handled. In this paper we do not consider indeterminacy further. To do so would complicate the exposition and we believe it is straightforward to extend our work to include indeterminate values using techniques proposed by Li *et al.* [8] (and since incorporated into XACML 3.0 [13]).

[3]For reference see Appendix C in the XACML standard [13].

$\wedge_{\mathrm{p}}$	0	1	⊥
0	0	0	0
1	0	1	⊥
⊥	0	⊥	⊥

(a) $\wedge_{\mathrm{p}}$

d	$\neg d$	$\sim d$
0	1	0
1	0	1
⊥	⊥	0

(b) ¬ and ∼

Figure 4: Decision operators in PTaCL

2.1.3 Policy representation

Tree-structured policies are evaluated in a bottom-up fashion and they must also be written in this bottom-up fashion: that is, by combining rules to form policies and policies to form policy sets. However, this may not be the most natural way of defining a policy. An administrator might simply wish to specify a decision for each of the possible outcomes arising from the evaluation of sub-policies $P_1, \ldots, P_n$. In particular, it would be natural to tabulate the possible decisions that can arise from the evaluation of $P_1, \ldots, P_n$, and for each row in the table, specify the value that P should take, as shown in Table 1 below. We may also view the decision tables in Figure 1 as two simple examples of a policy defined by the outcome of evaluating two sub-policies.

P_1	$\ldots$	P_n	P
0	$\cdots$	0	d_1
0	$\cdots$	1	d_2
0	$\cdots$	⊥	d_2
$\vdots$	$\ldots$	$\vdots$	$\vdots$

Table 1: A policy decision table

It is not at all obvious how we would represent such a policy by using $P_1, \ldots, P_n$ as rules and the standard rule- and/or policy-combining algorithms provided by XACML or PTaCL's decision operators. Moreover, as we will see in Section 3, XACML is not functionally complete, so it is possible to specify policies that cannot be represented in XACML at all (unless bespoke algorithms are written to cater for the policy).

2.2 Completeness properties

In an abstract sense, a policy may be thought of as some function from the set of requests to the set of authorization decisions. Table 1 is one possible representation of such a function. The expressivity of a policy language is a measure of the language's ability to represent an arbitrary function using the operators provided by the language. One of the contributions of this paper is to show that XACML is not functionally complete and to identify the limitations of XACML as a policy specification language.

Some languages [5] and policy algebras [12] are known to be *functionally complete*, in the sense that any arbitrary function can be expressed as a policy in the language or algebra. However, we believe that a practical policy specification language should also admit some kind of canonical representation of each policy, preferably a representation that is related to a policy's decision table. (Here we have in mind the correspondence between the truth table of an arbitrary Boolean function and its expression in disjunctive normal form.) Such a representation is likely to have significant benefits in terms of policy specification and request evaluation. A second contribution of this paper is to argue that PTaCL is unlikely to have a canonical representation for policies and to propose a set of decision-combining operators for a tree-structured language that does have a canonical representation.

3. XACML OPERATORS

In this section, we investigate what decision operators can be constructed using do, po, dup, pud and fa. In doing so, we characterize the expressive power of XACML policies. We first make the following remark:

REMARK 1. *Any operator constructed using* fa, po *and* do *will be an idempotent* $\cup$*-operator, since* $x \oplus \bot = \bot \oplus x = x$ *for* $\oplus \in \{\mathsf{fa}, \mathsf{po}, \mathsf{do}\}$. *A corollary of this observation is that the operators* do, po *and* fa *are indistinguishable when one of the arguments is* $\bot$.

This seems somewhat counterintuitive, creates redundancy, and also suggests that combinations of XACML operators will tend to behave in similar ways. In the rest of this section, we confirm these observations.

3.1 Dependencies

Interestingly, despite the fact that do and po are commutative operators, it is possible to construct non-commutative operators by combining these two operators. Specifically, we have the following, somewhat unexpected, result, which asserts that the fa algorithm is redundant.

PROPOSITION 1. *For all rules* r *and* r',

$$r \,\mathsf{fa}\, r' \equiv r \,\mathsf{po}\, (r \,\mathsf{do}\, r').$$

PROOF. The proof follows by inspection of the decision table in Figure 5. □

REMARK 2. *The ability to construct* fa *from* do *and* po *arises from the fact that* do *and* po *do not obey the identity* $x \oplus (y \otimes z) = (x \oplus y) \otimes (x \oplus z)$, *usually known as the* distributive law. *Specifically,*

$$0 \,\mathsf{po}\, (1 \,\mathsf{do}\, \bot) = 0 \,\mathsf{po}\, 1 = 1, \textit{whereas}$$
$$(0 \,\mathsf{po}\, 1) \,\mathsf{do}\, (0 \,\mathsf{po}\, \bot) = 1 \,\mathsf{do}\, 0 = 0;$$

d_1	d_2	d_1 do d_2	d_1 po $(d_1$ do $d_2)$	d_1 fa d_2
0	0	0	0	0
0	1	0	0	0
0	$\bot$	0	0	0
1	0	0	1	1
1	1	1	1	1
1	$\bot$	1	1	1
$\bot$	0	0	0	0
$\bot$	1	1	1	1
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 5: Encoding fa using do and po

and

$$1 \,\mathsf{do}\, (0 \,\mathsf{po}\, \bot) = 1 \,\mathsf{do}\, 0 = 0, \textit{whereas}$$
$$(1 \,\mathsf{do}\, 0) \,\mathsf{po}\, (1 \,\mathsf{do}\, \bot) = 0 \,\mathsf{po}\, 1 = 1.$$

We now show that dup and pud are also redundant. We first define the rules **1** and **0**, where, for all requests q,

$$[\![\mathbf{1}]\!](q) = 1 \quad \text{and} \quad [\![\mathbf{0}]\!](q) = 0$$

Rule **1** may be realized in XACML by defining a rule such that the rule is applicable to every request and its effect is "permit"; rule **0** may be realized in an analogous way. We can then define the unary operators deny-by-default (dbd) and permit-by-default (pbd), where

$$\mathsf{dbd}(r) \equiv (\mathbf{0} \,\mathsf{po}\, r) \quad \text{and} \quad \mathsf{pbd}(r) \equiv (\mathbf{1} \,\mathsf{do}\, r).$$

Note that dbd and pbd may be viewed as unary operators on the decision set $\{0, 1, \bot\}$, where $\mathsf{dbd}(x) = \mathsf{pbd}(x) = x$ if $x \in \{0, 1\}$; $\mathsf{dbd}(\bot) = 0$; and $\mathsf{pbd}(\bot) = 1$. We can use dbd and pbd to construct pud and dup.

PROPOSITION 2. *For all rules* r *and* r',

$$r \,\mathsf{pud}\, r' \equiv \mathsf{pbd}(r \,\mathsf{do}\, r') \quad \textit{and} \quad r \,\mathsf{dup}\, r' \equiv \mathsf{dbd}(r \,\mathsf{po}\, r').$$

PROOF. The proof for pud follows by inspection of the decision table in Figure 6. A similar decision table can be constructed for dup. □

d_1	d_2	d_1 do d_2	pbd$(d_1$ do $d_2)$	d_1 pud d_2
0	0	0	0	0
0	1	1	0	0
0	$\bot$	1	0	0
1	0	0	0	0
1	1	1	1	1
1	$\bot$	1	1	1
$\bot$	0	0	0	0
$\bot$	1	1	1	1
$\bot$	$\bot$	$\bot$	1	1

Figure 6: Encoding pud using do and pbd

We have shown there is a significant amount of duplication and redundancy between the 11 XACML rule-combining algorithms. In particular, we have shown it is sufficient, for the purposes of constructing new decision operators, to consider the decision operators do and po, together with the constant rules **0** and **1**, and the unary operators dbd and pbd.

3.2 Incompleteness

Any XACML policy set is constructed by combining XACML policies using policy-combining algorithms. The decision obtained by evaluating an XACML policy set is determined by the action of the policy-combining algorithm on decisions. Given the way in which policy evaluation works in XACML, this is equivalent to asking what functions we can build using $\{\mathbf{0}, \mathbf{1}, \mathsf{dbd}, \mathsf{pbd}, \mathsf{do}, \mathsf{po}\}$. We have seen that do and po essentially act as logical AND and OR on the set $\{0, 1\}$; and we have seen that we can define two unary operators (dbd and pbd) for policies, which correspond to unary operators on $\{0, 1, \bot\}$.

There are two types of operators, likely to be useful in practice, that cannot be constructed using the XACML operators.

- A $\cap$-operator $\oplus$ has the property that $x \oplus \bot = \bot \oplus x = \bot$ for any $x \in \{0, 1, \bot\}$. In this context, we do not make a conclusive decision if at least one of the inputs is unknown. The operators do' and po' in Figure 7 are examples of this type of operator.
- The second type of operator has the property that a conclusive decision is returned whenever one is implied by at least one of the arguments. In this context, $\bot$ is interpreted as a value that could be either 0 or 1 but is not known at the time of evaluation. Thus $1 \,\mathsf{po}\, \bot = \bot \,\mathsf{po}\, 1 = 1$, since $1 \,\mathsf{po}\, x = x \,\mathsf{po}\, 1 = 1$ for any $x \in \{0, 1\}$; similarly $0 \,\mathsf{do}\, \bot = \bot \,\mathsf{do}\, 0 = 0$. The operators do'' and po'' in Figure 7 are examples of this type of operator. Note that do'' is equivalent to $\wedge_{\mathrm{p}}$ (the conjunction operator in PTaCL).

More formally, we have the following result.

PROPOSITION 3. *It is not possible to construct $\cap$-operators using the XACML operators.*

PROOF. The proof follows from the following observations: (i) all the binary XACML operators are $\cup$-operators; (ii) any combination of $\cup$-operators is itself a $\cup$-operator (since $x \oplus \bot = x$ for any $x \in \{0, 1, \bot\}$ and any $\cup$-operator $\oplus$); (iii) the two unary operators remove $\bot$; and (iv) $x \oplus \bot \neq \bot$ and $\bot \oplus x \neq \bot$ for any $x \in \{0, 1\}$ for any $\cup$-operator $\oplus$. Thus it is impossible to construct an operator in which $x \oplus \bot = \bot$. □

do'	0	1	$\bot$
0	0	0	$\bot$
1	0	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

do''	0	1	$\bot$
0	0	0	0
1	0	1	$\bot$
$\bot$	0	$\bot$	$\bot$

po'	0	1	$\bot$
0	0	1	$\bot$
1	1	1	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

po''	0	1	$\bot$
0	0	1	$\bot$
1	1	1	1
$\bot$	$\bot$	1	$\bot$

Figure 7: Commutative, idempotent operators that cannot be constructed using XACML operators

We have shown there is a significant amount of duplication and redundancy in the XACML rule-combining algorithms. Specifically, only do and po are required to express all 11 combining algorithms. We have also shown that XACML is not functionally complete, and there are operators of practical relevance that cannot be constructed using the XACML operators.

In fact, there are only 22 binary quasi-idempotent operators that can be constructed from the XACML operators (of the 192 that are possible). These operators fall into one of four families: (i) six do operators; (ii) six po operators; (iii) five fa operators; and (iv) five la operators. Further details can be found in the appendix.

It is interesting to note that there is no way to negate policy decisions in XACML. Quite apart from the general incompleteness of XACML, the inability to negate decisions seems to be significant practical drawback to XACML, as negation is a useful unary policy operator in practice.

3.3 Using PTaCL operators

Crampton and Morisset showed that the three-valued logic expressed over the set $\{0, 1, \bot\}$ and defined by the operators $\wedge_{\mathrm{p}}, \neg$ and $\sim$ (Figure 4) is functionally complete. Essentially, they proved that the PTaCL operators could be used to construct the operators of a logic that was known to be functionally complete.

Given that PTaCL is functionally complete, there appears to be a good case for using the PTaCL operators in a language like XACML. The unary operator $\sim$ is already implicitly defined in XACML (as dbd), thus we only need to consider adding $\wedge_{\mathrm{p}}$ and $\neg$ to the minimal set of XACML combining algorithms $\{\mathsf{do}, \mathsf{po}\}$. (Recall fa, pud and dup can be defined in terms of do and po.) It is easy to see that we cannot achieve functional completeness by adding just $\neg$ or just $\wedge_{\mathrm{p}}$ to the set of XACML operators. In the case of $\neg$, we would still be unable to construct $\cap$-operators, as there is still no operator that can change a conclusive decision into $\bot$. On the other hand, if we include $\wedge_{\mathrm{p}}$ but not $\neg$, we are unable to reverse the 0 and 1 decisions. In short, we must include both operators if we wish to make XACML functionally complete.

Given that PTaCL is functionally complete anyway, it seems pointless to provide po, as do (or any of the other 10 XACML operators). In particular, we can define the following operator

$$d \vee_{\mathrm{p}} d' \stackrel{\text{def}}{=} \neg((\neg d) \wedge_{\mathrm{p}} (\neg d')).$$

It is then possible to show that

$$d \,\mathsf{po}\, d' \equiv (d \vee_{\mathrm{p}} (\sim d')) \wedge_{\mathrm{p}} ((\sim d) \vee_{\mathrm{p}} d'), \text{ and}$$
$$d \,\mathsf{do}\, d' \equiv \neg((\neg d) \,\mathsf{po}\, (\neg d')).$$

In other words, there appears to be a good case, at least from the perspective of functional completeness, for defining only three policy operators in an ABAC language such as XACML: negation, deny-by-default, and a form of deny-overrides that only returns 1 when both arguments are 1.

It would be easy to write three custom XACML combining algorithms to implement the PTaCL operators. (The front-end of an XACML-based system could continue to expose specific algorithms (such as the usual deny-overrides and permit-overrides), if required by the application, but these can be compiled down into the three basic operators.) More complex policy-combining algorithms can be constructed, as required, from the three basic operators. However, it is still

far from obvious how one would express an arbitrary policy decision table as a policy defined using the PTaCL operators.

4. CANONICAL COMPLETENESS IN MULTI-VALUED LOGICS

In this section, we introduce the theoretical foundations, based on results of Jobe [6], for developing an authorization language that is functionally complete and admits a simple normal form for policies, enabling the author of a policy to simply write down any desired policy from its decision table. We will use these foundations to propose a new set of policy operators for an ABAC language whose policies are evaluated in the same way as PTaCL and XACML.

4.1 Canonical suitability

Let L be a logic associated with a set V of m ordered values, $\{v_1, \ldots, v_m\}$, such that $v_1 < v_2 < \cdots < v_m$. Then L is *canonically suitable* if and only if there exist in L two formulas $\phi_{\max}$ and $\phi_{\min}$ of arity 2 such that $\phi_{\max}(x, y)$ returns $\max\{x, y\}$ and $\phi_{\min}(x, y)$ returns $\min\{x, y\}$.

EXAMPLE 1. *The 2-valued logic with values 0 and 1, and operators $\vee$ and $\neg$, representing disjunction and negation, respectively, is canonically suitable: $\phi_{\max}(x, y)$ is simply $x \vee y$, while $\phi_{\min}(x, y)$ is $\neg(\neg x \vee \neg y)$ (that is, conjunction).*

If a logic is canonically suitable, we will write $\phi_{\min}(x, y)$ and $\phi_{\max}(x, y)$ using infix binary operators: $x \curlywedge y$ and $x \curlyvee y$. For a 3-valued canonically suitable logic, with values $1, 2, 3$, the truth tables for $x \curlywedge y$ and $x \curlyvee y$ are shown in Figure 8.

x	y	$x \curlywedge y$	$x \curlyvee y$
3	3	3	3
3	2	2	3
3	1	1	3
2	3	2	3
2	2	2	2
2	1	1	2
1	3	1	3
1	2	1	2
1	1	1	1

Figure 8: Truth tables for $\curlywedge$ and $\curlyvee$ in a 3-valued canonically suitable logic

4.2 Selection operators and functional completeness

A formula containing n variables in an m-valued logic, is completely specified by a truth table containing n columns and m^n rows. However, not every truth table may be represented by a formula in a given logic. A logic is said to be *functionally complete* if for every positive integer n and every truth table containing n columns, there is a formula in the logic containing n variables whose evaluation corresponds to the truth table. We have seen that XACML is not functionally complete, while PTaCL is.

A *selection operator* $S_i^j(x_1, \ldots, x_n)$ is an n-ary operator whose truth table has value v_j $(1 \leqslant j \leqslant m)$ in row i $(1 \leqslant i \leqslant m^n)$, and v_1 in all other rows.[4] (Note that $S_i^1(x_1, \ldots, x_n)$ is the same for all i.) Illustrative selection operators are shown in Figure 9 for a 3-valued logic with values $1, 2, 3$.

$S_2^2(x)$	$S_3^2(x)$
1	1
2	1
1	2

$S_2^3(x, y)$	$S_6^2(x, y)$
1	1
3	1
1	1
1	1
1	1
1	2
1	1
1	1
1	1

Figure 9: Selection operators $S_2^2(x)$, $S_3^2(x)$, $S_2^3(x, y)$, and $S_6^2(x, y)$

Given the truth table of function $f : V^n \to V$, we can write down an equivalent function in terms of selection operators. Specifically, let

$$I = \{(i, j) : \text{row } i \text{ in } f\text{'s truth table contains value } v_j > v_1\};$$

then $f(x_1, \ldots, x_n)$ is equivalent to

$$\bigvee_{(i,j) \in I} S_i^j(x_1, \ldots, x_n),$$

because, informally, the effect of this function is to take the maximum value in each row of a table comprising selection operators chosen specifically to produce the correct value in the ith row. Jobe established a number of results connecting the functional completeness of a logic with the unary selection operators. These results are summarized in the following theorem.

THEOREM 1 (JOBE [6, THEOREMS 1, 2; LEMMA 1]). *A logic L is functionally complete if and only if each unary selection operator is equivalent to some formula in L.*

The proofs of Jobe's results are by induction and constructive. Informally, if each unary selection operator is equivalent to some formula in L, then we can construct a formula (in L) for any selection operator; and if we can construct a formula for any selection operator, then we can construct a formula for an arbitrary truth table. More formally, we write $\phi_i^j(x_1, \ldots, x_n)$ to denote the formula (in L) whose truth table is that of the selection operator $S_i^j(x_1, \ldots, x_n)$. (Note that such a formula may not exist for a given logic.) Given $\phi_i^j(x)$ and $\phi_i^j(x_1, \ldots, x_n)$ for all i, j, we can construct $\phi_i^j(x_1, \ldots, x_{n+1})$. Specifically,

$$\phi^k_{(i-1)m+j}(x_1, \ldots, x_{n+1}) \equiv \phi_i^k(x_1, \ldots, x_n) \curlywedge \phi_j^k(x_{n+1}), \quad (1)$$

with $1 \leqslant k \leqslant m$, $1 \leqslant i \leqslant m^n$, and $1 \leqslant j \leqslant m$. (Note that $(i-1)m + j$ takes values from 1 to m^{n+1} inclusive, as required.)

To see the intuition behind this construction, consider the effect of calculating $S_i^k(x_1) \curlywedge S_j^k(x_2)$. We may construct an $m \times m$ table, with the rows indexed by the values in the truth table for $S_i^k(x_1)$ and the columns indexed by the values in the truth table for $S_j^k(x_2)$. The entry in the rth row and cth column (by definition of $\curlywedge$) is k if $r = i$ and $c = j$ and 1 otherwise. Writing out this two-dimensional table as a

[4] Jobe called these J-operators; we prefer the more descriptive term "selection operator".

truth table with a single column, we obtain the truth table for $S^k_{3(i-1)+j}$. Using $S^2_2(x)$ and $S^3_2(y)$ from Figure 9, for example, and computing $S^2_2 \curlywedge S^3_2$, we obtain the following table:

$\curlywedge$	1	1	2
1	1	1	1
2	1	1	2
1	1	1	1

which may be written out as a truth table with a single column, corresponding to $S^2_6(x, y)$, as expected (compare last column in Figure 9). More generally, in (1), $S^k_i(x_1, \ldots, x_n) \curlywedge S^k_j(x_{n+1})$ represents the construction of a table with m^n rows and m columns, with truth values indexed by the values in the truth table for $S^k_i(x_1, \ldots, x_n)$ and columns indexed by the truth table for $S^k_j(x_{n+1})$.

4.3 Normal form

The *normal form* of formula ϕ in a canonically suitable logic is a formula ϕ' that has the same truth table as ϕ and has the following properties:

- the only binary operators it contains are $\curlyvee$ and $\curlywedge$;
- it may contain arbitrary unary operators (defined in terms of the unary operators of the logic);
- no binary operator is included in the scope of a unary operator;
- no instance of $\curlyvee$ occurs in the scope of the $\curlywedge$ operator.

In other words, given a canonically suitable logic L containing unary operators $\sharp_1, \ldots, \sharp_\ell$, a formula in normal form has the form

$$\bigvee_{i=1}^{r} \bigwedge_{j=1}^{s} \sharp_{i,j} x_{i,j}$$

where $\sharp_{i,j}$ is a unary operator defined by composing the unary operators in $\sharp_1, \ldots, \sharp_\ell$. In the usual 2-valued propositional logic with a single unary operator (negation) this corresponds to disjunctive normal form.

A canonically suitable logic is *canonically complete* if every unary selection operator can be expressed in normal form. It is known that there are canonically suitable 3-valued logics that are: (i) not functionally complete [6, 9]; and (ii) functionally complete but not canonically complete [6, Theorem 4].

Consider now the 3-valued logic E, whose operators $\wedge_e$, E_1 and E_2 are defined in Figure 10a. It is easy to establish that

$$x \curlywedge y \equiv x \wedge_e y \quad \text{and} \quad x \curlyvee y \equiv E_2(E_2(x) \wedge_e E_2(y)).$$

Thus E is canonically suitable [6, Theorem 6]. Henceforth, we will write $x \vee_e y$ to denote $E_2(E_2(x) \wedge_e E_2(y))$. The normal-form formulas for the unary selection operators are shown in Figure 10b. (Note that S^1_i is the same for all i.) Thus E is functionally and canonically complete [6, Theorem 7].

x	$E_1(x)$	$E_2(x)$
3	3	1
2	1	2
1	2	3

$\wedge_e$	3	2	1
3	3	2	1
2	2	2	1
1	1	1	1

(a) Operators

$S^1_i(x)$	$x \wedge_e E_1(x) \wedge_e E_2(x)$
$S^2_1(x)$	$E_1E_2(x) \wedge_e E_1E_2E_1(x)$
$S^2_2(x)$	$x \wedge_e E_2(x)$
$S^2_3(x)$	$E_2E_1(x) \wedge_e E_1(x)$
$S^3_1(x)$	$x \wedge_e E_1(x)$
$S^3_2(x)$	$E_2E_1(x) \wedge_e E_1E_2E_1(x)$
$S^3_3(x)$	$E_1E_2(x) \wedge_e E_2(x)$

(b) Normal forms for the unary selection operators

Figure 10: Jobe's logic E

5. A CANONICALLY COMPLETE LANGUAGE FOR ABAC POLICIES

We can immediately see that XACML is not canonically suitable, essentially because $\curlywedge$ must be a $\cap$-operator and such operators cannot be constructed using XACML combining algorithms (Proposition 3). (We have already seen that XACML is not functionally complete.)

While there is clearly an argument, based on functional completeness, for using PTaCL operators as the basis for a language to express ABAC policies, we still face the issue of actually expressing the desired policies in that language. It is all very well providing a set of functionally complete operators, but many policy authors may not be able to translate the desired policy into one using these operators.

The functional completeness of PTaCL implies that every unary selection operator has an equivalent formula in PTaCL. However, it is not clear that every unary selection operator has an equivalent formula in PTaCL *that is in normal form.*[5] Faced with this issue, we can adopt one of two approaches:

- We could try to determine whether each selection operator does indeed have an equivalent PTaCL formula that is in normal form, thus establishing that PTaCL is canonically complete.
- Alternatively, we can ask whether the PTaCL operators could be replaced with the operators from a logic that is known to be functionally and canonically complete.

We will adopt the latter approach, arguing that Jobe's logic E provides a suitable basis for a canonically complete language for ABAC policies. As well as obtaining functional and canonical completeness, we argue that the operators E_1 and E_2 have a natural interpretation in the context of access control.

[5]The fine-grained integration algebra (FIA) [12] is a functionally-complete policy algebra. The algebra defines two constants (one of which can be derived from the other), one unary operator and two binary operators. However, the emphasis on this work is very much on "integration" of multiple policies, rather than top-down specification of policies. Moreover, it is not clear how easily FIA can be integrated with standard XACML or whether it supports a normal form for policies.

The PTaCL(E) language, then, defines atomic policies and policies in exactly the same way as PTaCL. That is, an atomic policy has the form (t, d), where t is a target and $d \in \{0, 1, \bot\}$. We assume $0 < \bot < 1$, which corresponds with an intuitive understanding of the respective authorization decisions. Then a PTaCL(E) policy may be viewed as a formula in a propositional 3-valued logic, in which the variables correspond to atomic policies. A valuation on the variables is induced by the evaluation of a request, with atomic policy variable (t, d) evaluating to either $d \in \{0, 1\}$ or $\bot$, depending on whether the target is applicable or not (exactly as described in Section 2.1). In short, an atomic policy in PTaCL(E) is analogous to an XACML rule or an atomic PTaCL policy.

In the context of PTaCL(E), the values $1, 2, 3$ in Jobe's logic are translated into $0, \bot, 1$. The $\wedge_{\mathrm{e}}$ operator is a form of deny-overrides and E_2 negates the two conclusive decisions. Specifically, the PTaCL(E) operators $\wedge_{\mathrm{e}}$ and E_2 are equivalent to the PTaCL operators $\wedge_{\mathrm{p}}$ and $\neg$, respectively. Note also that

$$E_1(x) \equiv (x \vee_{\mathrm{p}} \bot) \wedge_{\mathrm{p}} (\sim(x \vee_{\mathrm{p}} \neg x)). \tag{2}$$

It is this equivalence, in fact, that Crampton and Morisset used to establish that PTaCL is functionally complete.[6]

The unary operator E_1 has the effect of flipping the values corresponding to 0 and $\bot$. Thus, we have

$$[\![E_1(t, 1)]\!](q) = \begin{cases} 1 & \text{if } [\![t]\!](q) = 1_{\mathrm{t}}, \\ 0 & \text{otherwise.} \end{cases}$$

In other words, E_1 acts as a deny-by-default operator. Similarly

$$[\![E_2E_1E_2(t, 0)]\!](q) = \begin{cases} 0 & \text{if } [\![t]\!](q) = 1_{\mathrm{t}}, \\ 1 & \text{otherwise.} \end{cases}$$

and $E_2E_1E_2$ acts as an allow-by-default (unary) operator.

The above observations suggest that Jobe's operators make an intuitively reasonable set of operators on which to base a 3-valued authorization language. Thus PTaCL(E) is functionally complete, its operators have intuitively reasonable interpretations, and it is canonically complete. We now illustrate why having a normal form may make it simpler to construct ABAC policies. Specifically, we represent the operator $\oplus_2$ from Figure 1 in normal form using Jobe's operators. To reiterate, it is impossible to represent $\oplus_2$ as any combination of XACML operators and it is difficult to see how to express $\oplus_2$ using the PTaCL operators (although it is theoretically possible to do so). We first represent the operator as a truth table:

x	y	$x \oplus_2 y$
0	0	0
0	$\bot$	0
0	1	$\bot$
$\bot$	0	0
$\bot$	$\bot$	$\bot$
$\bot$	1	$\bot$
1	0	$\bot$
1	$\bot$	$\bot$
1	1	1

By first representing $\oplus_2$ as a truth table, it is easy to establish that it is equivalent to

$$S_3^{\bot}(x, y) \vee_{\mathrm{e}} S_5^{\bot}(x, y) \vee_{\mathrm{e}} S_6^{\bot}(x, y) \vee_{\mathrm{e}} S_7^{\bot}(x, y) \vee_{\mathrm{e}} S_8^{\bot}(x, y) \vee_{\mathrm{e}} S_9^{1}(x, y).$$

Moreover:

$$S_3^{\bot}(x, y) \equiv S_1^{\bot}(x) \wedge_{\mathrm{e}} S_3^{\bot}(y) \qquad S_5^{\bot}(x, y) \equiv S_2^{\bot}(x) \wedge_{\mathrm{e}} S_2^{\bot}(y)$$
$$S_6^{\bot}(x, y) \equiv S_2^{\bot}(x) \wedge_{\mathrm{e}} S_3^{\bot}(y) \qquad S_7^{\bot}(x, y) \equiv S_3^{\bot}(x) \wedge_{\mathrm{e}} S_1^{\bot}(y)$$
$$S_8^{\bot}(x, y) \equiv S_3^{\bot}(x) \wedge_{\mathrm{e}} S_2^{\bot}(y) \qquad S_9^{1}(x, y) \equiv S_3^{1}(x) \wedge_{\mathrm{e}} S_3^{1}(y)$$
$$S_1^{\bot}(x) \equiv E_1E_2(x) \wedge_{\mathrm{e}} E_1E_2E_1(x) \qquad S_2^{\bot}(x) \equiv x \wedge_{\mathrm{e}} E_2(x)$$
$$S_3^{\bot}(x) \equiv E_2E_1(x) \wedge_{\mathrm{e}} E_1(x) \qquad S_3^{1}(x) \equiv E_1E_2(x) \wedge_{\mathrm{e}} E_2(x)$$

Hence, we can derive a formula in normal form for $\oplus_2$.

Of course, one would not usually construct the normal form by hand, as we have done above. Indeed, it is easy to develop an algorithm that would construct the normal form of a policy from its decision table. Thus, we have identified the formal foundations for a policy authorization language in which we can automate the construction of a policy in normal form, given the decision table for that policy.

Moreover, since our language is a tree-structured language, having exactly the same operational semantics as XACML and PTaCL, we can implement Jobe's operators as (custom) XACML combining algorithms and then specify XACML policies using these operators. Thus we can readily obtain a functionally and canonically complete policy language, whose policies can be embedded in the rich framework for ABAC provided by XACML (in terms of its languages for representing targets and requests) and its enforcement architecture (in terms of the policy enforcement, policy decision and policy administration points).[7]

Using the structure and evaluation strategy for PTaCL policies and the operators $\wedge_{\mathrm{e}}$, E_1 and E_2 makes it possible to define arbitrary policies and to represent them in normal form. We believe that this provides a number of advantages, in addition to those mentioned above, which we now briefly discuss. First, it is known that policy misconfigurations can be costly, both in terms of data leakage (when actions that should not be possible are authorized by the policy) and in terms of administration (when actions that should be possible are not authorized and the policy needs to be updated) [1]. We believe that the use of a canonically complete policy language is likely to make policy specification easier to understand for policy authors, thereby reducing the number of errors and policy misconfigurations. Future work will investigate whether this conjecture holds.

[6] It is easy to establish that the PTaCL formula $\sim x$ is equivalent to the PTaCL(E) formula $x \wedge_{\mathrm{e}} E_1(x)$. In other words, it is far more straightforward to represent the PTaCL operators using the operators in Jobe's logic, rather than representing the operators in Jobe's logic using the PTaCL operators (compare equation (2)).

[7] This is contrast to proposals in the literature, which require the use of non-standard components, such as multi-terminal binary decision diagrams [12] or non-deterministic finite automata [8].

Second, policies in normal form may be more efficient to evaluate. Given a formula in a 3-valued logic expressed in normal form, any literal that evaluates to 0 causes the entire clause to evaluate to 0, while any clause evaluating to 1 means the entire formula evaluates to 1. We may also be able to apply some of the equivalences described by Jobe to minimize the size of a formula in normal form, thereby further reducing the effort required to evaluate it. We hope to investigate this idea further in future work.

6. CONCLUDING REMARKS

The use of attribute-based access control languages such as XACML continues to rise as the demand for collaboration between industry partners becomes increasingly important. Furthermore, the widespread distribution of users and resources prompts a move away from traditional identity based authorization languages. This paper focuses on the way in which decisions (and hence authorization policies) are combined in ABAC authorization languages.

We analyzed the XACML rule- and policy-combining algorithms and identified various shortcomings of these algorithms. First, there is significant duplication and redundancy in the combining algorithms, with only two of the specified algorithms (do and po) required to express all of the XACML combining algorithms. Second, the XACML operators are not functionally complete; indeed, there are many useful unary and binary operators that cannot be represented using the XACML operators. We noted that PTaCL is functionally complete and, in particular, allows us to construct the XACML operators.

However, we argued that the way in which PTaCL (and XACML) policies must be written because of the underlying structure of the language is not particularly helpful to policy authors. Accordingly, we introduced the PTaCL(E) language that is both functionally and canonically complete. We believe that PTaCL(E) has considerable advantages over languages like PTaCL and XACML, and policy algebras. In particular, it is possible to simply read off a policy from a decision table and thus automate the specification of a policy from a description that will be intuitive and easy for the policy author to construct. We believe that this should help reduce the number of errors and policy misconfigurations, and hope to confirm this conjecture in future work. Moreover, unlike other approaches to enhancing the expressive power XACML [8, 11, 12], our proposed policy language requires no modification to XACML or additional processing. All that is required is the implementation (as custom XACML rule- and policy-combining algorithms) of the three policy operators we defined in Section 5. Again, this is something we plan to do in future work.

In addition, Jobe identifies a number of equivalences (between formulas in the logic E) that may be applied to reduce the size of a normal-form formula in E. We plan to investigate the relevance of these equivalences to authorization policies in future work. Another natural extension for future work is to analyze methods for handling errors in policy evaluation [4] and see how these methods can be extended to our canonically complete language.

Acknowledgements

Conrad Williams is a student in the Centre for Doctoral Training in Cyber Security, supported by EPSRC award EP/K035584/1.

7. REFERENCES

[1] Bauer, L., Garriss, S., and Reiter, M. K. Detecting and resolving policy misconfigurations in access-control systems. *ACM Trans. Inf. Syst. Secur. 14*, 1 (2011), 2.

[2] Bonatti, P. A., di Vimercati, S. D. C., and Samarati, P. An algebra for composing access control policies. *ACM Trans. Inf. Syst. Secur. 5*, 1 (2002), 1–35.

[3] Carminati, B., and Joshi, J., Eds. *SACMAT 2009, 14th ACM Symposium on Access Control Models and Technologies, Stresa, Italy, June 3-5, 2009, Proceedings* (2009), ACM.

[4] Crampton, J., and Huth, M. An authorization framework resilient to policy evaluation failures. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security* (2010), D. Gritzalis, B. Preneel, and M. Theoharidou, Eds., vol. 6345 of *Lecture Notes in Computer Science*, Springer, pp. 472–487.

[5] Crampton, J., and Morisset, C. PTaCL: A language for attribute-based access control in open systems. In *Principles of Security and Trust - First International Conference, POST 2012, Proceedings*, P. Degano and J. D. Guttman, Eds., vol. 7215 of *Lecture Notes in Computer Science*. Springer, 2012, pp. 390–409.

[6] Jobe, W. H. Functional completeness and canonical forms in many-valued logics. *The Journal of Symbolic Logic 27*, 04 (1962), 409–422.

[7] Kleene, S. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, NJ, 1950.

[8] Li, N., Wang, Q., Qardaji, W. H., Bertino, E., Rao, P., Lobo, J., and Lin, D. Access control policy combining: theory meets practice. In Carminati and Joshi [3], pp. 135–144.

[9] Łukasiewicz, J. Philosophische Bemerkungen zu mehrwertigen Systemen des Aussagekalküls. *Comtes rendus des séances de la Société des Sciences et des Lettres de Varsovie Classe III*, vol. 23 (1930), 55–57.

[10] Moses, T. eXtensible Access Control Markup Language (XACML) Version 2.0 OASIS Standard, 2005. http://docs.oasis-open.org/xacml/2.0/access-control-xacml-2.0-core-spec-os.pdf.

[11] Ni, Q., Bertino, E., and Lobo, J. D-algebra for composing access control policy decisions. In *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security* (2009), W. Li, W. Susilo, U. K. Tupakula, R. Safavi-Naini, and V. Varadharajan, Eds., ACM, pp. 298–309.

[12] Rao, P., Lin, D., Bertino, E., Li, N., and Lobo, J. An algebra for fine-grained integration of XACML policies. In Carminati and Joshi [3], pp. 63–72.

[13] Rissanen, E. eXtensible Access Control Markup Language (XACML) Version 3.0 OASIS Standard, 2012. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-os-en.html.

[14] Wijesekera, D., and Jajodia, S. A propositional policy algebra for access control. *ACM Trans. Inf. Syst. Secur. 6*, 2 (2003), 286–325.

APPENDIX

A. CONSTRUCTIBLE XACML BINARY OPERATORS

We now consider which binary operators can be constructed using the XACML operators. We will write $+$ and $-$ to denote pbd and dbd, respectively, in order to simplify the notation. There are four possible idempotent, $\cup$-operators that can be constructed using the XACML operators: for all $x \in D$, $x \oplus x$, $x \oplus \bot$ and $\bot \oplus x$ are pre-determined; only $0 \oplus 1$ and $1 \oplus 0$ may vary. These operators are do, po, fa and what we might call "last-applicable" (la).[8] We have $r_1 \,\mathsf{la}\, r_2 \equiv r_2 \,\mathsf{fa}\, r_1$, so we can construct each of these operators using the XACML operators. The commutative, idempotent $\cup$-operators are do and po.

We now consider operators having the general form $\diamond_1((\diamond_2 d_1) \oplus (\diamond_3 d_2))$ where $\diamond_1, \diamond_2, \diamond_3 \in \{-, +, \text{“”}\}$ (“” is used to denote that the unary operator is omitted) and $\oplus \in \{\mathsf{do}, \mathsf{po}, \mathsf{fa}, \mathsf{la}\}$. If either $\diamond_2$ or $\diamond_3$ are $-$ or $+$, the application of $\diamond_1$ has no effect as the operator $(\diamond_2 d_1) \oplus (\diamond_3 d_2)$ will be conclusive (since $\diamond d \in \{0, 1\}$ and $x \oplus \bot = \bot \oplus x = x$ for any $\oplus \in \{\mathsf{do}, \mathsf{po}, \mathsf{fa}, \mathsf{la}\}$ and any $x \in \{0, 1\}$). This has the effect of limiting the number of possible operators of this form. The possible choices for $\diamond_1, \diamond_2, \diamond_3$ and $\oplus$ are tabulated in Table 2, which results in 44 quasi-idempotent operators.

$\diamond_1$	$\diamond_2$	$\diamond_3$	$\oplus$	Possible Ops
“”	$+$	3	4	12
“”	$-$	3	4	12
“”	“”	2	4	8
3	“”	“”	4	12

Table 2: Choices for $\diamond_1, \diamond_2, \diamond_3$ and $\oplus$

However, not all of these operators are unique, given the following equivalences between operators.

PROPOSITION 4. *For any $x, y \in \{0, 1, \bot\}$,*

$$(-x) \,\mathsf{po}\, y = x \,\mathsf{po}\, (-y) = -(x \,\mathsf{po}\, y) = (-x) \,\mathsf{po}\, (-y);$$
$$(+x) \,\mathsf{po}\, y = (+x) \,\mathsf{po}\, (-y);$$
$$x \,\mathsf{po}\, (+y) = (-x) \,\mathsf{po}\, (+y);$$
$$(+x) \,\mathsf{do}\, y = x \,\mathsf{do}\, (+y) = +(x \,\mathsf{do}\, y) = (+x) \,\mathsf{do}\, (+y);$$
$$(-x) \,\mathsf{do}\, y = (-x) \,\mathsf{do}\, (+y);$$
$$x \,\mathsf{do}\, (-y) = (+x) \,\mathsf{do}\, (-y).$$

These results follow by inspection of the relevant decision tables. The intuition behind the first three results is that $0 \,\mathsf{po}\, x = \bot \,\mathsf{po}\, x$ for all $x \in \{0, 1\}$; an analogous observation holds for the second three.

Then we can construct the following operators using do and different combinations of the unary operators $-$ and $+$:

$$x \,\mathsf{do}_0\, y \stackrel{\text{def}}{=} x \,\mathsf{do}\, y \qquad x \,\mathsf{do}_1\, y \stackrel{\text{def}}{=} (-x) \,\mathsf{do}\, (-y)$$
$$x \,\mathsf{do}_2\, y \stackrel{\text{def}}{=} (-x) \,\mathsf{do}\, y \qquad x \,\mathsf{do}_3\, y \stackrel{\text{def}}{=} x \,\mathsf{do}\, (-y)$$
$$x \,\mathsf{do}_4\, y \stackrel{\text{def}}{=} -(x \,\mathsf{do}\, y) \qquad x \,\mathsf{do}_5\, y \stackrel{\text{def}}{=} +(x \,\mathsf{do}\, y)$$

[8]Although fa (and hence la) is a redundant operator, we continue their use as a compact way of expressing operators (and later families of operators) instead of the lengthy expressions using do and po.

These operators comprise what we call the deny-overrides family of operators. These operators are all distinct and operate on $\{0, 1\}$ in exactly the same way as do. Moreover,

$$0 \,\mathsf{do}_i\, \bot = \bot \,\mathsf{do}_i\, 0 = 0$$

for all i. They differ in their effect on elements in $\{1, \bot\}$, as shown in Figure 11. Notice that do_5 is equivalent to three other operators (by Remark 4). Note that do_2 and do_3 are not commutative.

do_0	1	$\bot$
1	1	1
$\bot$	1	$\bot$

do_1	1	$\bot$
1	1	0
$\bot$	0	0

do_2	1	$\bot$
1	1	1
$\bot$	0	0

do_3	1	$\bot$
1	1	0
$\bot$	1	0

do_4	1	$\bot$
1	1	1
$\bot$	1	0

do_5	1	$\bot$
1	1	1
$\bot$	1	1

Figure 11: The family of deny-overrides operators

Analogously, we can define a family of six permit-overrides operators which act on $\{1, \bot\}$ in exactly the same way as the deny-overrides operators in Figure 11. Therefore, in total, we can construct six quasi-idempotent deny-overrides operators and six quasi-idempotent permit-overrides operators, of which one is idempotent and four are commutative.

In a similar manner we can identify the first-applicable and last-applicable families, consisting of operators obtained using fa and la respectively. We observe the following equivalences between operators.

PROPOSITION 5. *For any $x, y \in \{0, 1, \bot\}$, $\diamond \in \{-, +\}$,*

$$(\diamond x) \,\mathsf{fa}\, y = (\diamond x) \,\mathsf{fa}\, (-y) = (\diamond x) \,\mathsf{fa}\, (+y);$$
$$-(x \,\mathsf{fa}\, y) = x \,\mathsf{fa}\, (-y);$$
$$+(x \,\mathsf{fa}\, y) = x \,\mathsf{fa}\, (+y);$$
$$x \,\mathsf{la}\, (\diamond y) = (-x) \,\mathsf{la}\, (\diamond y) = (+x) \,\mathsf{la}\, (\diamond y);$$
$$-(x \,\mathsf{la}\, y) = (-x) \,\mathsf{la}\, y;$$
$$+(x \,\mathsf{la}\, y) = (+x) \,\mathsf{fa}\, y.$$

These results follow by inspection of the relevant decision tables. Unlike the deny-overrides and permit-overrides families, we obtain only five distinct operators for the first/last-applicable families. This is clear from the restrictions placed on the decision tables, and follows immediately from the equivalences in Remark 4. Thus, in total, the 44 possible operators actually represent 22 distinct binary operators. The 44 operators and their duplicate forms are tabulated in Table 3.

Thus far, we only considered operators having the form

$$\diamond_1((\diamond_2 d_1) \oplus (\diamond_3 d_2)).$$

It is not obvious that these are the only forms that yield new, distinct binary operators. These forms only contain single instances of each decision variable d_1 and d_2, which raises the question of whether new operators can be constructed from forms which contain multiple instances of d_1 and d_2. Recall the definition of fa ($x \,\mathsf{fa}\, y \equiv x \,\mathsf{po}\, (x \,\mathsf{do}\, y)$), which is constructed using more than one instance of x. We now investigate whether the inclusion of multiple instances of d_1 and d_2 yields any further operators.

Op	Construction	Alternative forms
do_0	$x\ \mathsf{do}\ y$	
do_1	$(-x)\ \mathsf{do}\ (-y)$	
do_2	$(-x)\ \mathsf{do}\ y$	$(-x)\ \mathsf{do}\ (+y)$
do_3	$x\ \mathsf{do}\ (-y)$	$(+x)\ \mathsf{do}\ (-y)$
do_4	$-(x\ \mathsf{do}\ y)$	
do_5	$+(x\ \mathsf{do}\ y)$	$(+x)\ \mathsf{do}\ (+y), (+x)\ \mathsf{do}\ y, x\ \mathsf{do}\ (+y)$
po_0	$x\ \mathsf{po}\ y$	
po_1	$(+x)\ \mathsf{po}\ (+y)$	
po_2	$(+x)\ \mathsf{po}\ y$	$(+x)\ \mathsf{po}\ (-y)$
po_3	$x\ \mathsf{po}\ (+y)$	$(-x)\ \mathsf{po}\ (+y)$
po_4	$-(x\ \mathsf{po}\ y)$	
po_5	$+(x\ \mathsf{po}\ y)$	$(-x)\ \mathsf{po}\ (-y), (-x)\ \mathsf{do}\ y, x\ \mathsf{do}\ (-y)$
fa_0	$x\ \mathsf{fa}\ y$	
fa_1	$(-x)\ \mathsf{fa}\ y$	$(-x)\ \mathsf{fa}\ (-y), (-x)\ \mathsf{fa}\ (+y)$
fa_2	$(+x)\ \mathsf{fa}\ y$	$(+x)\ \mathsf{fa}\ (-y), (+x)\ \mathsf{fa}\ (+y)$
fa_3	$-(x\ \mathsf{fa}\ y)$	$x\ \mathsf{fa}\ (-y)$
fa_4	$+(x\ \mathsf{fa}\ y)$	$x\ \mathsf{fa}\ (+y)$
la_0	$x\ \mathsf{la}\ y$	
la_1	$x\ \mathsf{la}\ (-y)$	$(-x)\ \mathsf{la}\ (-y), (+x)\ \mathsf{la}\ (-y)$
la_2	$x\ \mathsf{la}\ (+y)$	$(-x)\ \mathsf{la}\ (+y), (+x)\ \mathsf{la}\ (+y)$
la_3	$-(x\ \mathsf{la}\ y)$	$(-x)\ \mathsf{la}\ y$
la_4	$+(x\ \mathsf{la}\ y)$	$(+x)\ \mathsf{la}\ y$

Table 3: Operator constructions and alternative forms

To answer this question, we developed program with the aim of enumerating all constructible binary operators by brute force. The program works by generating all operators that can be created by combining other operators. The program generates all binary operators which have the general form $\diamond x \oplus \triangle y$ where $\diamond, \triangle \in \{-, +, \text{“”}\}$ and $\oplus \in \{\mathsf{do}, \mathsf{po}\}$. Note we omit fa and la from the set of binary operators, as these operators (being expressible in terms of do and po) will be generated automatically as we recursively create operators. We initialize the array variables $x = [0, 0, 0, 1, 1, 1, \bot, \bot, \bot]$ and $y = [0, 1, \bot, 0, 1, \bot, 0, 1, \bot]$. We store the decision table of a binary operator in a similar array variable. We generate the $3 \times 2 \times 3 = 18$ decision tables for operators of the form $\diamond x \oplus \triangle y$, of which 8 are duplicates. We remove the duplicates, storing the decision tables for the remaining 12 operators in an array. The process is repeated with each item in the array being reused as an input for x and y in the general form of a binary operator. This second iteration generates $12^2 \times 18 = 2592$ operators, of which 22 are distinct operators. We once again reuse these operators as inputs for x and y, yielding the the same 22 distinct operators. As no new operators are generated, the program terminates. The 22 operators discovered via exhaustive search correspond exactly to the operators we constructed above. The decision tables for these operators are listed in Figure 12.

do_0	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	1	$\bot$

do_1	0	1	$\bot$
0	0	0	0
1	0	1	0
$\bot$	0	0	0

do_2	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	0	0

do_3	0	1	$\bot$
0	0	0	0
1	0	1	0
$\bot$	0	1	0

do_4	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	1	0

do_5	0	1	$\bot$
0	0	0	0
1	0	1	1
$\bot$	0	1	1

(a) Deny-overrides family

po_0	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	0	1	$\bot$

po_1	0	1	$\bot$
0	0	1	1
1	1	1	1
$\bot$	1	1	1

po_2	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	1	1	1

po_3	0	1	$\bot$
0	0	1	1
1	1	1	1
$\bot$	0	1	1

po_4	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	0	1	0

po_5	0	1	$\bot$
0	0	1	0
1	1	1	1
$\bot$	0	1	1

(b) Permit-overrides family

fa_0	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	1	$\bot$

fa_1	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	0	0

fa_2	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	1	1	1

fa_3	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	1	0

fa_4	0	1	$\bot$
0	0	0	0
1	1	1	1
$\bot$	0	1	1

(c) First-applicable family

la_0	0	1	$\bot$
0	0	1	0
1	0	1	1
$\bot$	0	1	$\bot$

la_1	0	1	$\bot$
0	0	1	0
1	0	1	0
$\bot$	0	1	0

la_2	0	1	$\bot$
0	0	1	1
1	0	1	1
$\bot$	0	1	1

la_3	0	1	$\bot$
0	0	1	0
1	0	1	1
$\bot$	0	1	0

la_4	0	1	$\bot$
0	0	1	0
1	0	1	1
$\bot$	0	1	1

(d) Last-applicable family

Figure 12: Constructible Binary Operators in XACML