Collaborative and distributed applications, such as dynamic coalitions and virtualized grid computing, often require integrating access control policies of collaborating parties. Such an integration must be able to support complex authorization specifications and the fine-grained integration requirements that the various parties may have. In this paper, we introduce an algebra for fine-grained integration of sophisticated policies. The algebra, which consists of three binary and two unary operations, is able to support the specification of a large variety of integration constraints. For ease of use, we also introduce a set of derived operators and provide guidelines for users to edit a policy with desired properties. To assess the expressive power of our algebra, we define notion of completeness and prove that our algebra is complete and minimal with respect to the notion. We then propose a framework that uses the algebra for the fine-grained integration of policies expressed in XACML. We also present a methodology for generating the actual integrated XACML policy, based on the notion of Multi-Terminal Binary Decision Diagrams. Experimental results have demonstrated both effectiveness and efficiency of our approach. In addition, we also discuss issues regarding obligations.

1. Introduction

Many distributed applications such as dynamic coalitions and virtual organizations need to integrate and share resources, and these integration and sharing will require the integration of access control policies. In order to define a common policy for resources jointly owned by multiple parties applications may be required to integrate policies from different sources into a single policy. Even in a single organization, there could be multiple policy authoring units. If two different branches of an organization have different or even conflicting access control policies, what policy should the organization as a whole adopt? If one policy allows the access to certain resources, but another policy denies such access, how can they be composed into a coherent whole? Approaches to policy integration are also crucial when dealing with large information systems. In such cases, the development of integrated policies may be the product of a bottom-up process under which policy requirements are elicited from different sectors of the organization, formalized in some access control language, and then integrated into a global access control policy.

When dealing with policy integration, it is well known that no single integration strategy works for every possible situation, and the exact strategy to adopt depends on the requirements by the applications and the involved parties. An effective policy integration mechanism should thus be able to support a flexible fine-grained policy integration strategy capable of handling complex integration specifications. Some relevant characteristics of such an integration strategy are as follows. First, it should be able to support 3-valued policies. A
3-valued policy may allow a request, deny a request, or not make a decision about the request. In this case we say the policy is not applicable to the request. Many widely used policy specification languages support 3-valued policies, such as XACML (2005) and firewall rule-sets. For example, firewall policies are 3-valued, in that a firewall rule may not match a request, may match a request and allow it, and may match a request and deny it. Furthermore, three-valued policies are necessary for combining partially specified policies, which are very likely to occur in scenarios that need policy integration. When two organizations are merging and need policy integration, it is very likely that the organizations are unaware or might not have jurisdiction over each other resources, and thus a policy in one organization may be “NotApplicable” to requests about resources in the other organization. Second, it should allow one to specify the behavior of the integrated policy at the granularity of requests and effects. In other words, one should be able to explicitly characterize a set of requests that need to be permitted or denied by the integrated policy. For example, users may require the integrated policy to satisfy the condition that for accesses to an object O1 policy P1 has the precedence, whereas for accesses to an object O2 policy P2 has precedence. Third, it should be able to handle domain constraints requiring the integrated policy to be applied to a restricted domain instead of the original domain. And fourth, it should be able to support policies expressed in rich policy languages, such as XACML with features like policy combining algorithms.

The problem of policy integration has been investigated in previous works. The concept of policy composition under constraints was first introduced by Bonatti et al. (2002). They proposed an algebra for composing access control policies and use logic programming and partial evaluation techniques for evaluating algebra expressions. Another relevant approach is by Wijesekera and Jajodia (2003) who proposed a propositional framework for composing access control policies. Those approaches have however a number of shortcomings. They support only limited forms of compositions. For example, they are unable to support compositions that take into account policy effects or policy jumps (i.e., if P1 permits, let P2 makes decision, otherwise P3 makes decision). They only model policies with two decision values, either “Permit” or “Deny”. It is not clear the scope or expressive power of their languages since they do not have any notion of completeness. They do not provide an actual methodology or an implementation for generating the integrated policies. Neither work relates their formalisms to any language used in practice.

In this paper, we propose a framework for the integration of access control policies that addresses the above shortcomings. The overall organization of our integration framework is outlined in Fig. 1. The core of our framework is the Fine-grained Integration Algebra (FIA). Given a set of input policies P1, P2, …, Pn, one is able to specify the integration requirements for these input policies through a FIA expression, denoted as \( f(P_1, P_2, \ldots, P_n) \) in Fig. 1. The FIA expression is then processed by the other components of the framework in order to generate the integrated policy. We demonstrate the effectiveness of our framework through an implementation that supports the integration of XACML policies. We choose XACML because of its widespread adoption and its many features, such as attribute-based access control and 3-valued policy evaluation. We use Multi-Terminal Binary Decision Diagrams (MTBDD) (Fujita et al., 1997) for representing policies and generating the integrated policies in XACML syntax. With the aid of the implementation of algebra operators, users can now easily specify their integration requirements by an expression and do not need to write a new complex policy combining algorithm by themselves. The novel contributions of this paper can be summarized as follows:

- We propose a fine-grained integration algebra for language independent 3-valued policies. We introduce a notion of completeness and prove that our algebra is minimal and complete with respect to this notion.
- We propose a framework that uses the algebra for the fine-grained integration of policies expressed in XACML. The method automatically generates XACML policies as the policy integration result. To the best of our knowledge, none of the existing approaches has generated real policies as policy integration output.
- We have carried out experimental studies which demonstrate the efficiency and practical value of our policy integration approach.

A preliminary version of this paper appears in Rao et al. (2009), where we presented the basic idea of the policy integration. In this paper, we make the following additional contributions. First, we prove the minimality of our algebra. Second, we analyze properties of the algebra and provide guidelines for users to easily edit a policy with desired properties. Third, we also discuss how to take into account obligations during policy integration. Fourth, we enrich our experimental study with a new set of experiments. In addition, we also provide more detailed description of the framework and algorithms as well as an extension to the related works.

The rest of the paper is organized as follows. Section 2 reviews related works on policy integration. Section 3 introduces background information on XACML and some preliminary definitions. Section 4 presents our fine-grained integration algebra. Section 5 discusses the expressiveness of the algebra. Section 6 presents detailed algorithms for generating well-formed integrated XACML policies. Section 7 reports our experimental study. Section 8 discusses some issues regarding obligations. Section 9 concludes the paper.
2. Related work

Many efforts have been devoted to policy composition (Backes et al., 2004a; Bonatti et al., 2002; McDaniel and Prakash, 2006; Schneider, 2000; Wijesekera and Jajodia, 2003; Halpern and Weissman, 2003, 2008; Jagadeesan et al., 2005; Bruns et al., 2007; Wijesekera and Jajodia, 2003). However, for very few of these approaches, the expressive power has been analyzed. Moreover, none of them generates real policies as result of policy composition. In what follows, we present a detailed comparison between our work and the most related approaches.

One early work on policy composition is the policy algebra proposed by Bonatti et al. (2002), which aims at combining authorization specifications originating from heterogenous independent parties. They model an access control policy as a set of ground (variable-free) authorization terms, where an authorization term is a triple of the form (subject, object, action). However, their algebra only supports 2-valued policies and they did not clearly point out what authorization specifications can be expressed and what cannot by using their algebra. Regarding the algebra implementation, they suggest to use logic programming, but did not show any experimental result. Compared to their algebra implementation, they suggest to use logic programming, but did not show any experimental result. Compared to their work, we have proved that our algebra can express any possible policy integration requirement and our implementation is based on representations used in model checking techniques which have been proven to be very efficient.

Halpern and Weissman (2003, 2008) have used first-order logic to specify policies and support the composition of policies in this context. Only those policies which do not have any conflicts when logically combined can be composed using their framework. In contrast, our work defines operators that can be used to resolve any potential conflicts that may arise when performing the composition. Wijesekera and Jajodia (2003) have proposed a propositional algebra for access control. They model policies as nondeterministic transformers of permission set assignments to subjects and interpret operations on policies as set-theoretic operations on the transformers. They make use of signed action terms to represent negative authorizations. Later, Jagadeesan et al. (2005) proposed a 3-valued policy algebra using the timed concurrent constraint programming paradigm and define boolean operators whose expressive power is equivalent to the algebra in Bonatti et al. (2002) in addition to added temporal features. However, this is only a theoretical work which did not have any implementation for generating the integrated policy.

Another related work is by Backes et al. (2004a) who proposed an algebra for combining enterprise privacy policies. They define conjunction, disjunction and scoping operations on 3-valued EPAL (Ashley et al., 2003) policies. However, they did not prove the completeness. In other words, they cannot support some integration requirements that can be expressed by our algebra. Mazzioloni et al. (2006) have proposed an extension to the XACML language, called policy integration preferences, using which a party can specify the approach that must be adopted when its policies have to be integrated with policies by other parties. They do not discuss mechanisms to perform such integrations. Also, the integration preferences discussed in such work are very limited and do not support fine-grained integration requirements as those presented in Section 5.2.

Most recently, Bruns et al. (2007) proposed an algebra for four-valued policies based on Belnap bilattice. In particular, they map four possible policy decisions, i.e. grant, deny, conflict and unspecified, to Belnap bilattice and claim that their algebra is complete and minimal. Moreover, they did not propose any implementation of their algebra.

Our work is also related to the area of many -valued logics. Most work in such area focuses on establishing criteria for Sheffer functions in m-valued logic. A Sheffer function is a single logical function that is complete. Martin (1954) isolates all binary sheffer functions in 3-valued logic and proves properties of such functions. Wheeler (Wheeler, 1966) proves a generalization of (Martin, 1954) and establishes the necessary and sufficient conditions for n-ary Sheffer functions in the context of 3-valued propositional calculus. Rousseau (1966) provides further generalization and proves the necessary and sufficient conditions for any finite algebra with a single operation to be complete. Arieli and Avron (1998) propose a propositional language with four-valued semantics and study the expressive power of their language. They also compare 3-valued and 4-valued formalisms. In contrast to these approaches, we do not find or establish criteria for all possible complete operators or functions for a 3-valued algebra. Instead, we focus on the definition of a set of operators that have intuitive semantic meaning in the context of combining 3-valued policies and study whether this set of operators is complete. We also study properties such as expressive power and minimality for this set of operators.

In addition, some aspects of our work are relevant to the area of access control policy analysis (Agrawal et al., 2005; Backes et al., 2004b; Koch et al., 2001; Lupu and Sloman, 1999; Moffett and Sloman, 1993; Schaad and Moffett, 2002; Zhang et al., 2005). McDaniel and Prakash (2006) have studied the problem of automated reconciliation of provisioning and authorization policies and have shown this problem to be intractable for more than two policies. A well-known approach for XACML policy analysis is proposed by Fisler et al. (2005), who have first introduced the use of MTBDDs to represent XACML policies. They use such MTBDDs to verify properties and perform change impact analysis on XACML policies. A major drawback of this approach is that it is limited to policies with conditions consisting only of atomic Boolean expressions of the form \( a = v \). By contrast, our approach supports a large variety of Boolean expressions, including range conditions.

3. Preliminaries

In this section, we first present a short overview of XACML to recall the main concepts of this access control policy language, and then introduce some preliminary definitions stating the notion of access control policy semantics that we adopt in our work.

3.1. An overview of XACML

XACML (2005) is the OASIS standard language for the specification of access control policies. It is an XML language capable
of expressing a large variety of policies, taking into account properties of subjects and protected objects as well as context information; several profiles have also been defined for it, such as a role profile and a privacy profile. Commercial implementations of XACML are also available (Parthenon XACML; Sun’s XACML).

XACML policies include the following components: a Target, a Rule set, a Rule combining algorithm and a set of Obligations. The Target identifies the set of requests that the policy applies to. Each Rule consists of Target, Condition and Effect elements. The rule Target has the same structure as the policy Target. It specifies the set of requests that the rule applies to. The Condition element may further refine the applicability established by the target. The Effect element specifies whether the requested actions should be allowed (“Permit”) or denied (“Deny”). The restrictions specified by the target and condition elements support the notion of attribute-based access control under which access control policies are expressed as conditions against the properties of subjects and protected objects. In XACML such conditions are represented as Boolean functions taking the request attribute values as input, and returning true or false depending on whether the input attributes satisfy certain conditions. If a request satisfies both the rule target and rule condition predicates, the rule is applicable to the request and will yield a decision as specified by the Effect element. Otherwise, the rule is not applicable to the request and will yield a “NotApplicable” decision. If an error occurs to the evaluation of a rule, an “Indeterminate” decision is returned. The 3-valued algebra discussed in this paper is applicable to error free XACML policies where in the authorization decision is one of “Permit”, “Deny” or “NotApplicable”.

The Rule combining algorithm is used to resolve conflicts among applicable rules with different effects. Obligations represent a set of operations that must be executed in conjunction with an authorization decision. An obligation is associated with either “Permit” or “Deny” decision.

We now introduce an example of XACML policies that will be used throughout the paper.

**Example 1.** Consider a company with two departments D1 and D2. Each department has its own access control policies for the data under its control. Assume that P1 and P2 are the access control policies of D1 and D2 respectively. P1 contains two rules, P1.Rul1,1 and P1.Rul1,2,1 states that the manager is allowed to read and update any data in the time interval [8am, 6pm]. P2.Rul2,1 states that any other staff is not allowed to read. P2 also contains two rules, P2.Rul2,1 and P2.Rul2,2, P2.Rul2,1 states that the manager and staff can read any data in the time interval [8am, 6pm], and P2.Rul2,2 states that the staff cannot perform any update action. For simplicity, we adopt the following succinct representation in most discussion, where “role”, “act” and “time” are attributes representing information on role, action and time, respectively.

- \( P1.Rul1,1: \text{role} = \text{manager}, \text{act} = \text{read or update}, \text{time} = [8\text{am}, 6\text{pm}], \text{effect} = \text{Permit} \).
- \( P1.Rul1,2: \text{role} = \text{staff}, \text{act} = \text{read}, \text{effect} = \text{Deny} \).
- \( P2.Rul2,1: \text{role} = \text{manager or staff}, \text{act} = \text{read}, \text{time} = [8\text{am}, 8\text{pm}], \text{effect} = \text{Permit} \).
- \( P2.Rul2,2: \text{role} = \text{staff}, \text{act} = \text{update}, \text{effect} = \text{Deny} \).

3.2. Definitions

Before we introduce our algebra we need to find a suitable definition for a policy. We propose a simple yet powerful definition for a policy according to which a policy is defined by the set of requests that are permitted by the policy and the set of requests that are denied by the policy. This simple notion will provide us with a precise characterization of the meaning of policy integration in terms of the sets of permitted and denied requests. In the rest of this paper, we use \( Y, N \) and \( NA \) to denote the “Permit”, “Deny” and “NotApplicable” decisions respectively.

We believe that such a notion is the most adequate for our integration framework in that our goal is exactly to provide a precise characterization of an integrated policy in terms of the set of permitted requests and the set of denied requests.

In our work, we assume the existence of a vocabulary \( \Sigma \) of attribute names and domains. Each attribute, characterizing a subject or an object or the environment, has a name \( a \) and a domain, denoted by \( \text{dom}(a) \), in \( \Sigma \). The following two definitions introduce the notion of access request (request, for short) and policy semantics.

**Definition 1.** Let \( a_1, a_2, \ldots, a_k \) be a set of attribute names and let \( v_i \in \text{dom}(a_i) (1 \leq i \leq k) \) in the vocabulary \( \Sigma \). \( r = \{(a_1, v_1), (a_2, v_2), \ldots, (a_k, v_k)\} \) is a request over \( \Sigma \). The set of all requests over \( \Sigma \) is denoted as \( R_\Sigma \).

**Example 2.** Consider policy \( P_1 \) from **Example 1**. An example of a request to which this policy applies is that of a manager wishing to read any resource at 10am. According to **Definition 1**, such request can be expressed as \( r = \{(\text{role, manager}), (\text{act, read}), (\text{time, 10am})\} \).

**Definition 2.** A 3-valued access control policy \( P \) is a function mapping each request to a value in \( \{Y, N, NA\} \). \( R_\Sigma, R_N \) and \( R_\Sigma \) denote the set of requests permitted, denied and not applicable by the policy \( P \) respectively, and \( R_\Sigma = R_\Sigma \cup R_N \), \( R_\Sigma \cap R_N = \emptyset \), \( R_N \cap R_\Sigma = \emptyset \), \( R_N \cap R_\Sigma = \emptyset \). We define a policy \( P \) as a 3-tuple \( (R_\Sigma, R_N, R_\Sigma) \).

It is worth noting that, though we define policies as set of requests, we do not need to enumerate all such requests when executing policy operations. The concept of request sets is only for a better understanding of policy operations. Our approach to formulating the definition of a policy is independent of the language in which access control policies are expressed. Therefore, our approach can be applied to languages other than XACML.

4. A Fine-grained integration algebra

The Fine-grained Integration Algebra (FIA) is given by \( (\Sigma, P_\Sigma, \cup, \cup, \cdot, P_{\Sigma}, \text{and}, \text{or}, \cdot) \), where \( \Sigma \) is the vocabulary of attribute names and their domains, \( P_\Sigma \) and \( P_{\Sigma} \) are two policy constants, \( \text{and} \) and \( \text{or} \) are two binary operators, and \( \cdot \) is two unary operators. It is a minimal and complete set of operators (a theorem is given in Section 5.3).
4.1. Policy constants and basic operators in FIA

We now describe the policy constants and operators in FIA. In what follows, \( P_1 = (R_{1N}, R_{1i}, R_{1r}) \) and \( P_2 = (R_{2N}, R_{2i}, R_{2r}) \) denote two policies to be combined, and \( P_1 = (R_{1N}, R_{1i}, R_{1r}) \) denotes the policy obtained from the combination. Operators on policies are described as set operations.

- Permit policy \((P_Y)\). \( P_Y \) is a policy constant that permits everything. Thus \( P_Y = (R_2, \phi, \phi) \)
- Deny policy \((P_N)\). \( P_N \) is a policy constant that denies everything. Thus \( P_N = (\phi, R_2, \phi) \)
- Addition (+). Addition of policies \( P_1 \) and \( P_2 \) results in a combined policy \( P_I \) in which requests that are permitted by either \( P_1 \) or \( P_2 \) are permitted, requests that are denied by one policy and are not permitted by the other are denied more precisely:

\[
P_I = P_1 + P_2 = \begin{cases} R_{1i}^r = R_{1i}^r \cup R_{2i}^r \\ R_{1N}^r = (R_{1N}^r / R_{2N}^r) \cup (R_{2N}^r / R_{1N}^r) \end{cases}
\]

**Example 3.** Consider Example 1. The result of adding two policies will give more authorization to both manager and staff. The integrated policy \( P_I \) may contain the following four rules, from which we can see that now the staff from department D1 are also allowed to read customer information.

- \( P_1, \text{Rule1: role} = \text{manager or staff, act} = \text{read, time} = [8am, 8pm], \text{effect} = \text{Permit} \)
- \( P_2, \text{Rule2: role} = \text{manager, act} = \text{update, time} = [8am, 6pm], \text{effect} = \text{Permit} \)
- \( P_1, \text{Rule3: role} = \text{staff, act} = \text{read, time} \neq [8am, 8pm], \text{effect} = \text{Deny} \)
- \( P_1, \text{Rule4: role} = \text{staff, act} = \text{update, effect} = \text{Deny} \)

- Intersection (&). Given two policies \( P_1 \) and \( P_2 \), the intersection operator returns a policy \( P_I \) which is applicable to all requests having the same decisions from \( P_1 \) and \( P_2 \). More precisely,

\[
P_I = P_1 \& P_2 = \begin{cases} R_{1i}^r = R_{1i}^r \cap R_{2i}^r \\ R_{1N}^r = (R_{1N}^r / R_{2N}^r) \cap (R_{2N}^r / R_{1N}^r) \end{cases}
\]

The integrated policy makes a decision only when the two policies agree. The intersection operator is useful in situations in which consensus is required for making an allow or deny decision.

**Example 4.** Consider Example 1 again, the result of intersecting two policies (\( P_1 = P_1 \& P_2 \)) is the following:

- \( P_1, \text{Rule1: role} = \text{manager, act} = \text{read, time} = [8am, 6pm], \text{effect} = \text{Permit} \)

A binary operator can be viewed as a function that maps a pair of values \((Y,N, NA)\) to one value. Table 1 gives this view of addition and intersection operators. A binary operator is represented using a matrix that illustrates the effect of integration for a given request \( r \). The first column of each matrix denotes the effect of \( P_1 \) with respect to \( r \) and the first row denotes the effect of \( P_2 \) with respect to \( r \).

**Negation (¬).** Given a policy \( P \), \( ¬P \) returns a policy \( P \), which permits (denies) all requests denied (permitted) by \( P \). The negation operator does not affect those requests that are not applicable to the policy. More precisely:

**Table 1 – Policy combination matrix of operator +, &.**

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>Y</th>
<th>N</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>NA</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>NA</td>
</tr>
<tr>
<td>N</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

\[
P_I = ¬P = \begin{cases} R_{1i}^r = R_{1i}^r \\ R_{1N}^r = R_{1N}^r \end{cases}
\]

**Domain projection (\( \Pi_{dc} \)).** The domain projection operator takes as parameter the domain constraint \( dc \) and restricts a policy to the set of requests identified by \( dc \).

**Definition 3.** A domain constraint \( dc \) takes the form \( ((a_1, range_1) | (a_2, range_2), ..., (a_k, range_k)) \), where \( a_1, a_2, ..., a_k \) are attribute names, and range \( (1 \leq i \leq k) \) is a set of values in \( \text{dom} (a) \).

Given a request \( r = ((a_1, v_1), ..., (a_m, v_m)) \), we say that \( r \) satisfies \( dc \) if the following condition holds: for each \( (a_i, v_j) \epsilon r (1 \leq j \leq m) \) there exists \( (a_i, range) \epsilon dc \), such that \( a_i = a \) and \( v_j \epsilon range_i \).

The semantics of \( \Pi_{dc}(P) \) is given by

\[
P_I = \Pi_{dc}(P) = \begin{cases} R_{1i}^r = \{ r \in R_{1i}^r \text{ and } r \text{ satisfies } dc \} \\ R_{1N}^r = \{ r \in R_{1N}^r \text{ and } r \text{ satisfies } dc \} \end{cases}
\]

4.2. FIA expressions

The integration of policies may involve multiple operators (as shown in Table 2), and hence we introduce the concept of FIA expressions.

**Definition 4.** A FIA expression is recursively defined as follows:

- If \( P \) is policy, then \( P \) is a FIA expression.
- If \( f_1 \) and \( f_2 \) are FIA expressions so are \( f_1 + f_2 \), \( f_1 \& f_2 \), and \( ¬(f_1) \).
- If \( f \) is a FIA expression and \( dc \) is a domain constraint then \( \Pi_{dc}(f) \) is a FIA expression.

In what follows we will use the terms “policy” and “expression” synonymously. In FIA expressions, the binary operators are viewed as left associative and unary operators are right associative. The precedence are \( ¬ \) and \( \Pi_{dc} \) together have the highest precedence, followed by \( & \) and then by \( + \). For example, \( P_1 + \Pi_{dc} P_2 + ¬P_3 \& P_4 \) is interpreted as \( ((P_1 + (\Pi_{dc} P_2)) + (¬P_3)) \& P_4 \).

FIA has algebraic properties including commutativity, associativity, absorption, distributivity, complement, idempotence, boundedness and involution as shown in Theorem 1. These properties are important for simplifying FIA expressions so that fewer operators need to be carried out for the policy integration.

\[1\] In case of an ordered domain, these sets can be represented by ranges.
Table 2 – Basic policy operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>$P_1 + P_2$</td>
</tr>
<tr>
<td>&amp;</td>
<td>Intersection</td>
<td>$P_1 &amp; P_2$</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
<td>$P_1 - P_2$</td>
</tr>
<tr>
<td>~</td>
<td>Negation</td>
<td>$-P$</td>
</tr>
<tr>
<td>Π</td>
<td>Projection</td>
<td>$Π_{\text{ext}}(P)$</td>
</tr>
</tbody>
</table>

Theorem 1. Let $P_1$, $P_2$ and $P_3$ denote policies. FIA has the following algebraic properties.

- **Commutativity:**
  
  
  
  $P_1 + P_2 = P_2 + P_1$;
  
  $P_1 \& P_2 = P_2 \& P_1$;

- **Associativity:**
  
  
  
  $(P_1 + P_2) + P_3 = P_1 + (P_2 + P_3)$;
  
  $(P_1 \& P_2) \& P_3 = P_1 \& (P_2 \& P_3)$;

- **Absorption:**
  
  
  
  $P_1 + (P_1 \& P_2) = P_1$;
  
  $P_1 \& (P_1 + P_2) = P_1$;

- **Distributivity:**
  
  
  
  $P_1 + (P_2 \& P_3) = (P_1 + P_2) \& (P_1 + P_3)$;
  
  $P_1 \& (P_2 + P_3) = (P_1 \& P_2) + (P_1 \& P_3)$;
  
  $Π_{\text{ext}}(P_1 + P_2) = Π_{\text{ext}}(P_1) + Π_{\text{ext}}(P_2)$;
  
  $Π_{\text{ext}}(P_1 \& P_2) = Π_{\text{ext}}(P_1) \& Π_{\text{ext}}(P_2)$;

- **Complement:**
  
  
  
  $P_1 + ¬P_1 = 1$;
  
  $P_1 \& ¬P_1 = 0$;

- **Idempotence:**
  
  
  
  $P_1 + P_1 = P_1$;
  
  $P_1 \& P_1 = P_1$;

- **Boundedness:**
  
  
  
  $P_1 + P = P_1$;
  
  $P_1 \& P = P_1$;

- **Involution:**
  
  
  
  $¬(¬P_1) = P_1$.

Theorem 1 can be easily proved by using the set representations of FIA operators and the well-known properties of the Boolean algebra, and hence we do not include the details here.

4.3. Derived operators

In this section, we introduce some commonly used operators. They are defined using the core operators.

- **Not-applicable policy** ($P_{\text{NA}}$). $P_{\text{NA}}$ is a policy constant that is not applicable for every request. Since the & operator applies only to requests that have common effects and $P_Y$ and $P_N$ have no requests with common effects, $P_Y \& P_N$ yields a policy that is not applicable for every request. Thus $P_{\text{NA}}$ can be defined as $P_Y \& P_N$.

- **Effect projection** ($Π_Y$ and $Π_N$). $Π_Y(P)$ restricts the policy $P$ to the requests allowed by it. It is defined as: $Π_Y(P) = P \& P_Y$. Similarly, $Π_N(P)$ restricts the policy $P$ to the requests denied by it; it is defined as $Π_N(P) = P \& P_N$. We are overloading $Π$ to denote both effect projection and domain projection; the meaning should be clear from the subscript.

- **Subtraction** (−). Given two policies $P_1$ and $P_2$, the subtraction operator returns a policy $P_1$ which is obtained by starting from $P_1$ and limiting the requests to which the integrated policy applies only to those to which $P_2$ does not apply. The subtraction operator is defined as:

  $P_1 - P_2 = (P_Y \& ¬(¬P_1 + P_2 + ¬P_2)) + (P_N \& (P_1 + P_2 + ¬P_2))$.

To see why this is correct, observe that $¬P_1 + P_2 + ¬P_2$ will deny a request if and only if $P_1$ allows it and $P_2$ gives NA for it. Thus $P_Y \& ¬(¬P_1 + P_2 + ¬P_2)$ allows a request if and only if $P_1$ allows it and $P_2$ gives NA it, and is not applicable for all other requests. Similarly, $P_N \& (P_1 + P_2 + ¬P_2)$ denies a request if and only if $P_1$ denies it and $P_2$ gives NA for it.

Example 5. Policies $P_1$ and $P_2$ in Example 1 grant different authorizations to the manager and staff, which means that some requests may receive different decisions from two policies. The integrated policy $P_1 = P_1 - P_2$ removes possible conflicts by providing a new policy, presented below, that only contains requests in the domain of $P_1$.

$P_{1, \text{Rul}_1}$: role = manager, act = update, time = [8am, 8pm], effect = Permit.

Precedence ($\triangleright$). Given two policies $P_1$ and $P_2$, the precedence operator returns a policy $P_1$ which allows the same decision as $P_1$ for any request applicable to $P_1$, and yields the same decisions as $P_2$ for the remaining requests. The precedence operator can be expressed as $P_1 + (P_2 - P_1)$. By limiting $P_2$ to requests that $P_1$ does not decide, this operator can be used as a building block for resolving possible conflicts between two policies.

For better understanding of the derived binary operators, Table 3 shows their policy combination matrices.

5. Expressiveness of FIA

In this section, we first show that our operators can express the standard policy-combining algorithms defined for XACML policies as well as other more complex policy integration scenario. We then show that the operators in FIA are minimal and complete in that any possible policy integration requirements can be expressed using a FIA expression. Finally, we discuss some interesting reasonability properties of FIA.

5.1. Expressing XACML policy combining algorithms in FIA

Although users can choose standard XACML policy-combining algorithms for simple policy combination while use our proposed algebra for other complicated combination, we show that our algebra is expressive enough to cover standard policy-combining algorithms.

In XACML there are six standard policy-combining algorithms as follows:

- **Permit-overrides**: The combined result is "Permit" if any policy evaluates to "Permit", regardless of the evaluation result of the other policies. If no policy evaluates to "Permit"
and at least one policy evaluates to “Deny”, the combined result is “Deny”. The combination of policies $P_1, P_2, \ldots, P_n$ under this policy-combining algorithm can be expressed as $P_1 + P_2 + \cdots + P_n$.

**Ordered-Permit-overrides:** The behavior of this policy combining algorithm is exactly the same as Permit-overrides except that the policies must be evaluated in the originally specified order. Thus, if $P_1, P_2, \ldots, P_n$ is an ordered set of policies, the combination of these policies under this policy-combining algorithm can be expressed as $P_1 + P_2 + \cdots + P_n$.

**Deny-overrides:** The combined result is “Deny” if any policy is encountered that evaluates to “Deny”. The combined result is “Permit” if no policy evaluates to “Deny” and at least one policy evaluates to “Permit”. Deny-overrides is the opposite of permit-overrides. By using the combination of the negation and addition operator, we can express deny-overrides as $\neg((\neg P_1) + (\neg P_2) + \cdots + (\neg P_n))$.

**Ordered-Deny-overrides:** The behavior of this policy combining algorithm is exactly the same as Deny-overrides except that the policies must be evaluated in the originally specified order. Thus, if $P_1, P_2, \ldots, P_n$ is an ordered set of policies to be combined, the combination of these policies under this policy-combining algorithm can be expressed as $\neg((\neg P_1) + (\neg P_2) + \cdots + (\neg P_n))$.

**First-applicable:** The combined result is the same as the result of the first-applicable policy. This combining algorithm can be expressed by using the precedence operator. Given policies $P_1, P_2, \ldots, P_n$, the expression is $P_1 \triangleright P_2 \triangleright \cdots \triangleright P_n$.

**Only-one-applicable:** The combined result corresponds to the result of the unique policy in the policy set which applies to the request. Specifically, if no policy or more than one policies are applicable to the request, the result of policy combination should be “NotApplicable”; if only one policy is considered applicable, the result should be the result of evaluating the policy.

When combining policies $P_1, \ldots, P_n$ under this policy-combining algorithm, we need to remove from each policy the requests applicable to all the other policies and then combine the results using the addition operator. The final expression is:

$$(P_1 - P_2 - P_3 - \cdots - P_n) + (P_2 - P_1 - P_3 - \cdots - P_n) + \cdots + (P_n - P_1 - P_2 - \cdots - P_{n-1})$$

### 5.2 Expressing complex policy integration requirements in FIA

Our algebra supports not only the aforementioned policy combining algorithms, but also other types of policy-combining requirements, like rule constraints. A rule constraint specifies decisions for a set of requests. It may require that the integrated policy has to permit a critical request. Such an integration requirement can be represented as a new policy. Let $P$ be a policy, and $c$ be the policy specifying an integration constraint. We can combine $c$ and $P$ by using the first-applicable combining algorithm. The corresponding expression is $c \triangleright P$.

A frequently used operator is to find the portion of a policy $P$ that differs from a policy $P_0$, which can be used to detect the difference between various versions of policies. This can be expressed as: $P \triangleright (-P_0)$.

By using the two policy constants, we can easily modify a policy $P$ as an open policy or a closed policy. An open policy of $P$ allows everything that is not explicitly denied, which can be represented as $P \triangleright P_\perp$. A closed policy of $P$ denies everything that is not explicitly permitted, which can be represented as $P \triangleright P_\perp$.

Our algebra can also express the policy jump (similar to if-then-else), a feature in the iptables firewall languages (Zhang et al., 2007). The specific requirement is that if a request is permitted by policy $P_1$, then the final decision on this request is given by policy $P_2$; otherwise, the final decision is given by policy $P_3$. This can be expressed using

$$\Pi_1(P_1 \triangleright P_2) + \Pi_2(-P_1 \triangleright P_2) + \Pi_1(-P_1 \triangleright P_3) + \Pi_3(P_1 \triangleright P_3)$$

Among the four sub-expressions, the first one gives $Y$ when both $P_1$ and $P_2$ do, and gives $\bar{N}$ in all other cases. Similarly, the second sub-expression gives $N$ when $P_1$ gives $Y$ and $P_2$ gives $N$, and gives $\bar{NA}$ otherwise. The third sub-expression gives $Y$ when $P_1$ gives $N$ and $P_3$ gives $Y$ and finally the fourth sub-expression gives $N$ when both $P_1$ and $P_3$ give $N$.

Next, we elaborate the example mentioned in the introduction where the combination requirements are given for parts of a policy.

**Example 6.** Consider the policies introduced in **Example 1**. Assume that the policies must be integrated according to the following combination requirement: for users whose role is manager, the access has to be granted according to policy $P_1$; for users whose role is a staff, the access has to be granted according to policy $P_2$.

The resultant policy will consist of two parts. One part is obtained from $P_1$ by restricting the policy to only deal with managers. Such extraction can be expressed in our algebra as $\Pi_{dc_1}(P_1)$ where $dc_1 = \{\text{role manager} \}, \{\text{act read update} \}, \{\text{time [8am, 6pm]} \}$. The other part is obtained from $P_2$ by restricting the policy to only deal with staff. Correspondingly, we can use the expression: $\Pi_{dc_2}(P_2)$ with $dc_2 = \{\text{role staff} \}, \{\text{act read update} \}, \{\text{time [8am, 8pm]} \}$. Finally, we have the following expression representing the integrated policy: $\Pi_{dc_1}(P_1) + \Pi_{dc_2}(P_2)$. The integrated policy $P_1$ is thus:

$P_1, \text{Rul}_1: \text{role} = \text{manager}, \text{act} = \text{read or update}, \text{time} = [8am, 6pm], \text{effect} = \text{Permit}$

$P_1, \text{Rul}_2: \text{role} = \text{staff}, \text{act} = \text{read}, \text{time} = [8am, 8pm], \text{effect} = \text{Permit}$

$P_1, \text{Rul}_3: \text{role} = \text{staff}, \text{act} = \text{update}, \text{effect} = \text{Deny}$

### 5.3 Completeness

While we have shown that many policy integration scenarios can be handled by the operators in the algebra, our list of examples is certainly not exhaustive. A question of both theoretical and practical importance is whether FIA can express all possible ways of integrating policies, that is, whether FIA is complete. Addressing this question requires choosing a suitable notion of completeness. There are different degrees of completeness, and we show that FIA is complete in the strongest sense. First, while Table 1 gave the policy combination matrices for the four binary operators, many other matrices are possible, and each such matrix can be viewed as a binary operator for combining two policies. As there are three possibilities for each cell in a matrix, namely, $Y$, $N$, and $\bar{NA}$, and there
are nine cells, the total number of matrices is $3^9 = 19683$. Second, when $n (\geq 2)$ policies are combined, policy combination can be expressed using a $n$-dimensional matrix. We show that each such $n$-dimensional matrix can be expressed using $(P_N, P_Y, +, \& , \neg)$. Finally, a fine-grained integration may use different policy combination matrices for different requests. We show that this can be handled by using the operator $\Pi_{nk}$ in addition to $(P_N, P_Y, +, \& , \neg)$.

**Theorem 2.** (Completeness) Given $n (\geq 1)$ policies $P_1, P_2, \ldots, P_n$, let $M^*(P_1, P_2, \ldots, P_n)$ be a $n$-dimensional policy combination matrix which denotes the combination result of the $n$ policies. There exists a FIA expression $f_I(P_1, P_2, \ldots, P_n)$ that is equivalent to $M^*(P_1, P_2, \ldots, P_n)$.

**Proof.** We prove this theorem by induction. The base case is when $n = 1$. Given a policy $P_1$, its 1-dimensional matrix (Table 3) contains entries corresponding to the Permit, Deny and NotApplicable request sets. For each entry, we aim to find an $I$ expression, i.e., $f$. Note that we do not need to consider the cell where $e = 1$. When $e_1 = Y$, $f_1 = P_Y \& P_1$; when $e_1 = N$, $f_1 = (P_Y \& P_1)$. Similarly, we can obtain $f_2$, which is $P_Y \& P_1$ for $e_2$ equal to N and $P_N \& P_1$ for $e_2$ equal to Y. $f_3 = P_Y - P_1$ when $e_3$ is Y and $P_N - P_1$ when $e_3$ is N. Finally, $f_1$ is the sum of three expressions, i.e., $f_1 = f_1 + f_2 + f_3$. Note that when all three entries are NA, the integrated policy will be $P_N$.

Assuming that when $n = k$ the theorem holds, we now consider the case when $n = k + 1$. As shown in Tables 4 and 5(a), $M^*(P_1, P_2, \ldots, P_n)$ has $3^n$ entries in total, each of which is denoted as $e_{ij}(1 \leq i \leq 3, 1 \leq j \leq 3^{k-1})$. Take entries $e_{ij}$ and $e_{i+1}$, as a $(k - 1)$-dimensional policy combination matrix, and we have three such $(k - 1)$-dimensional policy combination matrices corresponding to the policy $P_k$'s effect. Based on the assumption, we obtain the FIA expression for each cell for the $k - 1$ policies as shown in the column of $f^{k-1}(P_1, \ldots, P_{k-1})$.

Next, we extend $f^{k-1}(P_1, \ldots, P_{k-1})$ to $f^k(P_1, \ldots, P_k)$ for each cell in $M^*$ (in what follows we use $f^k$ and $f^h$ for short). According to the effect of $P_k$ and $e_{ij}$, we summarize the expressions of $f^k$ in Table 5(b). Note that we do not need to consider the cell where $e_{ij}$ is NA.

Finally, we add up $f^k$ for all the cells and obtain the expression $f(P_1, P_2, \ldots, P_k)$.

We have shown that the corollary holds for $n = 1$, and we also have shown that if the corollary holds for $n = k + 1$ then it holds for $n = k$. We can therefore state that it holds for all $n$.

So far, we have proved the completeness in the scenario when there is one $n$-dimensional combination matrix for all requests. In the following theorem, we further consider the fine-grained integration when there are multiple combination matrices each of which is corresponding to a subset of the requests.

**Definition 5.** A fine-grained integration specification is given by $[(R_1, M_1),(R_2, M_2),\ldots,(R_k, M_k)]$, where $R_1, R_2, \ldots, R_k$ form a partition of $\mathcal{R}$ (the set of all requests over the vocabulary $\Sigma$), i.e., $\cup R_i = R$ and $\cap R_i = \emptyset$ when $i \neq j$, and each $M_i(P_1, \ldots, P_n)(1 \leq i \leq k)$ is a $n$-dimensional policy combination matrix. This specification asks requests in each set $R_i$ to be integrated according to the matrix $M_i$.

**Theorem 3.** Given a fine-grained integration specification $[(R_1, M_1),(R_2, M_2),\ldots,(R_k, M_k)]$, if for each $R_k$, there exists $d_{c1}$, $d_{c2}$, \ldots, $d_{cm}$ such that $R_k = R(d_{c1}) \cup \cdots \cup R(d_{cm})$ (where $R(d_{c_j})$ denotes the set of requests satisfying $d_{c_j}$), then there exists a FIA expression $f_I(P_1, P_2, \ldots, P_n)$ that achieves the integration requirement.

**Proof.** We first use the domain projection operator $\Pi_{nk}$ to project each policy according to $d_{c1}$, $d_{c2}$, \ldots, $d_{cm}$. For requests in each $R(d_{c_j})$, there is one fixed $M_j$. By Theorem 2, there is a FIA expression (denoted as $f_{ij}$) for integrating policies $\Pi_{nk}(P_1), \ldots, \Pi_{nk}(P_n)$ according to $M_j$: Finally, $f_i$ is the addition of all $f_{ij}$s.

We note that the above theorem requires that each $R_i$ in the partition to be expressible in a finite number of domain constraints.

### 5.4. Minimal set of operators

Recall that FIA has $(P_Y, P_N, +, \& , \neg, \Pi_{nk})$. The operator $\Pi_{nk}$ is needed to deal with fine-grained integration. Operators $(P_Y, P_N, +, \& , \neg)$ are complete in the sense that any policy combination matrix can be expressed using them. A natural question is among the set $\Theta = (P_N, P_Y, P_{NA}, +, \& , \neg, \Pi_{nk}, \Pi_{nk}, \& , \neg, \Pi_{nk})$, what subsets are minimally complete. We say a subset of $\Theta$ is minimally complete, if operators in the subset are sufficient for defining all other operators in $\Theta$, and any smaller subset cannot define all operators in $\Theta$. The following theorem answers this question. The only redundancy in $(P_Y, P_N, +, \& , \neg)$ is that only one of $P_Y$ and $P_N$ is needed.
Theorem 4. Among the 10 operators in Θ, there are 12 minimally complete subsets. They are the 12 elements in the cartesian product 
\{-\}×\{P_Y, P_N\}×\{Π_Y, Π_N, &\}×\{+, >\}.

Proof 3.

- The policy constant P_N cannot be expressed using Θ \ {P_Y, P_N}.
  When given ¬, P_Y and P_N can be derived from each other: 
P_Y = ¬P_N and P_N = ¬P_Y.

We need to show that no policy expression using operators in Θ \ {P_Y, P_N} exists that is equivalent to P_N. Consider the information ordering among the three effects, defined by: Y > NA and N > NA. The key observation is that the operators in Θ \ {P_Y, P_N} are all non-increasing in the information ordering.

Suppose, for the sake of contradiction, that a policy expression f(P_1, P_2, ..., P_n) constructed from Θ \ {P_Y, P_N} and policies P_1, ..., P_n is equivalent to P_N. Then this must mean that no matter what actual policies are used to instantiate P_1, ..., P_n, the result is P_N. Let e_0 = f(P_N, P_N, ..., P_N) = P_N. We now use a structural induction to show that e_0 must give NA for every request and thus a contradiction follows. For the base case, we have policy constant P_N, this is true. For the unary operators, if e gives NA for a request, then Π_Y(e), Π_N(e), and ¬(e) are also NA. For the binary operators +, -, & and >>, if both operands are NA for a request, then the result is also NA for the request.

- The unary operator ¬ cannot be expressed using Θ \ {¬}. The key observation is that without ¬ one cannot switch Y and N.

Suppose, for the sake of contradiction, that e_0(P) is equivalent to ¬P. Let P be a policy that returns Y on r_1 and N on r_2. Then e_0(P) must return N on r_1 and Y on r_2, i.e., it must give (N,Y) on r_1 and r_2. We use structural induction to show that the result e_0(P) gives for r_1 and r_2 must be in \{(Y,N), (Y,Y), (N,N), (NA,Y), (NA,NA), (NA,Y).\} (NA,NA), (NA,NA). That is, if the answer for r_1 is N, then the answer for r_2 must be N, and if the answer for r_2 is Y, the answer for r_1 must be Y. Hence contradiction. For the base case, this holds for P and the three constants P_Y, P_N, P_N. One can verify that the above set of six pairs is closed under Π_Y, Π_N, +, -, & and >>.

- The binary operator & cannot be expressed using Θ \ {&}. & cannot be expressed using Θ \ {&}. Assume, for the sake of contradiction, that e_0(P_1, P_2) is equivalent to P_1 & P_2. Let P_1 be a policy that returns (Y,Y) on r_1 and r_2, and P_2 be a policy that returns (Y,N) on r_1 and r_2. Then e_0(P_1 & P_2) must return (Y,NA) on r_1 and r_2. We show that this is not possible. The key insight here is that without & Π_Y Π_N, one cannot get information asymmetry Y or N for one request and NA for another from symmetric policies. We use a structural induction to show that the result e_0(P_1 & P_2) gives for r_1 and r_2 must be an element of the following set: \{(Y,N), (Y,Y), (N,N), (NA,Y), (NA,NA), (NA,Y). (NA,NA).\} This holds for all P_1, P_2, P_Y, P_N, P_N. One can verify that the set is closed under ¬, +, -, & and >>.

Given ¬, Π_Y and Π_N can be expressed from each other: 
Π_Y(P) = P & ¬P_N, Π_N(P) = Π_Y(¬P). Π_Y can be expressed using {& Π_Y} by definition, and & can be expressed using {Π_N, +, ¬, Π_N}. 
Given {P_N, +, ¬}, & can be expressed using either Π_Y or Π_N:
P_1 & P_2 = (P_1 ∩ P_2) + (¬P_1 ∩ ¬P_2),

where 
P_1 ∩ P_2 = Π_Y(Π_Y(P_1) - (Π_N(P_N + P_2))),

and & can be defined using Π_Y, Π_N as:
P_1 & P_2 = (Π_Y((¬P_1 + P_2 + ¬P_2)) + (Π_N(P_1 + P_2 + ¬P_2))).

The effect of P_1 ∩ P_2 is to authorize any request that is authorized by both P_1 and P_2, and to be NA for all other requests.

- The binary operator + cannot be expressed using Θ \ {+. >>}. Suppose, for the sake of contradiction, that an expression e_0(P_1, P_2) constructed from P_1, P_2 and Θ \ {+. >>} is equivalent to P_1 + P_2. Let P be a policy that returns (Y,NA) on r_1 and r_2, and P_2 be a policy that returns (NA,NA) on r_1 and r_2. Then e_0(P_1, P_2) must return (Y,NA) on r_1 and r_2.

We use a structural induction to show that the result e(P_1, P_2) gives for r_1 and r_2 must be an element of the set \{(Y,Y), (Y,NA), (N,N), (N,Y), (NA,Y), (NA,NA), (NA,NA), (NA,NA).\} Hence contradiction. This is satisfied by P_1, P_2, P_Y, P_N, P_N. One can verify that these seven values are closed under ¬, Π_Y, Π_N, +, -.

However, + can be expressed using \{P_N, ¬, &, &\} or \{P_N, ¬, &\}, and >> can be expressed using \{P_N, +, &\}.

In particular, the + operator can be expressed using \{P_N, +, ¬, &\} as follows:
P_1 + P_2 = (P_1 & P_2) + (P_1 & P_2) + (P_1 & P_2) + (P_1 & P_2).

Similarly, the >> operator can be expressed using \{P_N, ¬, Π_Y, Π_N\}:
P_1 >> P_2 = (Π_Y(P_1) >> (Π_N(P_1) >> (Π_N(P_1) >> (Π_N(P_1)))

Recall that the >> operator is defined using + and - as P_1 >> P_2 = P_1 + (P_2 - P_1), and P_1 - P_2 = (P_Y & (¬(P_1 + P_2 - P_2))) + (P_N & (P_1 + P_2 - P_2)). Thus, >> can be expressed using \{P_N, ¬, &\}.

In summary, among the 10 operators in Θ, for completeness, we must have one in \{P_Y, P_N\}, one in \{Π_Y, Π_N, &\}, and one in \{+, >>\}. There are 12 combinations. It is not difficult to verify that every such combination is in fact complete. For example, once we have \{¬, P_Y, Π_Y\}, adding + allows us to derive &, and then derive >>, adding >> allows us to derive + and then &. There are thus 12 minimally complete subsets in Θ.

5.5. Reasonableness properties

As we have proved that FIA can express any possible policy integration, a natural question that may arise is that whether an integrated policy is reasonable based on human understanding. Therefore, for users to easily edit a policy for a desired effect, we provide some guidelines that guarantee certain properties of the integrated policy if it is specified by using a certain set of operators. These guidelines are referred to as reasonableness properties the concept of which was first introduced by Tschantz and Krishnamurthi (2006). In what follows, we present several reasonableness properties with respect to FIA. We use P_1, P_2, ..., P_n to denote original policies and P_l to denote the integrated policy.

2 Though in most cases whether a policy is reasonable depends on the application context.
Property 1 and 2 guarantee that the decision obtained from the integrated policy is consistent with that obtained from the original policies when all the original policies reach some type of agreement.

**Property 1.** Let $P_1$ be the integrated policy obtained by using any combination of operators in $\{+,&,>\}$. For any request $r$, if $P_1, \ldots, P_n$ yield the same decision, $P_I$ will also yield the same decision.

**Property 2.** Let $P_1$ be the integrated policy obtained by using any combination of operators in $\{\Theta_P,\wedge_P,\vee_P,\&_P,\|_P,\|_N,\|_N\}$. For any request $r$ that is not applicable to all policies $P_1, \ldots, P_n$, it is also not applicable to $P_I$.

One partial order on the set of access control decisions $\{Y,N,NA\}$ is the information order: $Y > NA, N > NA$, as both $Y$ and $N$ provide more information about a request than $NA$. The following properties are expressed based on a strict order $Y > N > NA$ which can be obtained by using such information order and preferring $Y$ to $N$.

Property 3 guarantees that the integrated policy will never grant a decision that is higher with respect to the strict information order than any of the original policies. For example, if a request is denied by one of the original policies, this request will never be able to allowed by the integrated policy because $Y > N$.

**Property 3.** Let $P_1$ be the integrated policy obtained by using any combination of operators in $\{\Theta_P,\wedge_P,\vee_P,\&_P,\|_P,\|_N,\|_N\}$. For any request $r$, $e_i(P_1) \leq \min(e_i(P_1), \ldots, e_i(P_n))$, where $e_i(P)$ denotes the decision of policy $P$ regarding $r$.

In contrast to Property 3, Property 4 guarantees that a request will obtain a decision from the integrated policy not lower with respect to the strict information order than that from any of the original policies.

**Property 4.** Let $P_1$ be the integrated policy obtained by using any combination of operators in $\{\Theta_P,+,>\}$. For any request $r$, $e_i(P_1) \geq \min(e_i(P_1), \ldots, e_i(P_n))$, where $e_i(P)$ denotes the decision of policy $P$ regarding $r$.

The listed properties can be easily verified and hence we omit the proof here. It is also worth noting that these properties are not exhaustive. According to different needs, more properties can be discovered and taken into account in integrated policy specification.

### 6. Integrated policy generation

In this section, we present an approach to automatically generate the integrated policy given the FIA policy expression. Internally, we represent each policy as a Multi-Terminal Binary Decision Diagram (MTBDD) (Fujita et al., 1997), and then perform operations on the underlying MTBDD structures to generate the integrated policy. We have chosen an MTBDD based implementation of the proposed algebra because (i) MTBDDs have proven to be a simple and efficient representation for XACML policies (Fisler et al., 2005) and (ii) operators in FIA can be mapped to efficient operations on the underlying policy MTBDDs. Our approach consists of three main phases:

1. **Policy representation:** For each policy $P_i$ in the FIA expression $f(P_1, P_2, \ldots, P_n)$, we construct a policy MTBDD, $T_i^P$.
2. **Construction of the integrated policy MTBDD:** We combine the individual policy MTBDD structures according to the operators in the FIA expression to construct the integrated policy MTBDD.
3. **Policy generation:** The integrated policy MTBDD is then used to generate the actual integrated XACML policy.

In what follows we will present the details of each step.

#### 6.1. Policy representation

Recall from Section 3 that we characterize a policy $P$ as a 3-tuple $R_i^P, R_i^P, R_i^{NA}$, where $R_i^P$ is the set of requests permitted by the policy, $R_i^P$ is the set of requests denied by the policy and $R_i^{NA}$ is the set of requests not applicable to the policy. Alternatively, we can define $P$ as a function $P : R \rightarrow E$ from the domain of requests $R$ onto the domain of effects $E$, where $E = \{Y, N, NA\}$.

An XACML policy can be transformed into a compound Boolean expression over request attributes (Anderson, 2003). A compound Boolean expression is composed of atomic Boolean expressions (AE) combined using the logical operations $\vee$ and $\wedge$. Atomic Boolean expressions that appear in most policies belong to one of the following two categories: (i) one-variable equality constraints, $a \circ c$, where $a$ is an attribute name, $c$ is a constant, and $\circ \in \{=, \neq\}$; (ii) one-variable range constraints, $a \circ c_1 \circ c_2$, where $a$ is an attribute name, $c_1$ and $c_2$ are constants, and $\circ \in \{<, \leq\}$.

**Example 7.** Policy $P_1$ from Example 1 can be defined as a function:

$$P_1(r) = \begin{cases} Y & \text{if } \text{role} = \text{manager} \land (\text{act} = \text{read} \lor \text{act} = \text{update}) \land \text{time} \leq \text{6pm} \\ N & \text{if } \text{role} = \text{staff} \land \text{act} = \text{read} \\ \text{NA} & \text{Otherwise} \end{cases}$$

where $r$ is a request of the form $\{(\text{role}, v_1), (\text{act}, v_2), (\text{time}, v_3)\}$.

We now encode each unique atomic Boolean expression $AE_i$ in a policy into a Boolean variable $x_i$ such that: $x_i = 0$ if $AE_i$ is false; $x_i = 1$ if $AE_i$ is true. The Boolean encoding for policy $P_i$ is given in Table 6.

Using the above Boolean encoding, a policy $P$ can be transformed into a function $P : B^R \rightarrow E$, over a vector of Boolean variables, $X = x_0, x_1, \ldots, x_n$, onto the finite set of effects.
E = \{Y, N, NA\}, where n is the number of unique atomic Boolean expressions in policy P. A request r corresponds to an assignment of the Boolean vector \( \mathbf{x} \), which is derived by evaluating the atomic Boolean expressions with attribute values specified in the request.

**Example 8.** After Boolean encoding, the policy \( P_1 \) is transformed into the function:

\[
P_1(\mathbf{x}) = \begin{cases} 
  Y & \text{if } x_0 \land (x_2 \lor x_3) \land x_4 \\
  N & \text{if } x_1 \land \neg x_4 \\
  \text{NA} & \text{Otherwise}
\end{cases}
\]

The transformed policy function can now be represented as a MTBDD. A MTBDD provides a compact representation of functions of the form \( f : \mathbb{B}^n \rightarrow \mathbb{R} \), which maps bit vectors over a set of variables (\( \mathbb{B}^n \)) to a finite set of results (\( \mathbb{R} \)). The structure of a MTBDD is a rooted acyclic directed graph. The internal (or non-terminal) nodes represent Boolean variables and the terminals represent values in a finite set. Each non-terminal node has two edges labeled 0 and 1 respectively. Thus when a policy is represented using a MTBDD, the non-terminal nodes correspond to the unique atomic Boolean expressions and the terminal nodes correspond to the effects. Each path in the MTBDD represents an assignment for the Boolean variables along the path, thus representing a request \( r \). The terminal on a path represents the effect of the policy for the request represented by that path. Note that different orderings on the variables may result in different MTBDD representations and hence different sizes of the corresponding MTBDD representation. Several approaches for determining the variable ordering that results in an optimally sized MTBDD can be found in Grumberg et al. (2003). For examples discussed in this paper, we use the variable ordering \( x_0 < x_1 < x_2 < x_3 < x_4 \). The MTBDD of the policy \( P_1 \) is shown in Fig. 2, where the dashed lines are 0-edges and solid lines are 1-edges.

Compound Boolean expression representing the policies to be integrated may have atomic Boolean expressions with matching attribute names but overlapping value ranges. In such cases, we need to transform the atomic Boolean expressions with overlapping value ranges into a sequence of new atomic Boolean expressions with disjoint value ranges, before performing the Boolean encoding. This transformation ensures independency of nodes in the MTBDD and avoids redundant checking. Moreover, it helps unify nodes in different policies. A generic procedure for computing the new atomic Boolean expression is described below.

Assume that the original value ranges of an attribute \( a \) are \([d_1, d_1'], [d_2, d_2'], ..., [d_m, d_m']\) (the superscript `\('` and `\('` denote lower and upper bound respectively). We sort the range bounds in an ascending order, and then employ a plane sweeping technique to obtain the disjoint ranges: \([d_1', d_2'], [d_3, d_4'], ..., [d_m, d_m']\), which satisfy the following three conditions: (i) \( d_i' < d_{i+1}' \) if \( D = [d_1, d_2, ..., d_m] \); (ii) \( U_{i-1}^m = [d_i', d_i] = U_i^{m-1} = [d_i, d_i'] \); and (iii) \( \mathcal{R}_{i-1} = [d_i', d_i] \).

Consider policy \( P_2 \) from Example 1. We can observe that the atomic Boolean expression \( 8am \leq \text{time} \leq 6pm \) in \( P_1 \) refers to the same attribute as in the atomic Boolean expression \( 8am \leq \text{time} \leq 8pm \) in \( P_2 \) and their value ranges overlap. In order to distinguish these two atomic Boolean expressions during the later policy integration, we split the value ranges and introduce the new atomic Boolean expression \( 6pm \leq \text{time} \leq 8pm \). The expression \( 8am \leq \text{time} \leq 8pm \) in \( P_2 \) is replaced with \( (8am \leq \text{time} \leq 6pm) \lor (6pm \leq \text{time} \leq 8pm) \). Boolean encoding is then performed for the two policies by considering unique atomic Boolean expressions across both policies.

**Example 9.** By introducing another atomic Boolean expression \( 6pm \leq \text{time} \leq 8pm \), i.e. \( x_5 \), the transformed function for policy \( P_2 \) is:

\[
P_2(\mathbf{x}) = \begin{cases} 
  Y & \text{if } (x_0 \lor x_3) \land (x_4 \lor x_5) \\
  N & \text{if } x_1 \land x_5 \\
  \text{NA} & \text{Otherwise}
\end{cases}
\]

Using the same variable ordering \( x_0 < x_1 < x_2 < x_3 < x_4 < x_5 \), we construct the MTBDD for \( P_2 \) shown in Fig. 2.

### 6.2. Construction of integrated policy MTBDD

Given the FIA expression \( f(P_1, P_2, ..., P_n) \) and the MTBDD representations \( T_1, T_2, ..., T_n \) of the policies \( P_1, P_2, ..., P_n \) respectively, we construct the integrated policy MTBDD \( T \), by performing the operations (specified in \( f \)) on the individual policy MTBDDs.

Operations on policies can be expressed as operations on the corresponding policy MTBDDs. Many efficient operations have been defined and implemented for MTBDDs (Fujita et al., 1997). In particular, we use the Apply operation defined on MTBDDs to perform the FIA binary operations \( \lor, \land, \neg \) and not operation defined on MTBDD to perform the FIA unary negation \( \neg \) operation. We introduce a new MTBDD operation called Projection to perform the effect projection and domain projection operations defined in FIA.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \mathbf{A}_{E_i} )</th>
<th>( x_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>role = manager</td>
<td>( x_0 )</td>
</tr>
<tr>
<td>1</td>
<td>role = staff</td>
<td>( x_1 )</td>
</tr>
<tr>
<td>2</td>
<td>act = read</td>
<td>( x_2 )</td>
</tr>
<tr>
<td>3</td>
<td>act = update</td>
<td>( x_3 )</td>
</tr>
<tr>
<td>4</td>
<td>( 8am \leq \text{time} \leq 6pm )</td>
<td>( x_4 )</td>
</tr>
</tbody>
</table>
Procedure Apply(Node₁, Node₂, OP)
Input: Node₁, Node₂ are MTBDD nodes, OP is a policy operation
1. initiate Node₁ // Node₂ is the combination result
2. if Node₁ and Node₂ are terminals then
3.   Node₁ ← (Node₁ OP Node₂, null, null)
4. else
5.   if Node₁.var = Node₂.var then
6.     Node₁.var ← Node₁.var
7.     Node₁.left ← Apply(Node₁.left, Node₂.left, OP)
8.     Node₁.right ← Apply(Node₁.right, Node₂.right, OP)
9.   if Node₁.var precedes Node₂.var then
10.    Node₁.var ← Node₂.var
11.   Node₁.left ← Apply(Node₁.left, Node₂, OP)
12.   Node₁.right ← Apply(Node₁.right, Node₂, OP)
13. if Node₂.var precedes Node₁.var then
14.    Node₂.var ← Node₁.var
15.   Node₂.left ← Apply(Node₂.left, Node₁, OP)
16.   Node₂.right ← Apply(Node₂.right, Node₁, OP)
17. return Node₁
end Apply.

Fig. 3 – Description of the Apply procedure.

The Apply operation combines two MTBDDs by a specified binary arithmetic operation. A high level description of the Apply operation is shown in Fig. 3, where var, left, right refer to the variable, left child and right child of a MTBDD node, respectively. The Apply operation traverses each of the MTBDDs simultaneously starting from the root node. When the terminals of both MTBDDs are reached, the specified operation is applied on the terminals to obtain the terminal for the resulting combined MTBDD. A variable ordering needs to be specified for the Apply procedure.

Example 10. The integrated MTBDD T₁ for the policy expression f(P₁, P₂) = P₁ + P₂ is obtained by using MTBDD operation Apply(T₁.root, T₂.root, +), where “root” refers to the root node of the corresponding MTBDD. Fig. 4 shows the integrated policy MTBDD. The same variable ordering x₀ < x₁ < x₂ < x₃ < x₄ < x₅ has been used in the construction of the integrated policy MTBDD.

The procedure for the Projection operation differs depending on the type of constraints. The procedure for performing effect projection operations is the following. For H₁v, those paths in T¹ that lead to N are redirected to the terminal NA. Similarly, for H₂v, those paths in T² that lead to Y are redirected to the terminal NA.

For the domain projection operation with domain constraint dc, we traverse the policy MTBDD from the top to the bottom and check the atomic Boolean expression associated with each node (denoted as Node). If the atomic Boolean expression of Node contains an attribute specified in dc, we compute the intersection of dc.var.domain (the domain of Node.var specified in dc) and the attribute’s current domain Node.var.domain. If the intersection is not empty, we update the attribute’s domain to the intersection. If the intersection is empty, we will remove Node. After removing Node, we need to adjust the pointer from its parent node by redirecting it to Node’s left child which leads to the path when N is not considered. After all nodes have been examined, those nodes that have no incoming edges are also removed. At the end, we check whether all attributes in the original policy appear at least once in the new policy obtained from the projection. If some attributes have been fully removed after the projection, it means that the original policy does not satisfy the domain constraints and hence the resultant policy will be empty. Fig. 6 summarizes the algorithm. For better understanding, an example of project operation on policy P₁ is given below.

Example 11. Fig. 5 shows the resultant MTBDD after the projection operation Πg₁P₁, where dc = (role.manager, (act. {read, update}), (time, [10am, 4pm])). Compared to the MTBDD of P₁ in Fig. 2, we can see that node x₂ is removed since it does not satisfy the domain constraint (role.manager), and node x₄ is modified to x₄ with the new domain [10am, 4pm].

If the projection operation contains both types of constraints, we apply one constraint first and then apply the other by using the previous two algorithms.

Thus, given an arbitrary FIA expression f(P₁, P₂, ..., Pₙ), we can use a combination of the Apply, not, Projection MTBDD operations on the policy MTBDDs to generate the integrated policy MTBDD. An example is given below.

Example 12. Consider the FIA policy expression for the only-one-applicable policy combining algorithm together with the domain constraint dc = (role.manager, (act. {read, update}), (time, [10am, 4pm])). Here, f(P₁, P₂) = Πg₁(Π₆(Π₁(P₁ - P₁) + (P₂ - P₁))). The integrated MTBDD can be obtained by using the Apply and Projection operations as follows:

Projection(Apply(Apply(T¹.root, T².root, -), Apply(T³.root, T⁴.root, -), +), dc).
6.3. XACML policy generation

In the previous section, we have presented how to construct the integrated MTBDD given any policy expression \( f \). Though such integrated MTBDD can be used to evaluate requests with respect to the integrated policy, they cannot be directly deployed in applications using the access control system based on XACML. Therefore, we develop an approach that can automatically transform MTBDDs to actual XACML policies. The policy generation consists of three steps:

1. Find the paths in the combined MTBDD that lead to the Y and N terminals, and represent each path as a Boolean expression over the Boolean variable of each node.
2. Map the above Boolean expressions to the Boolean expressions on actual policy attributes.
3. Translate the compound Boolean expression obtained in step 2 into a XACML policy.

We first elaborate on step 1. In the MTBDD, each node has two edges, namely 0-edge and 1-edge. The 0-edge and 1-edge of a node labelled \( x \), correspond to edge-expressions \( \overline{x} \) and \( x \), respectively. A path in the MTBDD corresponds to an expression which is the conjunction of edge-expressions of all edges along that path. We refer to this as a path-expression. Path-expressions corresponding to the same terminal have “OR” relationship. Fig. 7 shows an example of \( P_1 + P_2 \), where the left and right pictures show the paths leading to the Y and N terminals and the corresponding Boolean expressions, respectively.

Next, we replace Boolean variables in the path-expressions with the corresponding atomic Boolean expressions by using the mapping constructed in the Boolean encoding phase. During the transformation in each path-expression, we need to remove some redundant information. For instance, the resulting expression may contain an attribute with both equality and inequality functions like \( (\text{role} = \text{manager}) \land (\text{role} \neq \text{staff}) \). In that case, we only need to keep the equality function of the attribute.

Example 13. After the replacement, the Boolean expression of the N terminal in Fig. 7 is transformed as follows:

\[
(\text{role} = \text{staff} \land \text{act} = \text{update}) \lor (\text{role} = \text{staff} \land \text{act} = \text{read} \land \text{time} \neq [8am, 6pm] \land \text{time} = [6pm, 8pm])
\]

The last step is to generate the actual XACML policy from the compound Boolean expression obtained in previous step. Specifically, for each path-expression whose evaluation is Y, a permit rule is generated; and for each path-expression whose evaluation is N, a deny rule is generated. Attributes that appear in conditions of the rules in original policies still appear in conditions of the newly generated rules, and attributes that appear in targets in the original policies still appear in targets in the integrated policy. Here, we do not distinguish the policy target with rule target. All targets appear as rule targets. Finally, we select the “first-applicable” as the rule combining algorithm for the
integrated policy since the integrated policy will not contain conflicting rules.

Example 14. Consider policies $P_1$ and $P_2$ in Example 1, and the Boolean expression in Example 13. We generate the corresponding deny rules for the integrated policy of $P_1 + P_2$ as shown in Fig. 8.

7. Experimental study

We performed experiments to evaluate the time taken for performing FIA operations and the time for generating an integrated policy. We also examined the size of the generated integrated policy in terms of the number of rules and number of atomic Boolean expressions in each rule.

7.1. Experimental Settings

All experiments were conducted on a Pentium III 3 GHz 500 MB machine. MTBDD operations were implemented using the modified CUDD library developed in Fisler et al. (2005).

We implemented a random attribute based access control policy generator to generate XACML policies in Boolean form. Each policy contained atomic Boolean expressions on a set of predefined attribute names and values. The Boolean expressions corresponding to the Condition element of an XACML policy was derived by randomly concatenating atomic Boolean expressions with the logical $\lor$, $\land$ and $\neg$ operators. Each rule was randomly assigned to either permit or deny effect. Each policy was also associated with either a deny-override or permit-override rule combining algorithm.

7.2. Processing FIA expressions

In the first set of experiments, we measure the average time required for performing the FIA operations and the size of the obtained MTBDDs. Fig. 9 shows along the left y-axis the average time (in ms) for performing $+$ and $\preceq$ operations on policies in which the number of rules per policy was varied between 25 and 150 and the number of atomic Boolean expressions per policy was varied between 50 and 300. This graph shows along the right y-axis the average size, i.e., the number of nodes, in the corresponding integrated MTBDDs. To maintain readability of Fig. 9, we did not report the result for performing other operations which demonstrated similar trends.

From Fig. 9, we can observe that the average time taken to perform these operations increases with the increase in the size of the integrated MTBDDs, and it differs for different operators. The reason is that the actual time for performing operations depend on the size of the resulting integrated MTBDD. The larger the MTBDD is, the longer time the integration will take. Performing the $\preceq$ operation usually resulted in MTBDDs with a smaller size and hence it took lesser time. Considering that for typical policies we have encountered in real world applications the average number of atomic Boolean expressions lies between 10 and 50, the time trend observed in Fig. 9 is very encouraging.

7.3. Characteristics of integrated policies

We now study the characteristics of the integrated policy. Because the number of rules generated in the integrated policy is equal to the number of paths which can be exponential in the size of the integrated MTBDD a large number of rules can be generated. For the consideration of enforcement efficiency, we used ESPRESSO (Rao and Jacob, 1998), a two level logic minimizer, to reduce the number of rules. ESPRESSO uses state-of-the-art heuristic Boolean minimization algorithms to produce a minimal equivalent representation of two-valued or multiple-valued Boolean functions. The minterms from the integrated MTBDD were transformed to ESPRESSO inputs and the minimized output was used for policy generation. Table 7 summarizes the results obtained for $+$ operation performed on data sets that contained policies with 4 and 8 rules with an average 20 atomic Boolean expressions per policy. We observe that using ESPRESSO a substantial decrease in the number of rules and atomic Boolean expressions/terms was obtained. For the data sets

---

PolicyId=P1+P2, CombiningAlgorithm="First-applicable"
<RuleId=R1 Effect=Deny>
  <Target>
    <Subject role=staff>
    <Action act=update>
  </Target>
</Rule>
<RuleId=R2 Effect=Deny>
  <Target>
    <Subject role=staff>
    <Action act=read>
  </Target>
  <Condition time $\neq [8am, 6pm] \text{ AND time } \neq [6pm, 8pm]>$
</Rule>
<RuleId=R3 Effect=Permit> ...

---

Fig. 8 — The integrated XACML policy representing $P_1 + P_2$. 

Fig. 9 — Average time and average integrated MTBDD size for policies with varying number of rules and atomic Boolean expressions per policy.
used in our experiments we observed a 75–99% reduction in the number of rules and 35–71% reduction in the number of atomic Boolean expressions per rule.

7.4. Processing time of actual policy generation

Finally, we evaluated the time required for generating the integrated XACML policy. It is worth noting that this step is optional. Users can also directly use the integrated MTBDD for request evaluation. Fig. 10 shows the policy generation times for integrated MTBDDs with different number of paths. The time for policy generation is observed to be proportional to the number of paths in the integrated MTBDD. The number of paths in the integrated MTBDD is in turn determined by the nature of the compound Boolean expressions in the policies and the chosen variable ordering.

8. Obligations

In the discussions so far we have not considered the impact of the presence of obligations when integrating policies using FIA. In this section, we briefly discuss how to generate the correct set of obligations for an XACML policy that is generated as result of integrating policies using operations in FIA. We plan to extend the current algebra with the notion of obligations as part of future work.

An XACML policy can have obligations associated with “Permit” and/or “Deny” effects. According to the XACML standard specification, the set of obligations returned by the policy decision point for a given request is derived from the evaluation tree of policy sets and policies. The final set of obligations returned by the policy decision point includes obligations of policies along those paths of the tree where effect at each level is the same as the final effect returned by the policy decision point.

Treating the obligations similarly in our framework, we can obtain the set of obligations that must be enforced by integrated policy. The set of obligations can be easily derived when using the & or – operations for integrating policies. For example, consider the & operation. In this case, since a request in the integrated policy is permitted(denied) only when both the policies permit(deny) the request, the set of “Permit”(“Deny”) obligations enforced by the integrated policy must be the union of the set of “Permit”(“Deny”) obligations specified in the individual policies that are being combined. Similarly, when combining policies P₁ and P₂ using the – operation, a request is permitted(denied) in the combined policy when the request is permitted(denied) by P₁ and is not applicable to P₂ and hence the set of “Permit”(“Deny”) obligations enforced by the integrated policy is exactly the same as the “Permit”(“Deny”) obligations of P₁.

Deriving the set of obligations that must be enforced by the integrated policy is not straightforward in the case of the + and ÷ operations because when a request is permitted (denied) in the policy integrated using these operations, the decision of the individual policies participating in the integration with respect to this request is not known. However, this problem can be solved by generating an integrated XACML policy set instead of an integrated XACML policy from the integrated MTBDD. Note that each terminal in the integrated MTBDD corresponding to integration of two policies P₁ and P₂ using the + (÷) operation represents a cell in the matrix representation of + (÷) shown in Fig. 1. Thus the terminals correspond to E₁₁ − E₁₂(E₁,₂ ∈ Y, N, NA) and 1 ≤ i ≤ 9 where E₁₁ represents effect of P₁ and E₁₂ represents effect of P₂ in cell i. Each terminal is mapped to a final decision E₂ which could be one of Y, N or NA depending on the operation used to generate the integrated MTBDD. Using the same techniques for generating a single policy from a given MTBDD, we can generate a sub policy Sᵢ corresponding to cell E₁₁ − E₁₂ by collecting all paths leading to the E₁₁ − E₁₂ terminal. The correct set of obligations for each of these sub policies in the integrated policy set can be obtained based on the individual decisions E₁₁ and E₁₂ and the final decision E₂ given by the specific operation. That is, the obligations for a sub policy Sᵢ generated by traversing paths leading to E₁₁ − E₁₂ includes the obligations of policy P₁ corresponding to effect E₁₁ (“Permit” obligations if E₁₁ = Y or “Deny” obligations if E₁₁ = N) only if E₁₁ is the same as Eₙᵢ for k = 1, 2.

For example, consider an integrated MTBDD that corresponds to the + operation on two policies P₁ and P₂. Then, (i) For the policy corresponding to the Y − Y terminal the set of obligations is the union of the set of “Permit” obligations of P₁ and P₂, (ii) For the policy corresponding to the Y − N terminal the set of obligations is the union of the set of “Deny” obligations of P₁ and P₂, (iii) For the policies corresponding to the Y − NA (N − Y) and Y − NA (NA − Y) the set of obligations consists only of the set of “Permit” obligations of P₁(P₂), (iv) For the policy corresponding

<table>
<thead>
<tr>
<th># of Rules in a Policy</th>
<th>Rule Type</th>
<th>Without ESPRESSO</th>
<th>With ESPRESSO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg # of Rules</td>
<td>Avg # of terms per Rule</td>
<td>Avg # of Rules</td>
</tr>
<tr>
<td>4 Rules</td>
<td>Permit</td>
<td>790</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Deny</td>
<td>3625</td>
<td>21</td>
</tr>
<tr>
<td>8 Rules</td>
<td>Permit</td>
<td>20192</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Deny</td>
<td>131348</td>
<td>36</td>
</tr>
</tbody>
</table>

Fig. 10 – Average time (in seconds) for integrated policy generation.
to the $N\wedge(NA\wedge N)$ terminal the set of obligations consists only of the set of “Deny” obligations of $P_i(P_3)$.

Similar method can be used to derive the obligations for an XACML policy obtained by integrating $n$ policies $P_1, P_2, \ldots, P_n$ using arbitrary FIA operations. In this case, each terminal of the integrated MTBDD will correspond to $E_{i1} \wedge E_{i2} \wedge \cdots \wedge E_{in} \quad 1 \leq i \leq 3$. The set of obligations for the sub policy $S_i$ will include the obligations of policy $P_k$ corresponding to effect $E_{ik}$ (“Permit” obligations if $E_{ik} = \top$ or “Deny” obligations if $E_{ik} = \bot$) only if $E_{ik}$ is the same as $E_{ij}$ for $k = 1, 2, \ldots, n$. Note that the integrated policy set is only generated from the final integrated MTBDD and not for the MTBDDs obtained when performing the intermediate operations.

9. Conclusions

In this work we have proposed an algebra for the fine-grained integration of language independent policies. Our operations cannot only express existing policy-combining algorithms but can also express any arbitrary combination of policies at a fine granularity of requests, effects and domains, as we have proved in the completeness theorem. We have implemented this algebra and our experimental results indicate that the FIA operations can be performed efficiently.

We plan to extend this work in several interesting directions. One direction involves the extension with respect to obligations as discussed in the previous section. Another interesting question that arises when using such integration algebras is: how can one be sure that a given algebraic expression will behave as expected. We believe that we can leverage techniques from software engineering field to examine the integrated policy with a random number of requests and assure with some high probability that the expression indeed corresponds to the expected behavior. Another direction involves exploring different variable re-ordering techniques for MTBDD construction that can possibly further reduce the operation time and policy generation time.

References


Extensible access control markup language (XACML) version 2.0. OASIS Standard, 2005.


Prathima Rao received her PhD degree from the Department of Computer Science at Purdue University in August 2010 under Prof. Elisa Bertino. She obtained her Master’s degree from Electrical & Computer Engineering Department at Purdue in 2003. She got her Bachelor’s degree in Computer Science & Engineering from P.E.S Institute of Technology, Bangalore, India in 2000. She is currently a member of technical staff in the platform security team at Salesforce.com. Her main area of research is policy based security and in particular analysis of access control policies expressed in XACML. She is also interested in privacy preserving collaborative access control, novel applications of information retrieval techniques and issues relating to development of domain specific languages.

Dan Lin joined Missouri University of Science and Technology as an assistant professor in Aug. 2008. Before that, she was a postdoctoral research associate at Purdue University working with Prof. Elisa Bertino. During July to Sep. 2005, she was a visiting scholar at Aalborg University (Denmark) working with Prof. Christian S. Jensen. She obtained her B.S. from Fudan University (Shanghai, China) in 2002 and her Ph.D. from National University of Singapore in 2007 under the supervision of Prof. Beng Chin Ooi. Her main research interests cover many areas in the fields of database systems and information security. Her current research includes geographical information systems, spatial-temporal databases, location privacy, and access control policy analysis.

Elisa Bertino joined Purdue in January 2004 as professor in Computer Science and research director at CERIAS. Her research interests cover many areas in the fields of information security and database systems. Her research combines both theoretical and practical aspects, addressing applications on a number of domains, such as medicine and humanities. Professor Bertino is a Fellow of the Institute of Electrical and Electronics Engineers and a Fellow of ACM, and received the IEEE Computer Society Technical Achievement award in 2002 for outstanding contributions to database systems and database security and advanced data management systems. She recently received the 2005 Tsutomu Kanai Award by the IEEE Computer Society for pioneering and innovative research contributions to secure distributed systems. She is currently serving in the IEEE Computer Society Board of Governors and as Chair of ACM SIGSAC.

Ninghui Li is an Associate Professor of Computer Science at Purdue University, where he joined in 2003. He received a Bachelor’s degree from the University of Science and Technology of China in 1993 and a Ph.D. in Computer Science from New York University in 2000. Before joining Purdue, he was a Research Associate at Stanford University Computer Science Department for 3 years. Prof. Li’s research interests are in computer and information security and privacy. He has published over 80 referred papers in journals and conference proceedings. He has served on the Program Committees of more than 50 international conferences and workshops. In 2005, he received an NSF CAREER award.

Jorge Lobo joined IBM T. J. Watson Research Center in 2004. Previous to IBM he was principal architect at Teltier Technologies, a start-up company in the wireless telecommunication space acquired by Dynamicsoft and now part of Cisco System. Before Teltier he was an Associate Professor of CS at the University of Illinois at Chicago and a member of the Network Computing Research Department at Bell Labs. Jorge Lobo has more than 50 publications in international journals and conferences in the areas of Networks, Databases and AI. He is co-author of an MIT Press book on logic programming and an IBM Press book on policy technologies for self-managing systems. He is co-founder and member of the steering committee for the IEEE International Symposium on Policies for Distributed Systems and Networks. He has a PhD in CS from the University of Maryland at College Park, and an MS and a BE from Simon Bolivar University, Venezuela.