Attribute Expressions, Policy Tables and Attribute-Based Access Control

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

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

ABSTRACT

Attribute-based access control (ABAC) has attracted considerable interest in recent years, prompting the development of the standardized XML-based language XACML. ABAC policies written in languages like XACML have a tree-like structure, where leaf nodes are associated with authorization decisions and non-leaf nodes are associated with decision-combining algorithms. However, it may be difficult in XACML to construct a given policy due to the tree-structured nature of XACML and the way in which combining algorithms are defined. Furthermore, there is limited control over how requests are evaluated with respect to targets.

In this paper, we introduce the notion of an attribute expression, which generalizes the notion of a target, and show how attribute expressions are used to specify policies in tabular form. We demonstrate why representing policies in this manner is convenient, intuitive and flexible for policy authors, and provide a method for automatically compiling policy tables into machine-enforceable policies. Thus, we bridge the gap between a policy representation that is convenient for end-users and a policy that can be enforced by a PDP. We then describe various methods to reduce the size of policy tables.

In addition, we compare our language with XACML, highlighting various shortcomings of XACML and demonstrating how to express XACML policies in a tabular form. We then show how policy tables can be used as leaf nodes in a tree-structured language, providing a modular method for constructing enterprise-wide policies. Finally, we show how attribute expressions and policy tables can be used to make role-based access control and access control lists "attribute-aware".

KEYWORDS

Attribute-based access control; attribute expressions; AEPL; policy tables; XACML; PTACL

ACM Reference format:
Jason Crampton and Conrad Williams. 2017. Attribute Expressions, Policy Tables and Attribute-Based Access Control. In *Proceedings of SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA, ,* 12 pages.
DOI: http://dx.doi.org/10.1145/3078861.3078865

1 INTRODUCTION

Access control restricts the interactions that are possible between users (or programs operating under the control of users) and sensitive resources, and is an essential security service in any multi-user computing system. The most common means of implementing access control is to define an *authorization policy*, specifying which *requests* (that is, attempted user-resource interactions) are authorized and can thus be allowed. In a typical implementation, all requests are intercepted and evaluated with respect to the policy by trusted software components, often known as the *policy enforcement point* and *policy decision point*, respectively.

Thus, in general terms, an authorization policy is a function $P : Q \to D$, where Q is the set of requests and D is the set of authorization decisions. We assume 0 and 1 belong to D, representing the "deny" and "allow" decisions, respectively. Traditionally, Q was modeled as a set of triples of the form (s, o, a), where s is a subject, o is an object, and a is an action: a subject represents an authenticated entity, an object represents a protected resource, and an action is the means by which the subject wishes to interact with the object.

In recent years, we have seen the emergence of *attribute-based* access control, in part to cater for open, distributed computing environments where it is not necessarily possible to authenticate all entities directly. Instead, subjects and objects are associated with attributes, requests are collections of attributes associated with the subjects and objects, and these attributes determine whether a request is authorized or not. Thus we may imagine representing a policy as a table in which columns are indexed by attributes, rows represent the presence or otherwise of the respective attribute in a request, with the final column in the table indicating the authorization decision associated with a particular collection of attributes.

A simple example is shown in Figure 1, where 1 indicates the presence of the attribute in the request and 0 indicates the attribute is absent; the dash indicates that the presence or otherwise of a particular attribute is irrelevant to the decision. Thus the first row of the table indicates that the deny decision (0) should be returned if attributes a_1 and a_2 are present in a request. If no row exists for a particular combination, then we assume that the decision is $\perp$ ("not-applicable"); that is, the policy is "silent" for such a request and does not return a conclusive decision. Thus, for example, the decision is $\perp$ if attribute a_1 is not present in the request.

While it is certainly convenient and intuitive to represent authorization policies in tabular form, this representation is not compatible with many languages that have been developed for writing attribute-based authorization policies. XACML [11], PTaCL [6] and PBel [5], for example, are "tree-structured" languages, where policies are, essentially, terms in a logic-based formalism. These terms

a_1	a_2	a_3	d
1	1	–	0
1	0	1	1

Figure 1: A simple policy table

may be represented by trees, in which leaf nodes are attribute-decision pairs and interior nodes are attribute-operator pairs.[1]

Figure 2 illustrates the policy $((a_1, \mathrm{do}), ((a_2, 0), (a_3, 1)))$. In this case, the operator do represents the "deny-overrides" operator. A request is evaluated by first pruning the nodes that are not matched by the attributes in the request. Then the decisions in the remaining leaf nodes are combined using the policy combining operator(s). If, for example, all attributes are present in a request (so no pruning is performed), then the resulting decision is $0 \mathbin{\mathrm{do}} 1 = 0$, corresponding to the first line in Figure 1.

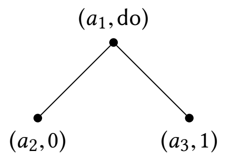

Figure 2: A simple tree-structured policy

In fact, the tree in Figure 2 represents an equivalent policy to the one tabulated in Figure 1, although this is not immediately apparent. It is this gap – between (i) how one is likely to conceive of a policy, and (ii) how one must construct the policy using existing languages for attribute-based access control – that provides the motivation for the work in this paper.

A further shortcoming of existing work on languages for attribute-based access control is the way in which requests and attributes are matched. Suppose we have an attribute name-value pair (n, v) and a request that contains multiple name-value pairs, including (n, v) and (n, v'), where $v' \neq v$. Then one might argue the request matches the attribute (since it contains (n, v)); on the other hand, one might argue it doesn't match the attribute (since it also contains (n, v')). XACML always assumes the former interpretation, which may be inappropriate if, for example, the policy author wishes to insist that the request contains exactly one name-value pair for the named attribute. Although PTaCL has a slightly more complex match semantics for requests and attributes, it ignores several possible match semantics that might be relevant in practice.

In this paper, we use prior work on canonical completeness to develop a new way of defining authorization policies using policy tables. In doing so, we support all possible match semantics for an attribute and request, thereby facilitating much greater control over policy specification. We then show how such tables can be automatically compiled into machine-enforceable policies. Thus, we are able to bridge the gap between a policy representation that is convenient for end-users and a policy that can be enforced by a PDP. We also demonstrate that policy tables can be used as the leaf nodes in a tree-structured language, thereby facilitating the modular construction of enterprise-wide policies. Finally, we show how such policy tables can be used to enhance existing access control paradigms, such as access control lists and role-based access control, by making them "attribute-aware".

[1] This is something of a simplification, but is a good approximation of how such policies are structured.

In summary, the main contributions of this paper are:

- the introduction of "attribute expressions" and match semantics for attribute based requests;
- the specification of a new policy authorization language AEPL, which represents policies as tables and provides a method for automatically converting policy tables into machine-enforceable policies;
- an overview of various policy compression methods which can be applied to AEPL policy tables;
- a method for converting tree-structured XACML policies into policy tables; and
- a demonstration of the various applications of AEPL, in enhancing existing paradigms such as access control lists and role-based access control.

In the following section we provide a brief introduction to canonical completeness for lattice-based logics, along with the specification of a 4-valued canonically complete logic [8], which we will use as the underlying logic for AEPL. In Section 3 we define the syntax and semantics of attribute expressions, along with a justification for why they are preferable to the use of traditional targets (found in XACML [11] and PTaCL [6]). Then, we specify the AEPL language, showing how policies are constructed as tables, and describe a method for automatically converting these tables into machine-enforceable policies. In Section 4, we investigate various methods for reducing the size of AEPL policy tables. We then discuss the limitations of targets in XACML, and demonstrate how an XACML policy can be converted into a policy table in Section 5. Section 6 discusses methods to build complex policies, enabling distributed specification and evaluation of policy tables, and ways to integrate policy tables with role-based access control and access control lists. We conclude the paper with a summary of our contributions and suggest ideas for future work.

2 BACKGROUND AND RELATED WORK

We focus our attention on 4-valued logics (and policy languages), where the set of decisions D is equal to $\{0, 1, \bot, \top\}$, corresponding to the authorization decisions "deny", "allow", "not-applicable" and "conflict", respectively [5, 10, 13]. In the context of decisions that arise from policy evaluation, we interpret 0 as a "deny" decision, 1 as "allow", $\bot$ as "not-applicable" and $\top$ as "conflict". We assume D is furnished with a partial ordering $\leqslant$ such that $(D, \leqslant)$ is a lattice, perhaps the most well-known logic of this kind being Belnap logic [4].

We begin with a brief introduction of Belnap logic in Section 2.1. Then, we summarize recent work by Crampton and Williams [8] in Section 2.2, which extends Jobe's [9] notions of canonical suitability, selection operators and canonical completeness to lattice-based logics, and their applications to attribute-based policy authorization languages.

We conclude the related work section with a description of a lattice-based 4-valued canonically complete logic [8]. (For discussion on the use of a 4-valued decision set and lattice-based logics in access control the reader is the referred to the literature [5, 10, 13])

2.1 Belnap logic

Belnap logic [4] is one of the most well-known lattice-based logics, and has been used as the formal basis for authorization languages such as PBel [5], Rumpole [10] and BelLog [13]. It was developed with the intention of defining ways to handle inconsistent and incomplete information in a formal manner and uses the truth values 0, 1, $\bot$, and $\top$, representing "false", "true", "lack of information" and "too much information", respectively. In the remainder of this paper, we will denote the four valued decision set $\{\bot, 0, 1, \top\}$ by 4.

The set of truth values in Belnap's logic admits two orderings: a truth ordering $\leqslant_t$ and a knowledge ordering $\leqslant_k$. In the truth ordering $\leqslant_t$, 0 is the minimum element and 1 is the maximum element, while $\bot$ and $\top$ are incomparable indeterminate values. In the knowledge ordering $\leqslant_k$, $\bot$ is the minimum element, $\top$ is the maximum element while 0 and 1 are incomparable. Both $(4, \leqslant_t)$ and $(4, \leqslant_k)$ are lattices, forming the interlaced bilattice illustrated in Figure 3.

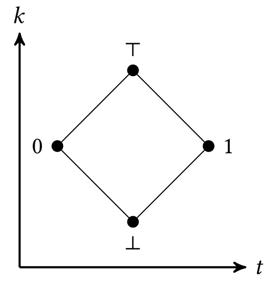

Figure 3: The 4 truth values in Belnap logic

We will assume the use of the knowledge ordering throughout this paper. The intuition is that $\bot$ ("not-applicable") represents no conclusive decision is possible because of lack of knowledge, while 0 and 1 are (incomparable) conclusive decisions, and $\top$ corresponds to too much knowledge because of conflicting conclusive decisions.

2.2 Canonical completeness

We now define canonical suitability, functional completeness, selection operators, normal form and canonical completeness for lattice-based logics [8].

Let $L = (V, \mathrm{Ops})$ be a logic associated with a lattice $(V, \leqslant)$ of truth values and a set of logical operators Ops. We omit V and Ops when no ambiguity can occur. We write $\Phi(L)$ to denote the set of formulae that can be written in the logic L.

We say L is *canonically suitable* if and only if there exist formulae $\phi_{\max}$ and $\phi_{\min}$ of arity 2 in $\Phi(L)$ such that $\phi_{\max}(x, y)$ returns $\sup\{x, y\}$ and $\phi_{\min}(x, y)$ returns $\inf\{x, y\}$. If a logic is canonically suitable, we will write $\phi_{\max}(x, y)$ and $\phi_{\min}(x, y)$ using infix binary operators as $x \curlyvee y$ and $x \curlywedge y$, respectively.

A function $f : V^n \to V$ is completely specified by a truth table containing n columns and m^n rows. However, not every truth table can be represented by a formula in a given logic $L = (V, \mathrm{Ops})$. L is said to be *functionally complete* if for every function $f : V^n \to V$, there is a formula $\phi \in \Phi(L)$ of arity n whose evaluation corresponds to the truth table. In this sense, XACML is not functionally complete [7], while PTaCL [6] and PBel [5] are.

Let $\underline{v}$ denote the minimum value in $(V, \leqslant)$. (Such a value must exist in a finite lattice.) We will write $\mathbf{a}$ to denote the tuple $(a_1, \dots, a_n) \in V^n$ when no confusion can occur. Then, for $\mathbf{a} \in V^n$, the n-ary *selection operator* $S^j_{\mathbf{a}}$ is defined as follows:

$$S^j_{\mathbf{a}}(\mathbf{x}) = \begin{cases} j & \text{if } \mathbf{x} = \mathbf{a}, \\ \underline{v} & \text{otherwise.} \end{cases}$$

Note $S^{\underline{v}}_{\mathbf{a}}(x) = \underline{v}$ for all $\mathbf{a}, \mathbf{x} \in V^n$. Illustrative examples of unary and binary selection operators (for Belnap logic) are shown in Figure 4.

x	$S^0_0(x)$	$S^\top_1(x)$
$\bot$	$\bot$	$\bot$
0	0	$\bot$
1	$\bot$	$\top$
$\top$	$\bot$	$\bot$

		y			
	$S^0_{(1,\top)}(x,y)$	$\bot$	0	1	$\top$
	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
x	0	$\bot$	$\bot$	$\bot$	$\bot$
	1	$\bot$	$\bot$	$\bot$	0
	$\top$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 4: Examples of selection operators in Belnap Logic

Selection operators play a central role in the development of canonically complete logics because an arbitrary function $f : V^n \to V$ can be expressed in terms of selection operators. Consider, for example, the function

$$f(x, y) = \begin{cases} \top & \text{if } x = 0, y = 1, \\ 0 & \text{if } x = y = 0, \\ 1 & \text{if } x = 1, y = \bot, \\ \bot & \text{otherwise.} \end{cases}$$

Then it is easy to confirm that

$$f(x, y) \equiv S^\top_{(0,1)}(x, y) \curlyvee S^0_{(0,0)}(x, y) \curlyvee S^1_{(1,\bot)}(x, y).$$

Moreover, $S^c_{(a,b)}(x, y) \equiv S^c_a(x) \curlywedge S^c_b(y)$ for any $a, b, c, x, y \in V$. Thus,

$$f(x, y) \equiv (S^\top_0(x) \curlywedge S^\top_1(y)) \curlyvee (S^0_0(x) \curlywedge S^0_0(y)) \curlyvee (S^1_1(x) \curlywedge S^1_\bot(y))$$

In other words, we can express f as the "disjunction" ($\curlyvee$) of "conjunctions" ($\curlywedge$) of unary selection operators.

More generally, 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

$$A = \{\mathbf{a} \in V^n : f(\mathbf{a}) > \bot\};$$

then, for all $\mathbf{x} \in V^n$,

$$f(\mathbf{x}) = \bigcurlyvee_{\mathbf{a} \in A} S^{f(\mathbf{x})}_{\mathbf{a}}(\mathbf{x}).$$

Jobe established a number of results connecting the functional completeness of a logic with the unary selection operators, summarized in the following theorem.

Theorem 2.1 (Jobe [9, 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 *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 $\vee$ and $\wedge$;
- no binary operator is included in the scope of a unary operator;
- no instance of $\vee$ occurs in the scope of the $\wedge$ operator.

A canonically suitable logic is *canonically complete* if every unary selection operator can be expressed in normal form. Crampton and Williams showed that there are considerable advantages to using a canonically complete logic as the basis for an authorization language [7]. In particular, it will often be easier to specify policies using a language based on a canonically complete logic and such policies can be compiled automatically into a normal form.

2.3 A 4-valued canonically complete logic

While Belnap logic is known to be functionally complete [3], Crampton and Williams [8] showed it is not canonically complete, essentially because only one unary operator (negation $\neg$) is defined. Crampton and Williams developed a new set of operators based on the knowledge ordering $(4, \leqslant_k)$, which produced a canonically complete 4-valued logic. For brevity, henceforth we denote the lattice $(4, \leqslant_k)$ by 4_k.

They defined two new unary operators $-$ and $\diamond$ whose truth tables are shown in Figure 5a: $-$ swaps the values of $\bot$ and $\top$; $\diamond$ permutes the values $\bot, 0, 1$ and $\top$. In addition, they reused the operator $\otimes$ which acts as the meet operation ($\wedge$) for the lattice 4_k, whose truth table is shown in Figure 5b. We represent this logic using the notation $L(4_k, \{-, \diamond, \otimes\})$.

d	$-d$	$\diamond d$
$\bot$	$\top$	0
0	0	1
1	1	$\top$
$\top$	$\bot$	$\bot$

(a) $-$ and $\diamond$

$\otimes$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	$\bot$	0
1	$\bot$	$\bot$	1	1
$\top$	$\bot$	0	1	$\top$

(b) $\otimes$

Figure 5: Canonically complete operators

Crampton and Williams [8] used the logic $L(4_k, \{-, \diamond, \otimes\})$ as a basis for the policy authorization language $\text{PTaCL}_4^{\leqslant}$, highlighting the numerous advantages that can be gained in using a canonically complete logic as the underlying logic, opposed to other languages such as PBel [5] which use the canonically incomplete Belnap logic. Accordingly, we will use the logic $L(4_k, \{-, \diamond, \otimes\})$ as the underlying logic when we develop our new policy authorization language.

3 THE AEPL LANGUAGE

As we noted in the introduction, an authorization policy may be represented as a function $P : Q \to D$, where Q is the set of requests and D is the set of decisions. In other words, $P(q)$ represents the result of evaluating policy P for request q, thereby determining whether q is authorized by policy P.

In attribute-based access control, a *request* is assumed to be a set of name-value pairs, where a name is an attribute and a value is taken from some domain over which the binary relations $=$, $\neq$, $<$, $\leqslant$, $>$ and $\geqslant$ are defined. We assume it is possible to determine whether a pair of values belongs to a given relation efficiently. In particular, we assume henceforth that all attribute values are strings of bounded length defined over some alphabet Σ.

It is usually impossible to specify an attribute-based policy P by specifying a decision for every possible request, given the size of the domain of P. Thus, it is usual to specify P in terms of the relationship between a policy and the attribute name-value pairs that constitute a request. This relationship is typically expressed in terms of "targets", which are predicates specified in terms of attributes values and whose truth values are determined by comparing the attribute values specified in the target with those present in the request.

3.1 Attribute expressions

Our attribute-expression policy language, AEPL, is based on the idea of an *attribute expression.* Informally, the input to P is a tuple of logical values, and those values are determined by "matching" a request to a set of attribute expressions. More formally, we define an *attribute expression* to be a tuple $(n, v, \sim, \oplus)$, where n is an attribute name, v is an attribute value or regular expression, $\sim$ is an associative, commutative, binary relation, and $\oplus$ is a binary operator.

Given a binary relation R defined over some domain V, we write $\overline{R}$ to denote the complement of R: that is $(a, b) \in \overline{R}$ iff $(a, b) \notin R$. We will usually write R as an infix relation $\sim$ and $\overline{R}$ as $\nsim$. Typical examples of R and $\overline{R}$ include $=$ and $\neq$, $<$ and $\not<$ (that is, $\geqslant$).

In many cases, v will be an attribute value and $\sim$ will be a comparison operator, such as equality or greater-than. However, the use of regular expressions means that more complex attribute expressions may be defined. Given a regular expression e (defined over Σ), let $\mathcal{L}(e)$ denote the set of strings that match e. Then we define $\sim_e$ to be the set of pairs $\{(e, w) : w \in \mathcal{L}(e)\}$. Moreover, for any regular expression e, there exists a regular expression $\overline{e}$, the *complement* of e, such that $w \notin \mathcal{L}(e)$ if and only if $w \in \mathcal{L}(\overline{e})$.

In the interests of clarity of exposition, we will assume henceforth that $\sim$ is always $=$ (corresponding to exact string matching). Note that this does not affect the generality of our approach: we only require that it is efficient to determine whether a pair belongs to the relation $\sim$.

The operator $\oplus$ determines the result of evaluating a request in which some name-value pairs match the attribute expression and some don't. (This contrasts with the approach taken in XACML and PTaCL.) We discuss this in more detail in Section 3.2.

We define three binary operators in Figure 6: $\vee$ and $\wedge$ are defined on the set $\{0, 1, \bot\}$; and ! is defined on $\{0, 1, \bot, \top\}$. Since $\oplus$ is associative and commutative, the expression

$$(\ldots((x_1 \oplus x_2) \oplus x_3) \oplus \cdots \oplus x_{k-1}) \oplus x_k)$$

may be written without ambiguity as $\bigoplus_{i=1}^{k} x_i$.

$\wedge$	$\bot$	0	1
$\bot$	$\bot$	0	1
0	0	0	0
1	1	0	1

$\vee$	$\bot$	0	1
$\bot$	$\bot$	0	1
0	0	0	1
1	1	1	1

!	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Figure 6: Binary operators for attribute expressions

3.2 Evaluating requests

A *request* is a set of name-value pairs of the form (n,v). The *evaluation* of a request $q = \{(n_1,v_1),\ldots,(n_\ell,v_\ell)\}$ with respect to an attribute expression $\alpha = (n,v,\sim,\oplus)$ is denoted by $\mathsf{eval}(q,\alpha)$. Informally, $\mathsf{eval}(q,\alpha)$ is determined by combining the results of evaluating the elements of the request (n_i,v_i) using $\oplus$. More formally, we define:

$$\mathsf{eval}(\emptyset,(n,v,\sim,\oplus)) = \bot_{\mathrm{m}};$$

$$\mathsf{eval}(\{(n',v')\},(n,v,\sim,\oplus)) = \begin{cases} 1_{\mathrm{m}} & \text{if } n = n' \text{ and } v \sim v', \\ 0_{\mathrm{m}} & \text{if } n = n' \text{ and } v \nsim v', \\ \bot_{\mathrm{m}} & \text{otherwise.} \end{cases}$$

We say a name-value pair (n',v') *matches* an attribute expression α if $\mathsf{eval}(\{(n',v')\},\alpha) = 1_{\mathrm{m}}$; and we say (n',v') *does not match* α if $\mathsf{eval}(\{(n',v')\},\alpha) = 0_{\mathrm{m}}$. Throughout this section, we use the subscript m (for "match") to denote explicitly logical values that arise from the evaluation of attribute expressions (request matches). In Section 3.3, we use the subscript d to denote logical values that arise from policy evaluation (authorization decisions). When no ambiguity can occur we will omit the subscripts.

A request $q = \{(n_1,v_1),\ldots,(n_\ell,v_\ell)\}$ may contain two name-value pairs, one of which matches $\alpha = (n,v,\sim,\oplus)$ and one which doesn't. The choice of operator $\oplus$ determines how the results of the matches will be combined. Formally, we have

$$\mathsf{eval}(q,(n,v,\sim,\oplus)) = \bigoplus_{i=1}^{k} \mathsf{eval}(\{(n_i,v_i)\},(n,v,\sim,\oplus)).$$

Thus, we have the following possibilities.

- $\mathsf{eval}(q,(n,v,\sim,\vee)) = 1_{\mathrm{m}}$ if there exists i such that $\mathsf{eval}((n_i,v_i),(n,v,\sim,\vee)) = 1_{\mathrm{m}}$; in other words, if the request contains at least one name-value pair that matches the attribute expression.
- $\mathsf{eval}(q,(n,v,\sim,\wedge)) = 0_{\mathrm{m}}$ if there exists i such that $\mathsf{eval}((n_i,v_i),(n,v,\sim,\wedge)) = 0_{\mathrm{m}}$; in other words, if the request contains at least one name-value pair that does not match the attribute expression.
- $\mathsf{eval}(q,(n,v,\sim,!)) = \top_{\mathrm{m}}$, indicating conflict, if there exist i and j such that $\mathsf{eval}((n_i,v_i),(n,v,\sim,!)) = 0_{\mathrm{m}}$ and $\mathsf{eval}((n_j,v_j),(n,v,\sim,!)) = 1_{\mathrm{m}}$.

It is worth noting that neither XACML nor PTaCL provide this level of control over how a request is evaluated with respect to a target (the concept analogous to an attribute expression). Roughly speaking, target evaluation in both languages only returns 0_{m} or 1_{m} and (effectively) always assumes the use of $\vee$ when a request contains attribute values that both match and don't match a target.

3.3 AEPL policies

A *policy* in AEPL is a pair $P = (A(P),F(P))$, where

- $A(P) = \{\alpha_1,\ldots,\alpha_\ell\}$ is a set of attribute expressions,
- $D_i \subseteq \{\bot_{\mathrm{m}},0_{\mathrm{m}},1_{\mathrm{m}},\top_{\mathrm{m}}\}$ is the range of values that $\mathsf{eval}(q,\alpha_i)$ can take, and
- $F : D_1 \times \cdots \times D_\ell \to \{\bot_{\mathrm{d}},0_{\mathrm{d}},1_{\mathrm{d}},\top_{\mathrm{d}}\}$ is a function.

Then we define:

$$P(q) = F(\mathsf{eval}(q,\alpha_1),\ldots,\mathsf{eval}(q,\alpha_\ell)).$$

We will write A and F for $A(P)$ and $F(P)$, respectively, when P is clear from context. We will also write $\mathsf{eval}(q,A)$ for $\mathsf{eval}(q,\alpha_1),\ldots,\mathsf{eval}(q,\alpha_\ell)$ where no confusion can occur.

We may visualize F as a table having $\ell + 1$ columns. The first ℓ columns are indexed by the attribute expressions in A. The entries in the ith column are the possible values that $\mathsf{eval}(q,\alpha_i)$ can take. The final entry in the row with entries $d_1,\ldots,d_\ell$ is $F(d_1,\ldots,d_\ell)$. In other words, policies are defined in the form suggested in the introduction and illustrated in Figure 1. Thus we specify an AEPL policy in two steps: define the relevant attribute expressions; and then define the policy table. Note that a policy is defined directly in terms of the match relationships that exist between the elements of a request and the policy's attribute expressions.

In the remainder of the paper, we use an example of a simple policy containing two attribute expressions. Let $P_{\mathrm{ex}} = (A_{\mathrm{ex}},F_{\mathrm{ex}})$ be a policy, where

$$A_{\mathrm{ex}} = \{\alpha_1,\alpha_2\} = \{(n_1,v_1,=,\wedge),(n_2,v_2,=,\wedge)\}$$

and F_{ex} is defined in Figure 7.

$x_1 = \mathsf{eval}(q,\alpha_1)$	$x_2 = \mathsf{eval}(q,\alpha_2)$	$P_{\mathrm{ex}}(x_1,x_2)$
$\bot_{\mathrm{m}}$	$\bot_{\mathrm{m}}$	$\bot_{\mathrm{d}}$
$\bot_{\mathrm{m}}$	0_{m}	$\bot_{\mathrm{d}}$
$\bot_{\mathrm{m}}$	1_{m}	1_{d}
0_{m}	$\bot_{\mathrm{m}}$	0_{d}
0_{m}	0_{m}	0_{d}
0_{m}	1_{m}	0_{d}
1_{m}	$\bot_{\mathrm{m}}$	1_{d}
1_{m}	0_{m}	0_{d}
1_{m}	1_{m}	1_{d}

Figure 7: Policy function defined as a table

The decisions in the final column of Figure 7 are determined by the policy author, for each combination of attribute expression matches. This allows for precise specification of how the policy P_{ex} should behave under each different attribute expression evaluation outcome and differs significantly from the evaluation of targets in XACML and PTaCL. We discuss this in more detail in Section 5.

3.4 Policies in normal form

Crampton and Williams [7] extended work by Jobe [9] to develop a method for converting arbitrary tables representing functions of the form $F : D^n \to D$, where D is the set of values in a multi-valued logic, into an equivalent logical formula using selection operators. We now show how this method can be applied to AEPL policies to generate standardized policy representations that can be evaluated automatically.

We write $\mathcal{D}$ to denote $D_1 \times \cdots \times D_\ell$. Given $(x_1, \dots, x_\ell) \in \mathcal{D}$, we will write $\mathbf{x}$ where no ambiguity can occur. For each row in the table representing F, we may construct an equivalent logical formula comprising a "disjunction" of selection operators. Specifically, if $F(\mathbf{a}) = d$ for $\mathbf{a} \in \mathcal{D}$, then, we may write this as $S^d_{\mathbf{a}}(\mathbf{x})$. Let $\mathcal{D}^+ = \{\mathbf{x} \in \mathcal{D} : F(\mathbf{x}) \neq \bot\}$. Then

$$F(\mathbf{x}) = \bigvee_{\mathbf{a} \in \mathcal{D}^+} S^d_{\mathbf{a}}(\mathbf{x}) \qquad \text{and} \qquad S^d_{(a_1,\dots,a_\ell)}(\mathbf{x}) \equiv \bigwedge_{i=1}^{\ell} S^d_{a_i}.$$

Hence F may be represented as a disjunction of conjunctions of unary selection operators [8].

Consider F_{ex}, and let $x_1 = \mathrm{eval}(q, \alpha_1)$ and $x_2 = \mathrm{eval}(q, \alpha_2)$. Then, we may express the policy $P_{\text{ex}}(q) = F_{\text{ex}}(x_1, x_2)$ as a disjunction of selection operators:

$$\begin{aligned} F_{\text{ex}}(x_1,x_2) \equiv\ & S^{\bot}_{(\bot,\bot)}(x_1,x_2) \vee S^{\bot}_{(\bot,0)}(x_1,x_2) \vee S^{1}_{(\bot,1)}(x_1,x_2) \\ & \vee S^{0}_{(0,\bot)}(x_1,x_2) \vee S^{0}_{(0,0)}(x_1,x_2) \vee S^{0}_{(0,1)}(x_1,x_2) \\ & \vee S^{1}_{(1,\bot)}(x_1,x_2) \vee S^{0}_{(1,0)}(x_1,x_2) \vee S^{1}_{(1,1)}(x_1,x_2). \end{aligned}$$

This, in turn, may be represented as a disjunction of conjunctions of unary selection operators (as described above).

Furthermore, Crampton and Williams have derived expressions for the unary selection operators in terms of the operators $\{-, \diamond, \otimes\}$ (see Appendix A). Hence, we can derive a formula in normal form for the policy $P_{\text{ex}} = (A_{\text{ex}}, F_{\text{ex}})$. Of course, one would not usually construct the normal form by hand, as we have done above. Indeed, Crampton and Williams [8] have developed an algorithm which takes an arbitrary policy expressed as a table as input, and outputs the equivalent normal form expressed in terms of the operators $\{-, \diamond, \otimes\}$.

3.5 AEPL policy trees

An AEPL *policy* is a pair (A, F). While this method of policy specification provides an intuitive and flexible method for defining policies, it will not scale to situations where many attribute expressions need to be specified and evaluated. In this case, it makes sense to use the policy-combining operators found in XACML and other tree-structured authorization languages to combine the results of evaluating multiple policies, each using a small number of attribute expressions. Thus, the set of attribute expressions in each policy will act in the same way as a target in a language like XACML.

In this section we revise the syntax and semantics for policy evaluation of the tree-structured ABAC language PTaCL$^{\leqslant}_4$ [8]. We use the operators $-, \diamond$ and $\otimes$ from PTaCL$^{\leqslant}_4$ (defined in Section 2.3) and demonstrate how policies of the form (A, F) can be used in tree-structured policies, and how we may replace targets with attribute expressions.

We define an *attribute expression based target*, or simply an *AE-target*, to be a pair (A, T), where $A = \{\alpha_1, \dots, \alpha_\ell\}$ is a set of attribute expressions and $T \subseteq D_1 \times \cdots \times D_\ell$, where D_i is the set of values that $\mathrm{eval}(q, \alpha_i)$ can take. If q is a request and T is an AE-target such that $\mathrm{eval}(q, A) \in T$ then q is said to *match* T.

Given an AEPL policy $P = (A, F)$, then P, $\diamond P$ and $-P$ are AEPL policy trees, where

$$(\diamond P)(q) \overset{\text{def}}{=} \diamond(P(q)) \qquad \text{and} \qquad (-P)(q) \overset{\text{def}}{=} -(P(q)).$$

If P_1 and P_2 are AEPL policy trees and T is an AE-target, then $(T, P_1 \otimes P_2)$ is a AEPL policy tree, where

$$(T, P_1 \otimes P_2)(q) \overset{\text{def}}{=} \begin{cases} P_1(q) \otimes P_2(q) & \text{if } \mathrm{eval}(q, A) \in T, \\ \bot & \text{otherwise.} \end{cases}$$

The ability to use AEPL policies (A, F) as leaf nodes (atomic policies) in AEPL policy trees provides us with a number of advantages (over PTaCL$^{\leqslant}_4$). We get the additional expressive power of policy specification for leaf nodes, together with the full power and functional completeness of PTaCL$^{\leqslant}_4$. By facilitating high-level operators in addition to specifying policies via a policy table and attribute expressions, we provide a hybrid means of constructing policies, which can be both bottom-up and top-down. This provides a great deal of flexibility and expressivity for policy authors. We can support low level policies specified by functions, and merge the policies using high-level policy operators. In addition, we have greater control of the applicability of policies due to the use of targets based on attribute expression (over "traditional" targets in PTaCL$^{\leqslant}_4$). We discuss PTaCL$^{\leqslant}_4$ targets and their limitations in more depth in Section 5.2.

4 POLICY COMPRESSION

While representing a policy P as a pair (A, F) is more concise and an easier task than specifying a decision for every possible request, the policy tables will be large when many attribute expressions are involved. To tackle this problem, we now investigate methods for policy compression, with the aim of reducing the size of these policy tables.

4.1 Removing redundancies

We begin with the following two remarks about methods for merging and omitting rows from policy tables.

Remark 1. *Suppose $F(a, x_2) = d$ for all $x_2 \in D_2$, as illustrated in the policy table fragment below.*

x_1	x_2	$P(x_1, x_2)$
a	$\bot$	d
a	0	d
a	1	d
a	$\top$	d

Then it is easy to show that

$$S^d_{(a,\bot)}(x_1, x_2) \vee S^d_{(a,0)}(x_1, x_2) \vee S^d_{(a,1)}(x_1, x_2) \vee S^d_{(a,\top)}(x_1, x_2)$$

is equivalent to $S^d_a(x_1)$. (This equivalence is formally established in Table 14 in Appendix B.) And this may be represented in tabular form as a single row, shown below, where we use – to signify that x_2 can take any value.

x_1	x_2	$P(x_1, x_2)$
a	–	d

Remark 2. *We may omit any rows from the policy table in which the final column contains the value $\bot$. Recall*

$$S^d_{\mathbf{a}}(\mathbf{x}) = \begin{cases} d & \text{if } \mathbf{x} = \mathbf{a}, \\ \bot & \text{otherwise.} \end{cases}$$

Moreover, $(\bot \vee x) = (x \vee \bot) = x$ for all $x \in \{0, 1, \bot, \top\}$. Thus

$$S^{d_1}_{\mathbf{a_1}}(\mathbf{x}) \vee S^{d_2}_{\mathbf{a_2}}(\mathbf{x}) \vee \ldots \vee S^{d_n}_{\mathbf{a_n}}(\mathbf{x}) = \bot,$$

except when $\mathbf{x} \in \{\mathbf{a_1}, \ldots, \mathbf{a_n}\}$.

Thus, we may assume the policy returns $\bot$ if the table does not contain an entry for a particular tuple $\mathbf{x}$. In this case, we say the policy is *not applicable* for any request q such that $\text{eval}(q, A) = \mathbf{x}$.

Returning to our example policy P_{ex}, we apply the results from the remarks above to reduce the size of the policy table which defines the function F_{ex}. First, note that

$$F_{\text{ex}}(0, \bot) = F_{\text{ex}}(0, 0) = F_{\text{ex}}(0, 1) = 0.$$

Thus, by Remark 1, we may merge these three rows into a single row, represented by $F_{\text{ex}}(0, -) = 0$. In addition, by Remark 2, we may omit the rows $F_{\text{ex}}(\bot, \bot)$ and $F_{\text{ex}}(\bot, 0)$ since they contain $\bot$ in the final column. Hence, we have the reduced policy table shown in Figure 8.

$x_1 = \text{eval}(q, \alpha_1)$	$x_2 = \text{eval}(q, \alpha_2)$	$P_{\text{ex}}(x_1, x_2)$
$\bot_{\text{m}}$	1_{m}	1
0_{m}	–	0
1_{m}	$\bot_{\text{m}}$	1
1_{m}	0_{m}	0
1_{m}	1_{m}	1

Figure 8: Reduced policy table

Expressing the policy $P_{\text{ex}}(q) = F_{\text{ex}}(x_1, x_2)$ as a disjunction of selection operators, we have

$$F_{\text{ex}}(x_1, x_2) \equiv S^1_{(\bot,1)}(x_1, x_2) \vee S^0_0(x_1) \vee S^1_{(1,\bot)}(x_1, x_2) \vee \\ S^0_{(1,0)}(x_1, x_2) \vee S^1_{(1,1)}(x_1, x_2).$$

We may apply Remark 1 directly during policy specification. If, for example, a policy author decides during the construction of a policy table that if $x_1 = 0$ then the value of x_2 is irrelevant, the policy should return 0 (the case in our example). In other words, we can, if desired, directly encode a deny-overrides or allow-overrides in the policy table when certain attribute expressions are matched or not matched. And we can allow the policy author to use – as syntactic sugar for a "decision" in the policy table, thereby saving the policy author from entering multiple rows (as seen in Figure 7).

4.2 Policies as Boolean functions

We now demonstrate that it is possible to reduce certain policies to Boolean functions. Specifically, if $F(\mathbf{x}) \in \{0, 1\}$ for all $\mathbf{x} \in \mathcal{D}$, then we can eliminate the values $\bot$ and $\top$ from the policy table. In particular, we may replace an attribute expression $\alpha = (n, v, \sim, \oplus)$ with two simpler attribute expressions $\alpha_1 = (n, v, \sim)$ and $\alpha_2 = (n, v, \nsim)$. We then encode the semantics of $\oplus$ directly in a policy table only containing 0s and 1s.

Consider the example in Table 9. There are four values in the decision set $D = \{\bot, 0, 1, \top\}$, and there are four unique combinations of 0 and 1, represented by the four rows in Table 9b. Each of these values arises because of matches or the absence of matches, thus allowing us to encode the semantics of $\oplus$ directly in a policy table only containing 0s and 1s.

$(n, v, \sim, !)$	F
$\bot$	0
0	0
1	1
$\top$	0

(a) Policy containing $\bot$ and $\top$

$(n, v, \sim)$	$(n, v, \nsim)$	F
0	0	0
0	1	0
1	0	1
1	1	0

(b) Policy using only 0 and 1

Figure 9: Converting a simple policy into a Boolean function

There are a number of advantages in expressing an attribute expression $\alpha = (n, v, \sim, \oplus)$ as a combination of two attribute expressions $(n, v, \sim)$ and $(n, v, \nsim)$ and encoding the semantics of $\oplus$ directly. In particular, the resulting policy table contains only binary values. Hence, we may employ existing techniques for Boolean function minimization [2].

5 COMPARISON WITH XACML AND PTACL

Having defined the AEPL policy authorization language, we now provide a brief summary of XACML and compare it with AEPL. We discuss the limitations of targets in XACML and PTaCL, before showing how XACML rules and policies may be represented in AEPL. We conclude by showing how an XACML policy set may be represented using a single AEPL policy.

5.1 XACML targets

An XACML *target*, like an attribute expression, is expressed in terms of attribute name-value pairs. It is used to determine whether a rule, a policy or a policy set is applicable to a request.

A target is defined in terms of `AllOf` and `AnyOf` elements. The `AllOf` element is used to group such pairs. Such an element is "matched" by a request if the request matches each of the name-value pairs. The `AnyOf` element is used to group `AllOf` elements. Such an element is matched if any one of the `AllOf` elements is matched. Evaluation of an XACML target returns one of two values ("matched" or "not-matched"), unlike the evaluation of an attribute expression.

Moreover, evaluation of an XACML target disregards whether a request contains a name-value pair that doesn't match a target if it also contains a name-value pair that does match. In other words, XACML provides less control over target evaluation than our approach for the evaluation of attribute expressions.

5.2 PTaCL targets

A PTaCL target is defined to be a tuple (n, v, f), where n is an attribute name, v is an attribute value and f is a binary predicate. The key difference between PTaCL targets and attribute expressions, comes in the choice of the binary operator $\oplus$ present in attribute expressions. Attribute expressions allow this operator to be selected from the set $\{\wedge, \vee, !\}$, dependent on the way in which conflicting attribute values should be handled. Targets in PTaCL implicitly assume that the operator $\vee$ is always used. This makes it impossible to distinguish scenarios where there are two name value pairs (n', v') and (n'', v'') such that $n = n', v = v'$ and $n = n'', v \neq v''$. Furthermore, PTaCL targets, much like XACML targets, are unable to return $\top$. Hence, our attribute expressions are more expressive, and provide greater control over their evaluation compared to the evaluation of targets in PTaCL.

5.3 XACML rules

An XACML rule is specified by a target t and a decision d (known as the "effect" of the rule in XACML), which may be either "allow" or "deny". The evaluation of a rule for a given request returns d if the request matches the target and "not-applicable" otherwise. Thus, an XACML rule may be encoded as a particularly simple policy table. Specifically:

- each `AllOf` element is encoded as a row in the table, in which the last entry is always d; and
- the `AnyOf` element is encoded by the different rows in the table.

Clearly, our policy tables can encode more general structures than XACML rules. Informally, a policy table would have to be encoded using two XACML rules, one for the rows for which the decision is 1 and one for the rows for which the decision is 0. Even so, such an encoding could not, for example, return $\top$. In other words, AEPL provides a richer framework than the target-decision paradigm for specifying the "leaf" policies in tree-structured languages such as XACML or PTaCL.

5.4 XACML policies

We now illustrate how attribute expressions can be used to encode an entire XACML policy set directly. Consider the tree-structured policy P illustrated in Figure 10, where dov and pov represent the XACML deny-overrides and permit-overrides combining algorithms respectively. This policy tree represents a XACML policy set (t_1, dov) which contains one policy (t_3, po) one rule $(t_2, 0)$; and the policy (t_3, pov) in turn contains two rules $(t_4, 1)$ and $(t_5, 0)$[2]. We assume for simplicity that each target is a single name-attribute pair. In Section 6.1 we explain how we can extend our method of specifying a policy as a pair (A, F) to more complex scenarios.

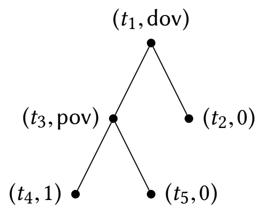

Figure 10: A simple tree-structured policy

Then we can represent this policy as the following policy table, which is created by considering every possible outcome of matching requests to targets. We write x_i to denote $\text{eval}(q, t_i)$, and – to signify the value of $\text{eval}(q, t_i)$ is irrelevant.

x_1	x_2	x_3	x_4	x_5	$P(q)$
0	–	–	–	–	$\perp$
1	1	–	–	–	0
1	0	0	–	–	$\perp$
1	0	1	1	–	1
1	0	1	0	1	0
1	0	1	0	0	$\perp$

Hence, P may be represented by the pair (A, F), where $A = \{t_1, t_2, t_3, t_4, t_5\}$ and $F : \{0,1\}^5 \to \{\perp, 0, 1\}$ is defined in the table above. Then

$$F(x_1, \dots, x_5) \equiv S^0_{(1,1)}(x_1, x_2) \vee S^1_{(1,0,1,1)}(x_1, x_2, x_3, x_4) \vee S^0_{(1,0,1,0,1)}(x_1, x_2, x_3, x_4, x_5).$$

Note that we need not include the selection operators $S^{\perp}_0(x_1)$, $S^{\perp}_{(1,0,0)}(x_1, x_2, x_3)$ and $S^{\perp}_{(1,0,1,0,0)}(x_1, x_2, x_3, x_4, x_5)$ representing the first, third and sixth rows, since the policy evaluates to $\perp$ in these rows.

The representation of the policy tree in Figure 10 can be reduced to a simple policy table, which in turn is reduced into a formula comprising just three selection operators. By expressing this policy tree as a policy table, it is much easier for a policy author to understand how this policy will behave under each different result of target evaluation. Furthermore, we have a simple formula that can

[2]This is a simplification in terms of the structure of XACML policy sets, policies and rules but approximate enough for the sake of exposition. For explicit definitions the reader is referred to the XACML standard [11].

be automatically converted into a machine-enforceable policy and evaluated by a PDP.

6 APPLICATIONS

We now demonstrate how complex policies can be built, enabling distributed policy specification and evaluation (much as in XACML and PTaCL). Furthermore, we explore how policy tables can be used to enhance existing access control paradigms, such as role-based access control and access control lists. Informally, in the first case, we show that a set of attribute expressions in an AEPL policy table (each of which evaluates to an element in $\{0,1,\bot,\top\}$) may be replaced with a set of policies. And in the second case, we show that the policy decisions in an AEPL table can be replaced with a set of role identifiers or similar.

6.1 Complex policies as tables

By specifying policies as a pair (A, F), we implicitly restrict the depth of policies (or policy trees) to one. However, this may not be the way in which some policies are structured in the real world (indeed, most policy trees in XACML have depth greater than one). We now develop a method for constructing more complex policies from simple AEPL policies, using the structure of simple policies as a template. A complex policy $\mathcal{P}$ is a pair $(\{P_1,\ldots,P_\ell\}, F)$, where $P_i = (A_i, F_i)$ is a simple AEPL policy. We define

$$\mathcal{P}(q) = F(P_1(q),\ldots,P_\ell(q)).$$

Now, instead of the columns of the policy table representing the function F being indexed by attribute expressions, they are indexed by policies. Each row represents a possible combination of the values that may arise from the evaluation of the respective policies $P_1,\ldots,P_n$. Of course, each of these policies is itself defined by a set of attribute expressions and function (A_i, F_i), which will need to be specified and evaluated first. The policy $\mathcal{P}$, much like previous policies, can be automatically converted into a machine-enforceable form via the use of selection operators. Hence, we have developed a way in which to combine arbitrary policies into machine-enforceable form. This approach can be easily scaled, providing the means to construct policies of any desired depth (much in the same manner as XACML policies).

One of the main advantages in using this method for building up complex policies, is the distributed nature in which it can be applied. For instance, in a large organization, each department could construct their own complex policy, which can be converted into a tree. Each department's policy can then become a node in a bigger tree, and be combined with other policies through the use of another policy table. A simple example of this is shown in Figure 11, demonstrating how four individual policies P_1, P_2, P_3 and P_4 produced by each department can be combined in a policy table to create the overall organizations policy P_{org}.

This policy table can be specified by someone who understands the complete policy structure of the organization and can place adequate restrictions on the interactions of policies between departments. Constructing policies in this manner allows each department to design their own specific policy, without the need to worry about how their policy interacts with other department's policies. The

P_1	P_2	P_3	P_4	P_{org}
d_1	d_2	d_3	d_4	d_{org}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 11: Combining policies in another table

combination of policies is then moderated by a person who understands the organization wide policy strategy. We believe this both simplifies specification of complex corporate policies, and reduces the likely number of misconfigurations and errors.

In addition to the distributed nature in which policies can be specified using the approach, policy evaluation may also be distributed. In the real world it is common practice for multiple PDPs to be deployed, and this architecture may be leveraged by our method of specifying complex policies. For instance, imagine the scenario where a central PDP is in charge of evaluating the organization's policy P_{org}. This central PDP may then delegate the evaluation of policies $P_1,\ldots,P_4$ to other PDPs, and combine the resulting decisions that are reported back by each PDP. There are many reasons why distributing the evaluation of policies in this way could be advantageous: (i) the load on the central PDP is reduced, (ii) free or available PDPs are fully utilised, (iii) the evaluation time for policies is reduced, and (iv) in some instances requests may even be evaluated locally.

6.2 ABAC policies for RBAC

In role-based access control (RBAC) [12], we tend to assume that users are authorized for roles on the basis of identity. With the emergence of attribute-based access control, we have an alternative option: authorizing users on the basis of their attributes. Al-Kahtani and Sandhu [1] created a model for attribute-based user-role assignment, in which an enterprise defines a set of rules that are triggered to automatically assign roles to users. The motivation for a mechanism to do this, is to reduce the number of manual user-to-role assignments that are required, which can become troublesome in large environments such as utility companies and popular online websites [1].

We now demonstrate how we can automatically assign roles to users using policies in AEPL. Previously, authorization policies were represented by a function $P : Q \to D$, where Q is the set of requests and D is the set of decisions. Now, we represent a role assignment authorization policy by a function $P : Q \to R$, where Q is the set of requests and R is the set of roles. The function P is used to determine how users are assigned to roles based on their attributes. Consider the simple example for an attribute-role table which assigns roles based on the attribute "age" for the purpose of filtering age-restricted content, shown in Table 1 [1].

Let P be a policy which comprises of four attribute expressions $\alpha_1, \alpha_2, \alpha_3$ and α_4 where $\alpha_1 = (\text{age}, 3, \geq), \alpha_2 = (\text{age}, 11, \geq), \alpha_3 = (\text{age}, 16, \geq)$ and $\alpha_4 = (\text{age}, 18, \geq)$[3], with policy function F, defined in Figure 12.

[3]The choice of operator $\oplus$ is irrelevant in this example, since requests will not contain two pairs (n, v') and (n, v'') such that n is age and $v' \neq v''$, hence we omit it.

$\text{eval}(q,(\text{age},3,\geq))$	$\text{eval}(q,(\text{age},11,\geq))$	$\text{eval}(q,(\text{age},16,\geq))$	$\text{eval}(q,(\text{age},18,\geq))$	Role
1	0	0	0	Child
1	1	0	0	Juvenile
1	1	1	0	Adolescent
1	1	1	1	Adult

Figure 12: Policy table for attribute-based role assignment

Table 1: Age to role assignment

Age	Role
≥ 3	Child
≥ 11	Juvenile
≥ 16	Adolescent
≥ 18	Adult

Hence, we have represented the age-to-role assignment as a policy $P = (A, F)$, which may in turn be represented as a combination of selection operators and converted into a machine-enforceable policy. It is easy to imagine how this methodology could be extended and applied in a setting where it is useful to automatically assign roles to users on the basis of attributes rather than identity. Our approach is scalable, simple for policy authors to understand and can be applied with various other techniques discussed throughout this paper such as ways to compress policies and using policies as leaf nodes in tree-structured languages, to produce a machine-enforceable policy which assigns roles to users.

6.3 Access control lists

In an access control system using access control lists based on identifiers, a user is associated with one or more identifiers: a unique user identifier (UID) and zero or more group identifiers (GIDs). Each object is associated with an access control list (ACL). Each ACL may be modelled as a list of access control entries (ACEs), where an ACE comprises an identifier and a set of authorized actions. Finally, a request contains a UID and a set of GIDs, an object identifier (OID) and a requested action. The UID and GIDs in the request will be compared with those in the ACEs of the object's ACL and a decision will be reached based on the actions that are authorized by the ACEs and those that have been requested.

We may extend this idea of identity-based ACLs to attribute-based ACLs. Each ACE contains a group identifier, as before, which represents an attribute-based policy. Then, we represent a group membership policy as a function $P : Q \to G$, were Q is the set of requests and G is the set of group identifiers. The policy P specifies the attributes that a user must have to be regarded as a member of that group. We may represent this policy using a set of attribute expressions A and a function F defined as a policy table, in an identical manner to that illustrated in Section 6.2. Hence, we can use AEPL to support attribute-based access control in a ACL-based system.

7 CONCLUDING REMARKS

The development and specification of attribute-based access control languages such as XACML [11], PTaCL [6] and PBel [5] will continue to increase to meet the demand for open, distributed, interconnected and dynamic systems. While XACML is a standardized language, constructing some policies may be difficult (and may be impossible due to the functional incompleteness of XACML [7]), and there is little support or guidance provided for policy authors. This problem provides the primary motivation for this paper: the development of a convenient and intuitive method for authoring policies, which can be expressed in a form that may be easily evaluated by a PDP.

In this paper, we make important contributions to the development of ABAC authorization languages. First, we define attribute expressions, which provide greater control over how requests are evaluated, compared to XACML [11] and PTaCL [6]. Then, we specify policies as tables, in which the columns are indexed by attribute expressions. Defining policies in this manner is both simple and intuitive, and allows policy authors to specify how a policy will behave under each different evaluation of attribute expressions. In XACML and other tree-structured languages, it can be difficult to foresee how large policies will evaluate under different requests. This can often lead to policy misconfigurations and errors. In addition, we demonstrate how policy tables may be automatically compiled into machine-enforceable policies, and explore various methods for policy compression, thus reducing the size of policy tables.

Second, we compare XACML and PTaCL with AEPL, showing that AEPL provides more control over target evaluation than XACML and PTaCL. Furthermore, we show how an XACML policy can be converted into a policy table. By representing XACML policies in a tabular form, it becomes easier for a policy author to understand how policies will behave under each different result of target evaluation. Finally, we demonstrate the various applications of AEPL. We show how complex policies can be constructed as tables, thus enabling a distributed method for building and evaluating enterprise-wide policies. We also show how policy tables can be used in RBAC [12] and ACLs for role and group assignments respectively, making these paradigms "attribute-aware".

There is a considerable amount of future work which naturally proceeds from the work in this paper. We plan to develop policy authoring software that provides a graphical user interface for policy authors, allowing them to specify attribute expressions, and construct a table for the desired policy. This table will then be automatically converted into a machine-enforceable policy. Naturally, we hope to develop a modified XACML PDP that can evaluate attribute expressions and the machine-enforceable policies produced by policy authoring software at a PDP. There is also motivation to develop a tool which converts a policy expressed as a tree-structured policy into an equivalent policy table, much like the example in Section 5.4. This will help facilitate the smooth transition from XACML tree-structured policies to policies defined as tables. We would also like to investigate how arbitrary conditions (found in XACML) may be included in AEPL

We would also like to conduct a usability study to test the hypothesis that the construction of AEPL policy tables is easier and less error-prone than writing XACML policies. Our vision of this study requires the participants to construct a policy presented in natural language, first as a tree-structured XACML policy, and then as a policy table in AEPL. We may then compare various elements such as (i) the ease with which the testers could construct each policy, (ii) whether the XACML and AEPL policy are equivalent, (iii) the "correctness" of each policy (how close they are to the described policy in natural language), and (iv) other metrics such as the time taken to construct each policy.

REFERENCES

[1] Mohammad A. Al-Kahtani and Ravi S. Sandhu. 2002. A Model for Attribute-Based User-Role Assignment. In *18th Annual Computer Security Applications Conference (ACSAC 2002), 9-13 December 2002, Las Vegas, NV, USA*. IEEE Computer Society, 353–362. DOI: https://doi.org/10.1109/CSAC.2002.1176307

[2] Eric Allender, Lisa Hellerstein, Paul McCabe, Toniann Pitassi, and Michael E. Saks. 2006. Minimizing DNF Formulas and AC0 Circuits Given a Truth Table. In *21st Annual IEEE Conference on Computational Complexity (CCC 2006), 16-20 July 2006, Prague, Czech Republic*. IEEE Computer Society, 237–251. DOI: https://doi.org/10.1109/CCC.2006.27

[3] Ofer Arieli and Arnon Avron. 1998. The Value of the Four Values. *Artif. Intell.* 102, 1 (1998), 97–141. DOI: https://doi.org/10.1016/S0004-3702(98)00032-0

[4] Nuel D Belnap Jr. 1977. A useful four-valued logic. In *Modern uses of multiple-valued logic*. Springer, 5–37.

[5] Glenn Bruns and Michael Huth. 2011. Access control via Belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Trans. Inf. Syst. Secur.* 14, 1 (2011), 9. DOI: https://doi.org/10.1145/1952982.1952991

[6] Jason Crampton and Charles Morisset. 2012. PTaCL: A Language for Attribute-Based Access Control in Open Systems. In *Principles of Security and Trust - First International Conference, POST 2012, Proceedings*, Pierpaolo Degano and Joshua D. Guttman (Eds.). Lecture Notes in Computer Science, Vol. 7215. Springer, 390–409. DOI: https://doi.org/10.1007/978-3-642-28641-4_21

[7] Jason Crampton and Conrad Williams. 2016. On Completeness in Languages for Attribute-Based Access Control. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, SACMAT 2016, Shanghai, China, June 5-8, 2016*, X. Sean Wang, Lujo Bauer, and Florian Kerschbaum (Eds.). ACM, 149–160. DOI: https://doi.org/10.1145/2914642.2914654

[8] Jason Crampton and Conrad Williams. 2017. Canonical Completeness in Lattice-Based Languages for Attribute-Based Access Control. *CoRR* abs/1702.04173 (2017). http://arxiv.org/abs/1702.04173 To appear in the Proceedings of CODASPY 2017; pre-print available at http://arxiv.org/abs/1702.04173.

[9] William H. Jobe. 1962. Functional Completeness and Canonical Forms in Many-Valued Logics. *J. Symb. Log.* 27, 4 (1962), 409–422. DOI: https://doi.org/10.2307/2964548

[10] Srdjan Marinovic, Naranker Dulay, and Morris Sloman. 2014. Rumpole: An Introspective Break-Glass Access Control Language. *ACM Trans. Inf. Syst. Secur.* 17, 1 (2014), 2:1–2:32.

[11] Erik Rissanen. 2012. 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.

[12] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *IEEE Computer* 29, 2 (1996), 38–47. DOI: https://doi.org/10.1109/2.485845

[13] Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David A. Basin. 2014. Decentralized Composite Access Control. In *POST (Lecture Notes in Computer Science)*, Vol. 8414. Springer, 245–264.

A UNARY SELECTION OPERATOR ENCODINGS

Figure 13 shows the normal forms of the unary selection operators S_a^b. Note that $S_a^{\perp}(x) = \perp$ for all $a, x \in \{0, 1, \perp, \top\}$.

$S_a^{\perp}(x)$	$x \wedge (\diamond - x) \wedge (\diamond - \diamond - x)$
$S_{\perp}^{0}(x)$	$\diamond(x) \wedge (-\diamond - \diamond - x) \wedge (-\diamond - \diamond x)$
$S_{0}^{0}(x)$	$x \wedge (-x) \wedge (-\diamond - x)$
$S_{1}^{0}(x)$	$(\diamond - \diamond - x) \wedge (-\diamond - \diamond - x) \wedge (\diamond x)$
$S_{\top}^{0}(x)$	$(\diamond - x) \wedge (-\diamond - x) \wedge (\diamond\diamond - \diamond x)$
$S_{\perp}^{1}(x)$	$(\diamond - \diamond x) \wedge (-\diamond - \diamond x) \wedge (\diamond\diamond x)$
$S_{0}^{1}(x)$	$(\diamond x) \wedge (-\diamond x) \wedge (\diamond - x)$
$S_{1}^{1}(x)$	$x \wedge (-x) \wedge (\diamond\diamond - \diamond x)$
$S_{\top}^{1}(x)$	$(\diamond - \diamond - x) \wedge (-\diamond - \diamond - x) \wedge (\diamond\diamond - x)$
$S_{\perp}^{\top}(x)$	$(-\diamond - x) \wedge (-\diamond - \diamond - x) \wedge (-x)$
$S_{0}^{\top}(x)$	$(\diamond - \diamond - x) \wedge (\diamond - \diamond x) \wedge (\diamond\diamond x)$
$S_{1}^{\top}(x)$	$(\diamond x) \wedge (\diamond - x) \wedge (\diamond - \diamond\diamond - \diamond x)$
$S_{\top}^{\top}(x)$	$x \wedge (-\diamond x) \wedge (-\diamond - \diamond x)$

Figure 13: Normal forms for the unary selection operators

B EQUIVALENCE OF SELECTION OPERATORS

Figure 14 establishes the following equivalence of $S_a^d(x_1)$ and

$$S_{(a,\perp)}^{d}(x_1, x_2) \vee S_{(a,0)}^{d}(x_1, x_2) \vee S_{(a,1)}^{d}(x_1, x_2) \vee S_{(a,\top)}^{d}(x_1, x_2).$$

x_1	x_2	$S^d_{(a,\perp)}(x_1,x_2)$	$S^d_{(a,0)}(x_1,x_2)$	$S^d_{(a,1)}(x_1,x_2)$	$S^d_{(a,\top)}(x_1,x_2)$	$S^d_a(x_1)$
a	$\perp$	d	$\perp$	$\perp$	$\perp$	d
a	0	$\perp$	d	$\perp$	$\perp$	d
a	1	$\perp$	$\perp$	d	$\perp$	d
a	$\top$	$\perp$	$\perp$	$\perp$	d	d

Figure 14: Equivalence of selection operators