Discovery and Resolution of Anomalies in Web Access Control Policies

Hongxin Hu, *Member, IEEE,* Gail-Joon Ahn, *Senior Member, IEEE,* and Ketan Kulkarni

Abstract—Emerging computing technologies such as Web services, service-oriented architecture, and cloud computing has enabled us to perform business services more efficiently and effectively. However, we still suffer from unintended security leakages by unauthorized actions in business services while providing more convenient services to Internet users through such a cutting-edge technological growth. Furthermore, designing and managing Web access control policies are often error-prone due to the lack of effective analysis mechanisms and tools. In this paper, we represent an innovative policy anomaly analysis approach for Web access control policies, focusing on XACML (eXtensible Access Control Markup Language) policy. We introduce a policy-based segmentation technique to accurately identify policy anomalies and derive effective anomaly resolutions, along with an intuitive visualization representation of analysis results. We also discuss a proof-of-concept implementation of our method called `XAnalyzer` and demonstrate how our approach can efficiently discover and resolve policy anomalies.

Index Terms—Access control policies, XACML, conflict, redundancy, discovery and resolution

1 Introduction

With the tremendous growth of Web applications and Web services deployed on the Internet, the use of a policy-based approach has recently received considerable attention to accommodate the security requirements covering large, open, distributed and heterogeneous computing environments. XACML (eXtensible Access Control Markup Language) [25], which is a general purpose access control policy language standardized by the Organization for the Advancement of Structured Information Standards (OASIS), has been broadly adopted to specify access control policies for various applications, especially Web services [28]. In an XACML policy, multiple rules may overlap, which means one access request may match several rules. Moreover, multiple rules within one policy may conflict, implying that those rules not only overlap each other but also yield different decisions. Conflicts in an XACML policy may lead to both safety problem (e.g. allowing unauthorized access) and availability problem (e.g. denying legitimate access).

An intuitive means for resolving policy conflicts by a policy designer is to remove all conflicts by modifying the policies. However, resolving conflicts through changing the policies is notably difficult, even impossible, in practice from many aspects. First, the number of conflicts in an XACML policy is potentially large, since an XACML policy may consist of hundreds or thousands of rules. Second, conflicts in XACML policies are probably very complicated, because one rule may conflict with multiple other rules, and one conflict may be associated with several rules. Besides, an XACML policy for a distributed application may be aggregated from multiple parties. Also, an XACML policy may be maintained by more than one administrator. Without a priori knowledge on the original intentions of policy specification, changing a policy may affect the policy's semantics and may not resolve conflicts correctly. Furthermore, in some cases, a policy designer may intentionally introduce certain overlaps in XACML policy components by implicitly reflecting that only the first rule is important. In this case, conflicts are not an error, but intended, which would not be necessary to be changed.

Since the *conflicts* in XACML policies always exist and are hard to be eliminated, XACML defines four different combining algorithms to automatically resolve conflicts [25]: *Deny-Overrides*, *Permit-Overrides*, *First-Applicable* and *Only-One-Applicable*. Unfortunately, XACML currently lacks a systematic mechanism for precisely detecting conflicts. Identifying conflicts in XACML policies is critical for policy designers since the correctness of selecting a combining algorithm for an XACML policy or policy set component heavily relies on the information from conflict diagnosis. Without precise conflict information, the effectiveness of combining algorithms for resolving policy conflicts cannot be guaranteed.

Another critical problem for XACML policy analysis is *redundancy* discovery and removal. A rule in an XACML policy is redundant if every access request that matches the rule also matches other rules with the same effect. As the response time of an access request largely depends on the number of rules to be parsed within a policy, redundancies in a policy may adversely affect the performance of policy evaluation. Therefore, policy redundancy is treated as policy *anomaly* as well. Redundancy elimination can be regarded as one of effective solutions for optimiz-

- *H. Hu is with the Department of Computer and Information Sciences, Delaware State University, Dover, DE 19901, USA. E-mail: hhu@desu.edu.*
- *G-J. Ahn is with the Security Engineering for Future Computing (SEFCOM) Laboratory, and the Ira A. Fulton School of Engineering, Arizona State University, Tempe, AZ 85287, USA. E-mail: gahn@asu.edu.*
- *K. Kulkarni is with the NVIDIA, Sunnyvale, CA 94086, USA. E-mail: ketankulkarni29@gmail.com.*

ing XACML policies and improving the performance of XACML evaluation.

Policy anomaly detection has recently received a great deal of attention [3], [13], [23], [29], especially, in firewall policy analysis. Corresponding policy analysis tools, such as `Firewall Policy Advisor` [3], `FIREMAN` [29] and `FAME` [13], with the goal of discovering firewall policy anomalies have been developed. However, we cannot directly adopt those prior anomaly analysis approaches for XACML due to several reasons. First, most prior approaches mainly have the capability to detect *pairwise* policy anomalies, while a complete anomaly detection should consider all policy components as a whole piece. In other words, prior policy anomaly analysis approaches are still needed to be improved [4]. Second, the structure of firewall policies is flat but XACML has a hierarchical structure supporting recursive policy specification. Third, a firewall policy only supports one conflict resolution strategy (*first-match*) to resolve conflicts but XACML has four rule/policy combining algorithms. Additionally, a firewall rule is typically specified with fixed fields, while an XACML rule can be multi-valued. Therefore, a new policy analysis mechanism is desirable to cater the requirements from anomaly analysis in XACML policies.

In this paper, we introduce a policy-based segmentation technique, which adopts a binary decision diagram (BDD)-based data structure to perform set operations, for policy anomaly discovery and resolution. Based on this technique, an *authorization space* defined by an XACML policy or policy set component can be divided into a set of disjoint segments. Each segment associated with a unique set of XACML components indicates an overlapping relation (either *conflicting* or *redundant*) among those components. Accurate anomaly information is crucial to the success of anomaly resolution. For example, conflict diagnosis information provided by a policy analysis tool can be utilized to guide the policy designers in selecting appropriate combining algorithms. Moreover, we present a grid-based representation technique to show policy anomaly diagnosis information in an intuitive manner, facilitating more efficient policy anomaly management. Besides, we observe that current XACML conflict resolution mechanisms are too restrictive by applying only one combining algorithm to resolve all identified conflicts within an XACML policy or policy set component. Also, many other desirable conflict resolution strategies exist [14], [16], [17], but cannot be directly supported by XACML. Thus, we additionally provide a flexible and extensible policy conflict resolution method in this paper. In addition, based on our policy-based segmentation technique, we provide an effective redundancy discovery mechanism, where both rule redundancies within one policy and redundancies across multiple policies or policy sets can be detected and eliminated. Furthermore, we implement a policy analysis tool `XAnalyzer` based on our approach along with extensive experiments.

The rest of this paper is organized as follows. Section 2 briefly discusses anomalies in XACML policies. We describe the underlying data structure for XACML representation based on binary decision diagrams in Section 3. Section 4 presents our conflict detection and resolution approaches. In Section 5, we address our redundancy discovery and removal approaches. In Section 6, we discuss the implementation of our tool `XAnalyzer` and the evaluation of our approach. Section 7 overviews the related work and we conclude this paper in Section 8.

2 BACKGROUND

2.1 Overview of XACML

XACML has become the *de facto* standard for describing access control policies and offers a large set of built-in functions, data types, combining algorithms, and standard profiles for defining application-specific features. At the root of all XACML policies is a *policy* or a *policy set*. A *policy set* is composed of a sequence of *policies* or other *policy sets* along with a *policy combining algorithm* and a *target*. A *policy* represents a single access control policy expressed through a *target*, a set of *rules* and a *rule combining algorithm*. The *target* defines a set of subjects, resources and actions the policy or policy set applies to. A *rule set* is a sequence of rules. Each *rule* consists of a *target*, a *condition*, and an *effect*. The *target* of a rule determines whether an access request is applicable to the rule and it has a similar structure as the target of a policy or a policy set.

An XACML policy often has conflicting rules or policies, which are resolved by four different *combining algorithms*: *Deny-Overrides*, *Permit-Overrides*, *First-Applicable* and *Only-One-Applicable* [25]. Figure 1 shows an example XACML policy. The root policy set PS_1 contains two policies, P_1 and P_2, which are combined using *First-Applicable* combining algorithm. The policy P_1 has three rules, r_1, r_2 and r_3, and its rule combining algorithm is *Deny-Overrides*. The policy P_2 includes two rules r_4 and r_5 with *Deny-Overrides* combining algorithm. In this example, there are four subjects: *Manager*, *Designer*, *Developer* and *Tester*; two resources: *Reports* and *Codes*; and two actions: *Read* and *Change*. Note that both r_2 and r_3 define conditions over the *Time* attribute.

2.2 Anomalies in XACML Policies

An XACML policy may contain both policy components and policy set components. Often, a rule anomaly occurs in a policy component, which consists of a sequence of rules. On the other hand, a policy set component consists of a set of policies or other policy sets, thus anomalies may also arise among policies or policy sets. We address XACML policy anomalies at both policy level and policy set level.

- **Anomalies at Policy Level:** A rule is ***conflicting*** with other rules, if this rule overlaps with others but defines a different effect. For example, the *deny* rule r_1 is in conflict with the *permit* rule r_2 in Figure 1 because rule r_2 allows the access requests from a designer to change codes in the time interval [8:00, 17:00], which are supposed to be denied by r_1; and a rule is

```
<PolicySet PolicySetId="PS1" PolicyCombiningAlgId="First-Applicable">
 <Target/>
 <Policy PolicyId="P1" RuleCombiningAlgId="Deny-Overrides">
  <Target/>
  <Rule RuleId="r1" Effect="Deny">
   <Target>
     <Subjects><Subject>      Designer   </Subject>
               <Subject>      Tester     </Subject></Subjects>
     <Resources><Resource> Codes       </Resource></Resources>
     <Actions><Action>         Change     </Action></Actions>
   </Target>
  </Rule>
  <Rule RuleId="r2" Effect="Permit">
   <Target>
     <Subjects><Subject>      Designer   </Subject>
               <Subject>      Developer </Subject></Subjects>
     <Resources><Resource> Reports     </Resource>
                <Resource> Codes       </Resource></Resources>
     <Actions><Action>         Read       </Action>
              <Action>         Change     </Action></Actions>
   </Target>
   <Condition>     8:00 ≤ Time ≤ 17:00        </Condition>
  </Rule>
  <Rule RuleId="r3" Effect="Deny">
   <Target>
     <Subjects><Subject>      Designer   </Subject></Subjects>
     <Resources><Resource> Reports     </Resource>
                <Resource> Codes       </Resource></Resources>
    <Actions><Action>         Change     </Action></Actions>
   </Target>
   <Condition>     12:00 ≤ Time ≤ 13:00       </Condition>
  </Rule>
 </Policy>
 <Policy PolicyId="P2" RuleCombiningAlgId="Permit-Overrides">
  <Target/>
  <Rule RuleId="r4" Effect="Deny">
   <Target>
     <Subjects><Subject>      Developer </Subject></Subjects>
     <Resources><Resource> Reports   </Resource></Resources>
     <Actions><Action>         Change     </Action></Actions>
   </Target>
  </Rule>
  <Rule RuleId="r5" Effect="Permit">
   <Target>
     <Subjects><Subject>      Manager    </Subject>
               <Subject>      Designer   </Subject></Subjects>
     <Resources><Resource> Reports     </Resource>
                <Resource> Codes       </Resource></Resources>
     <Actions><Action>         Change     </Action></Actions>
   </Target>
  </Rule>
 </Policy>
</PolicySet>
```

Fig. 1. An example XACML policy.

redundant if there is other same or more general rules available that have the same effect. For instance, if we change the effect of r_2 to *Deny*, r_3 becomes redundant since r_2 will also deny a designer to change reports or codes in the time interval [12:00, 13:00].

- **Anomalies at Policy Set Level:** Anomalies may also occur across policies or policy sets in an XACML policy. For example, considering two policy components P_1 and P_2 of the policy set PS_1 in Figure 1, P_1 is ***conflicting*** with P_2, because P_1 permits the access requests that a developer changes reports in the time interval [8:00, 17:00], but which are denied by P_2. On the other hand, P_1 denies the requests allowing a designer to change reports or codes in the time interval [12:00, 13:00], which are permitted by P_2. Supposing the effect of r_2 is changed to *Deny* and the condition of r_2 is removed, r_4 is turned to be ***redundant*** with respect to r_2, even though r_2 and r_4 are placed in different policies P_1 and P_2, respectively.

A policy anomaly may involve in multiple rules. For example, in Figure 1, access requests that a designer changes codes in the time interval [12:00, 13:00] are permitted by r_2, but denied by both r_1 and r_3. Thus, this conflict associates with *three* rules. For another example, suppose the effect of r_3 is changed to *Permit* and the subject of r_3 is replaced by *Manager* and *Developer*. If we only examine *pairwise* redundancies, r_3 is not a redundant rule. However, if we check multiple rules simultaneously, we can identify r_3 is redundant considering r_2 and r_5 together. We observe that precise anomaly diagnosis information is crucial for achieving an effective anomaly resolution. In this paper, we attempt to design a systematic approach and corresponding tool not only for accurate anomaly detection but also for effective anomaly resolution.

3 Underlying Data Structure

Our policy-based segmentation technique introduced in subsequent sections requires a well-formed representation of policies for performing a variety of set operations. Binary Decision Diagram (BDD) [9] is a data structure that has been widely used for formal verification and simplification of digital circuits. In this work, we leverage BDD as the underlying data structure to represent XACML policies and facilitate effective policy analysis.

Given an XACML policy, it can be parsed to identify subject, action, resource and condition attributes. Once these attributes are identified, all XACML rules can be transformed into Boolean expressions [5]. Each Boolean expression of a rule is composed of atomic Boolean expressions combined by logical operators $\vee$ and $\wedge$. Atomic Boolean expressions are treated as equality constraints or range constraints on attributes (e.g. $Subject = \text{“}Designer\text{”}$) or on conditions (e.g. $8:00 \leq Time \leq 17:00$).

Example 1: Consider the example XACML policy in Figure 1 in terms of *atomic Boolean expressions*. The Boolean expression for rule r_1 is:

$$(Subject = \text{“}Designer\text{”} \vee Subject = \text{“}Tester\text{”}) \wedge (Resource = \text{“}Codes\text{”}) \wedge (Action = \text{“}Change\text{”})$$

The Boolean expression for rule r_2 is:

$$(Subject = \text{“}Designer\text{”} \vee Subject = \text{“}Developer\text{”}) \wedge (Resource = \text{“}Reports\text{”} \vee Resource = \text{“}Codes\text{”}) \wedge (Action = \text{“}Read\text{”} \vee Action = \text{“}Change\text{”}) \wedge (8:00 \leq Time \leq 17:00)$$

Boolean expressions for XACML rules may consist of atomic Boolean expressions with overlapping value ranges. In such cases, those atomic Boolean expressions are needed to be transformed into a sequence of new atomic Boolean expressions with disjoint value ranges. Agrawal et al. [1] have identified different categories of such atomic Boolean expressions and addressed corresponding solutions for those issues. We adopt similar approach to construct our Boolean expressions for XACML rules.

We encode each of the atomic Boolean expression as a Boolean variable. For example, an atomic Boolean expression *Subject=“Designer”* is encoded into a Boolean variable S_1. A complete list of Boolean encoding for the example XACML policy in Figure 1 is shown in Table 1. We then utilize the Boolean encoding to construct Boolean expressions in terms of Boolean variables for XACML rules.

TABLE 1
Atomic Boolean expressions and corresponding Boolean variables for P_1.

Unique Atomic Boolean Expression	Boolean Variable
$Subject =$ "$Designer$"	S_1
$Subject =$ "$Tester$"	S_2
$Subject =$ "$Developer$"	S_3
$Subject =$ "$Manager$"	S_4
$Resource =$ "$Reports$"	R_1
$Resource =$ "$Codes$"	R_2
$Action =$ "$Read$"	A_1
$Action =$ "$Change$"	A_2
$8:00 \leq Time < 12:00$	C_1
$12:00 \leq Time < 13:00$	C_2
$13:00 \leq Time \leq 17:00$	C_3

Example 2: Consider the example XACML policy in Figure 1 in terms of *Boolean variables*. The Boolean expression for rule r_1 is:

$(S_1 \vee S_2) \wedge (R_2) \wedge (A_2)$

The Boolean expression for rule r_2 is:

$(S_1 \vee S_3) \wedge (R_1 \vee R_2) \wedge (A_1 \vee A_2) \wedge (C_1 \vee C_2 \vee C_3)$

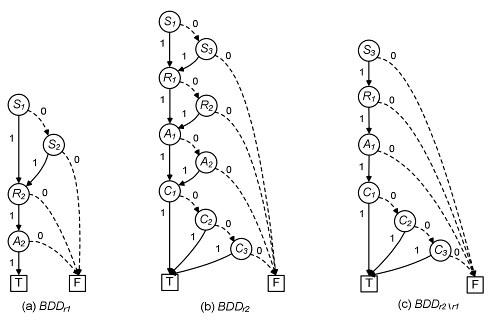


Fig. 2. Representing and operating on rules of XACML policy with BDD.

BDDs are acyclic directed graphs which represent Boolean expressions compactly. Each nonterminal node in a BDD represents a Boolean variable, and has two edges with binary labels, 0 and 1 for *nonexistent* and *existent*, respectively. Terminal nodes represent Boolean value `T` (True) or `F` (False). Figures 2(a) and 2(b) give BDD representations of two rules r_1 and r_2, respectively.

Once the BDDs are constructed for XACML rules, performing set operations, such as unions ($\cup$), intersections ($\cap$) and set differences ($\setminus$), required by our policy-based segmentation algorithms (see Algorithm 1 and Algorithm 2) is efficient as well as straightforward. Figure 2(c) shows an integrated BDD, which is the difference of r_2' BDD from r_1' BDD ($r_2 \setminus r_1$). Note that the resulting BDDs from the set operations may have *less* number of nodes due to the canonical representation of BDD.

4 Conflict Detection and Resolution

We first introduce a concept of *authorization space*, which adopts aforementioned BDD-based policy representation to perform policy anomaly analysis. This concept is defined as follows:

Definition 1: (**Authorization Space**). Let R_x, P_x and PS_x be the set of rules, policies and policy sets, respectively, of an XACML policy x. An authorization space for an XACML policy component $c \in R_x \cup P_x \cup PS_x$ represents a collection of all access requests [1] Q_c to which a policy component c is applicable.

4.1 Conflict Detection Approach

Our conflict detection mechanism examines conflicts at both policy level and policy set level for XACML policies. In order to precisely identify policy conflicts and facilitate an effective conflict resolution, we present a policy-based segmentation technique to partition the entire authorization space of a policy into disjoint authorization space segments. Then, conflicting authorization space segments (called *conflicting segment* in the rest of this paper), which contain policy components with different effects, are identified. Each conflicting segment indicates a policy conflict.

4.1.1 Conflict Detection at Policy Level

A policy component in an XACML policy includes a set of rules. Each rule defines an authorization space with the effect of either `permit` or `deny`. We call an authorization space with the effect of `permit` *permitted space* and an authorization space with the effect of `deny` *denied space*.

Algorithm 1 shows the pseudocode of generating conflicting segments for a policy component P. An entire authorization space derived from a policy component is first partitioned into a set of disjoint segments. As shown in lines 17-33 in Algorithm 1, a function called `Partition()` accomplishes this procedure. This function works by adding an authorization space s derived from a rule r to an authorization space set S. A pair of authorization spaces must satisfy one of the following relations: *subset* (line 19), *superset* (line 24), *partial match* (line 27), or *disjoint* (line 32). Therefore, one can utilize set operations to separate the overlapped spaces into disjoint spaces.

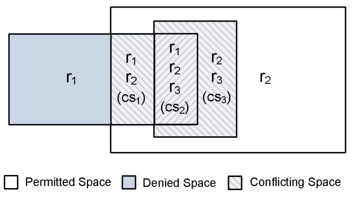
Conflicting segments are identified as shown in lines 6-10 in Algorithm 1. A set of conflicting segments $CS : \{cs_1, cs_2, \ldots, cs_n\}$ from conflicting rules has the following three properties:

1) All conflicting segments are pairwise disjoint: $cs_i \cap cs_j = \emptyset, 1 \leq i \neq j \leq n$;
2) Any two different requests q and q' within a single conflicting segment (cs_i) are matched by exact same set of rules: $GetRule(q) = GetRule(q')$ [2], $\forall q \in cs_i, q' \in cs_i, q \neq q'$; and
3) The effects of matched rules in any conflicting segments contain both "`Permit`" and "`Deny`."

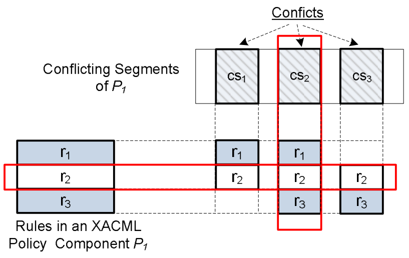
To facilitate the correct interpretation of analysis results, a concise and intuitive representation method is necessary. For the purposes of brevity and understandability, we first employ a two dimensional geometric representation for each authorization space segment. Note that a rule in an XACML policy typically has multiple fields, thus a complete representation of authorization space should be

1. We only consider single-valued requests in this definition. Multi-valued requests are discussed in Section 5.

2. $GetRule()$ is a function that returns all rules matching a request.



(a) Disjoint segments of authorization space for policy P_1


Conficts
Conflicting Segments of P_1
cs1
cs2
cs3
r1
r2
r3
Rules in an XACML Policy Component P_1

(b) Grid representation of policy conflict in policy P_1

Fig. 3. Authorization space representation for policy P_1 in the example XACML policy.

Algorithm 1: Identify Disjoint Conflicting Authorization Spaces of Policy P

```
Input: A policy $P$ with a set of rules.
Output: A set of disjoint conflicting authorization spaces $CS$ for $P$.
1  /* Partition the entire authorization space of $P$ into disjoint spaces*/
2  $S.New()$;
3  $S \longleftarrow$ Partition_P($P$);
4  /* Identify the conflicting segments */
5  $CS.New()$;
6  foreach $s \in S$ do
7      /* Get all rules associated with a segment $s$ */
8      $R' \longleftarrow GetRule(s)$;
9      if $\exists r_i \in R', r_j \in R', r_i \neq r_j$ and $r_i.Effect \neq r_j.Effect$ then
10         $CS.Append(s)$;

11 Partition_P($P$)
12 $R \longleftarrow GetRule(P)$;
13 foreach $r \in R$ do
14     $s_r \longleftarrow AuthorizationSpace(r)$;
15     $S \longleftarrow$ Partition($S, s_r$);
16 return $S$;
17 Partition($S, s_r$)
18 foreach $s \in S$ do
19     /* $s_r$ is a subset of $s$*/
20     if $s_r \subset s$ then
21         $S.Append(s \setminus s_r)$;
22         $s \longleftarrow s_r$;
23         $Break$;
24     /* $s_r$ is a superset of $s$*/
25     else if $s_r \supset s$ then
26         $s_r \longleftarrow s_r \setminus s$;
27     /* $s_r$ partially matches $s$*/
28     else if $s_r \cap s \neq \emptyset$ then
29         $S.Append(s \setminus s_r)$;
30         $s \longleftarrow s_r \cap s$;
31         $s_r \longleftarrow s_r \setminus s$;
32 $S.Append(s_r)$;
33 return $S$;
```

multi-dimensional. Also, we utilize colored rectangles to denote two kinds of authorization spaces: *permitted space* (white color) and *denied space* (grey color), respectively. Figure 3(a) gives a representation of the segments of authorization space derived from the policy P_1 in the XACML example policy shown in Figure 1. We can notice that five unique disjoint segments are generated. In particular, three conflicting segments cs_1, cs_2 and cs_3 are identified, representing three policy conflicts.

When a set of XACML rules interacts, one overlapping relation may be associated with several rules. Meanwhile, one rule may overlap with multiple other rules and can be involved in a couple of overlapping relations (overlapping segments). Different kinds of segments and associated rules can be viewed like Figure 3(a). However, it is still difficult for a policy designer or administrator to figure out how many segments one rule is involved in. To address the need of a more precise conflict representation, we additionally introduce a grid representation that is a *matrix-based* visualization of policy conflicts, in which space segments are displayed along the horizontal axis of the matrix, rules are shown along the vertical axis, and the intersection of a segment and a rule is a grid that displays a rule's subspace covered by the segment.

Figure 3(b) shows a grid representation of conflicts in the policy P_1 in our example policy. We can easily determine which rules are covered by a segment, and which segments are associated with a rule. For example, as shown in Figure 3(b), we can notice that a conflicting segment cs_2, which points out a conflict, is related to a rule set consisting of three rules r_1, r_2 and r_3 (highlighted with a horizontal red rectangle), and a rule r_2 is involved in three conflicting segments cs_1, cs_2 and cs_3 (highlighted with a vertical red rectangle). Our grid representation provides a better understanding of policy conflicts to policy designers and administrators with an overall view of related segments and rules.

4.1.2 Conflict Detection at Policy Set Level

There are two major challenges that need to be taken into consideration when we design an approach for XACML analysis at policy set level.

1) XACML supports four rule/policy combining algorithms: *First-Applicable*, *Only-One-Applicable*, *Deny-Overrides*, and *Permit-Overrides*.
2) An XACML policy is specified recursively and therefore has a hierarchical structure. In XACML, a policy set contains a sequence of policies or policy sets, which may further contain other policies or policy sets.

Each authorization space segment also has an *effect*, which is determined by the XACML components covered by this segment. For nonconflicting segments, the effect of a segment equals to the effect of components covered by this segment. Regarding conflicting segments, the effect of a segment depends on the following four cases of combining algorithm ($\mathcal{CA}$), which is used by the owner (a policy or a policy set) of the segment.

1) $\mathcal{CA}$=*First-Applicable*: In this case, the effect of a conflicting segment equals to the effect of the first component covered by the conflicting segment.

2) $\mathcal{CA}$=*Permit-Overrides*: The effect of a conflicting segment is always assigned with "`Permit`," since there is at least one component with "`Permit`" effect within this conflicting segment.
3) $\mathcal{CA}$=*Deny-Overrides*: The effect of a conflicting segment always equals to "`Deny`."
4) $\mathcal{CA}$=*Only-One-Applicable*: The effect of a conflicting segment equals to the effect of only-applicable component.

Algorithm 2: Identify Disjoint Conflicting Authorization Spaces of Policy Set PS

```
Input: A policy set $PS$ with a set of policies or other policy sets.
Output: A set of disjoint conflicting authorization spaces $CS$ for $PS$.
1  /* Partition the entire authorization space of $PS$ into disjoint spaces*/
2  $S.New()$;
3  $S \longleftarrow$ Partition_PS($PS$);
4  /* Identify the conflicting segments */
5  $CS.New()$;
6  foreach $s \in S$ do
7      $E \longleftarrow GetElement(s)$;
8      if $\exists e_i \in E, e_j \in E, e_i \neq e_j$ and $e_i.Effect \neq e_j.Effect$ then
9          $CS.Append(s)$;

10 Partition_PS($PS$)
11 $S^{''}.New()$;
12 $C \longleftarrow GetChild(PS)$;
13 foreach $c \in C$ do
14     $S^{'}.New()$;
15     /* c is a policy*/
16     if $IsPolicy(c) = true$ then
17         $S^{'} \longleftarrow$ Partition_P($c$);
18     /* c is a policy set*/
19     else if $IsPolicySet(c) = true$ then
20         $S^{'} \longleftarrow$ Partition_PS($c$)
21     $E^P.New()$;
22     $E^D.New()$;
23     foreach $s^{'} \in S^{'}$ do
24         if Effect($s^{'}$) = Permit then
25             $E^P \longleftarrow E^P \cup s^{'}$;
26         else if Effect($s^{'}$) = Deny then
27             $E^D \longleftarrow E^D \cup s^{'}$;
28     $S^{''} \longleftarrow$ Partition($S^{''}, E^P$);
29     $S^{''} \longleftarrow$ Partition($S^{''}, E^D$);
30 return $S^{''}$;
```

To support the recursive specifications of XACML policies, we parse and model an XACML policy as a tree structure [20], [21]. Algorithm 2 shows the pseudocode of identifying disjoint conflicting authorization spaces for a policy set PS. In order to partition authorization spaces of all nodes contained in a policy set tree, this algorithm recursively calls the partition functions, `Partition_P()` and `Partition_PS()`, to deal with the policy nodes (lines 16-17) and the policy set nodes (lines 19-20), respectively. Once all children nodes of a policy set are partitioned, we can then represent the authorization space of each child node (E) with two subspaces *permitted subspace* (E^P) and *denied subspace* (E^D) by aggregating all "`Permit`" segments and "`Deny`" segments, respectively, as follows:

$$\begin{cases} E^P = \bigcup_{s_i \in S_E} s_i & \text{if } Effect(s_i) = Permit \\ E^D = \bigcup_{s_i \in S_E} s_i & \text{if } Effect(s_i) = Deny \end{cases} \quad (1)$$

where S_E denotes the set of authorization space segments of the child node E.

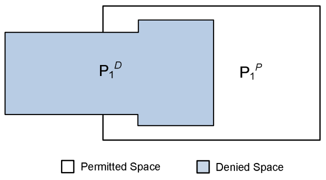


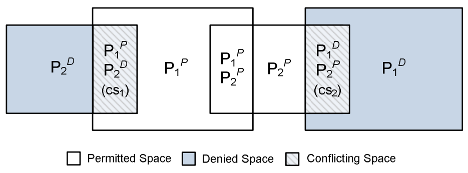
Fig. 4. Aggregation of authorization spaces for policy P_1 in the example XACML policy.

For example, since the combining algorithm of the policy P_1 in our example XACML policy is *Deny-Overrides*, the effects of three conflicting segments shown in Figure 3 are "`Deny`". Figure 4 shows the result of aggregating authorization spaces of the policy P_1, where two subspaces P_1^P and P_1^D are constructed.

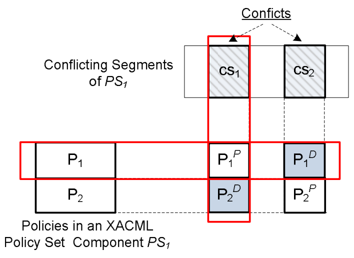
In order to generate segments for the policy set PS, we can then leverage two subspaces (E^P and E^D) of each child node (E) to partition existing authorization space set belonging to PS (lines 28-29). Figure 5(a) represents an example of the segments of authorization space derived from policy set PS_1 in our example policy (Figure 1). We can observe that seven unique disjoint segments are generated, and two of them cs_1 and cs_2 are conflicting segments. We additionally give a grid representation of conflicts in the policy set PS_1 shown in Figure 5(b). Then, we can easily identify that the conflicting segment cs_1 is related to two subspaces: P_1's *permitted* subspace P_1^P and P_2's *denied* subspace P_2^D, and the policy P_1 is associated with two conflicts, where P_1's *permitted* subspace P_1^P is involved in the conflict represented by cs_1 and P_1's *denied* subspace P_1^D is related to the conflict represented by cs_2.

4.2 Fine-Grained Conflict Resolution

Once conflicts within a policy component or policy set component are identified, a policy designer can choose appropriate conflict resolution strategies to resolve those identified conflicts. However, current XACML conflict resolution mechanisms have limitations in resolving conflicts effectively. First, existing conflict resolution mechanisms in XACML are too restrictive and only allow a policy designer to select one combining algorithm to resolve all identified conflicts within a policy or policy set component. A policy designer may want to adopt different combining algorithms to resolve different conflicts. Second, XACML offers four conflict resolution strategies. However, many conflict resolution strategies exist [14], [17], but cannot be specified in XACML. Thus, it is necessary to seek a comprehensive conflict resolution mechanism for more effective conflict resolution. Towards this end, we introduce a flexible and extensible conflict resolution framework to achieve a fine-grained conflict resolution as shown in Figure 6.



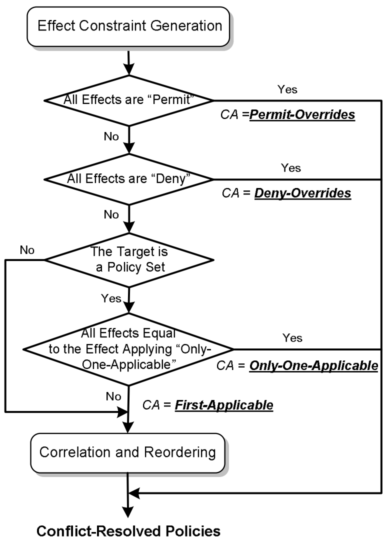
(a) Disjoint segments of authorization space for policy set PS_1



Conficts
Conflicting Segments of PS_1
cs_1
cs_2
P_1
P_2
P_1^P
P_1^D
P_2^D
P_2^P
Policies in an XACML Policy Set Component PS_1

(b) Grid representation of policy conflicts in policy set PS_1

Fig. 5. Authorization space representation for policy set PS_1 in the example XACML policy.



Effect Constraint Generation
All Effects are "Permit"
Yes
CA =Permit-Overrides
No
All Effects are "Deny"
Yes
CA = Deny-Overrides
No
No
The Target is a Policy Set
Yes
All Effects Equal to the Effect Applying "Only-One-Applicable"
Yes
CA = Only-One-Applicable
No
CA = First-Applicable
Correlation and Reordering
Conflict-Resolved Policies

Fig. 6. Fine-grained conflict resolution framework.

4.2.1 Effect Constraint Generation from Conflict Resolution Strategy

Our conflict resolution framework introduces an *effect constraint* that is assigned to each conflicting segment. An effect constraint for a conflicting segment defines a desired response (either `permit` or `deny`) that an XACML policy should take when any access request matches the conflicting segment. The effect constraint is derived from the conflict resolution strategy applied to the conflicting segment, using a similar process of determining the effect of a conflicting segment described in Section 4.1.2. A policy designer chooses an appropriate conflict resolution strategy for each identified conflict by examining the features of conflicting segment and associated conflicting components. In our conflict resolution framework, a policy designer is able to adopt different strategies to resolve conflicts indicated by different conflicting segments. In addition to four standard XACML conflict resolution strategies, user-defined strategies [17], such as *Recency-Overrides*, *Specificity-Overrides* and *High-Majority-Overrides*, can be implied in our framework as well. For example, applying a conflict resolution strategy, *High-Majority-Overrides*, to the second conflicting segment cs_2 of policy P_1 depicted in Figure 3, an effect constraint *Effect = "Deny"* will be generated for cs_2.

4.2.2 Conflict Resolution Based on Effect Constraints

A key feature of adopting *effect constraints* in our framework is that other conflict resolution strategies assigned to resolve different conflicts by a policy designer can be *automatically* mapped to standard XACML combining algorithms, without changing the way that current XACML implementations perform. As illustrated in Figure 6, an XACML combining algorithm can be derived for a target component by examining all effect constraints of the conflicting segments. If all effect constraints are "`Permit`," *Permit-Overrides* is selected for the target component to resolve all conflicts. In case that all effect constraints are "`Deny`," *Deny-Overrides* is assigned to the target component. Then, if the target component is a policy set and all effect constraints can be satisfied by applying *Only-One-Applicable* combining algorithm, *Only-One-Applicable* is selected as the combining algorithm of the target component. Otherwise, *First-Applicable* is selected as the combining algorithm of the target component. In order to resolve all conflicts within the target component by applying *First-Applicable*, the process of reordering conflicting components is compulsory to enable that the first-applicable component in each conflicting segment has the same effect with corresponding effect constraint.

Practically, one XACML component may get involved in multiple conflicts. In this case, removing such a component to satisfy one effect constraint may violate other effect constraints. Therefore, we cannot resolve a conflict individually by reordering a set of conflicting components associated with one conflict. On the other hand, it is also inefficient to deal with all conflicts together by reordering all conflicting components simultaneously. Thus, we next

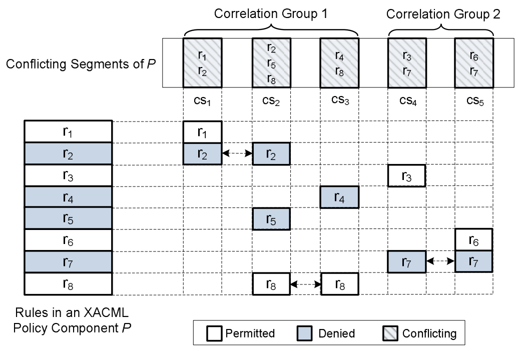


Fig. 7. Example of conflicting segment correlation.

introduce a correlation mechanism to identify dependent relationships among conflicting segments. The major benefit of identifying dependent relationships for a conflict resolution is to lessen the searching space of reordering conflicting components.

Figure 7 shows an example for conflicting segment correlation, considering an XACML policy component P with eight rules. Five conflicting segments are identified in this example. Several rules in this XACML policy component are involved in multiple conflicts. For example, r_2 contributes to two policy conflicts corresponding to two conflicting segments cs_1 and cs_2, respectively. Also, r_8 is associated with two conflicting segments cs_2 and cs_3. Suppose we want to satisfy the effect constraint of cs_2 by reordering associated conflicting rules, r_2, r_5 and r_8. The position change of r_2 and r_8 would affect conflicting segments, cs_1 and cs_2, respectively. Thus, a dependent relationship can be derived among cs_1, cs_2 and cs_3 with respect to the conflict resolution. Similarly, we can identify the dependent relation between cs_4 and cs_5. We organize those conflicting segments with a dependent relationship as a group called *conflict correlation group*. The pseudocode of an algorithm for identify conflict correlation groups is given in Algorithm 3.

Algorithm 3: Conflicting Segment Correlation

```
Input: A set of conflicting segments, $C$.
Output: A set of groups for correlated segment, $G$.
$G.New()$;
$g \longleftarrow G.NewGroup()$;
foreach $c \in C$ do
    $R \longleftarrow GetRule(c)$;
    foreach $g \in G$ do
        foreach $c^{'} \in GetSegment(g)$ do
            $R^{'}.Append(GetRule(c^{'}))$;
        if $R \cap R^{'} \neq \varnothing$ then
            $g.Append(c)$;
        else
            $G.NewGroup().Append(c)$;
return $G$;
```

Algorithm 4: Checking if a Rule r is Irremovable in a Policy P: IrremovableCheck(r, P)

```
Input: A rule $r$ and the policy $P$ contains $r$.
Output: True or false.
$Flag \longleftarrow$ F ;
/* Case 1: rule combining algorithm is First-Applicable */
if $\mathcal{P}.\mathcal{CA} =$ First-Applicable then
    foreach $r' \in GetSubsequentRule(r, P)$ do
        if $r'.Effect \neq r.Effect$ then
            $Flag \longleftarrow$ T ;
/* Case 2: rule combining algorithm is Permit-Override */
if $\mathcal{P}.\mathcal{CA} =$ Permit-Override then
    if $r.Effect = Permit$ then
        foreach $r' \in GetOtherRule(r, P)$ do
            if $r'.Effect = Deny$ then
                $Flag \longleftarrow$ T ;
/* Case 3: rule combining algorithm is Deny-Override */
if $\mathcal{P}.\mathcal{CA} =$ Deny-Override then
    if $r.Effect = Deny$ then
        foreach $r' \in GetOtherRule(r, P)$ do
            if $r'.Effect = Permit$ then
                $Flag \longleftarrow$ T ;
if $Flag = T$ then
    return $true$;
else
    return $false$;
```

5 Redundancy Discovery and Removal

Our redundancy discovery and removal mechanism also leverage the policy-based segmentation technique to explore redundancies at both policy level and policy set level.

5.1 Redundancy Elimination at Policy Level

We employ following steps to identify and eliminate redundant rules at policy level.

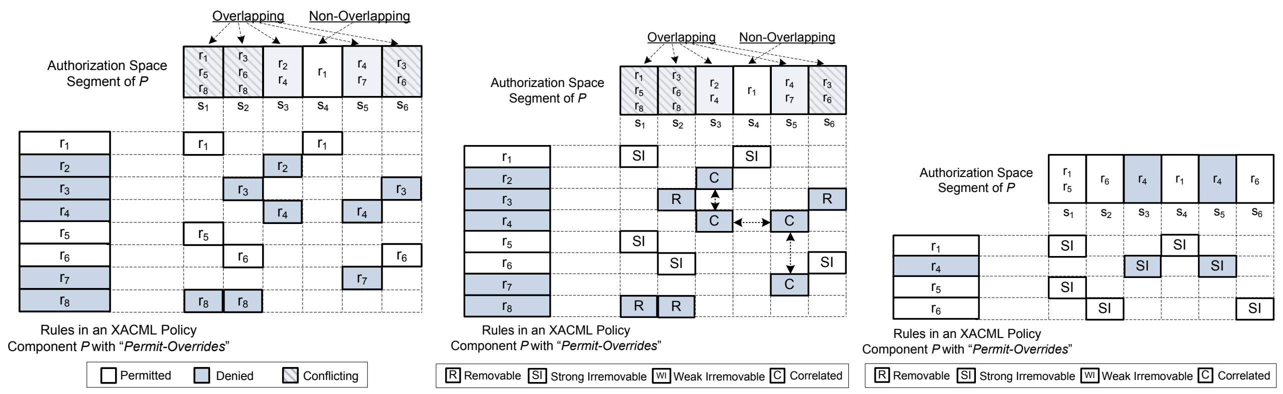
5.1.1 Authorization Space Segmentation

We first perform the policy segmentation function `Partition_P()` defined in Algorithm 1 to divide the entire authorization space of a policy into disjoint segments. We classify the policy segments in following categories: *non-overlapping* segment and *overlapping* segment, which is further divided into *conflicting overlapping* segment and *non-conflicting overlapping* segment. Each *non-overlapping* segment associates with one unique rule and each *overlapping* segment is related to a set of rules, which may conflict with each other (*conflicting overlapping* segment) or have the same effect (*non-conflicting overlapping* segment). Figure 8(a) illustrates a grid representation of authorization space segmentation for a policy with eight rules. In this example, one policy segment s_4 is a *non-overlapping* segment. Other policy segments are *overlapping* segments, including three *conflicting overlapping* segments s_1, s_2 and s_6, and two *non-conflicting overlapping* segments s_3 and s_5.

5.1.2 Irremovable Rule Identification Considering Multi-valued Requests

An XACML request may be multi-valued. For example, an XACML request can be "a person, who is both a Developer and a Designer, wants to change reports", where the subject has two values, *Developer* and *Designer*. A multi-valued request may match several rules, which do not overlap with each other in terms of single-valued requests. For instance, the above multi-valued request matches both r_4 and r_5 in the example policy shown in Figure 1, although r_4 and r_5 have no overlapping relation considering single-valued requests. We observe that an XACML rule may be *removable* with respect to single-valued requests but *irremovable* taking into account multi-valued requests. Therefore, we introduce a process to examine whether an XACML rule is *irremovable* considering multi-valued requests based on the three cases of rule combining algorithm ($\mathcal{CA}$).

1) $\mathcal{CA}$=*First-Applicable*: In this case, since a multi-valued request may match the examined rule and



(a) Authorization space segmentation (b) Property assignment (c) Redundancy removal

Fig. 8. Example of eliminating redundancies at policy level.

any subsequent rule(s) in the policy, if there is a subsequent rule with a different effect, the examined rule is considered *irremovable*.

2) $\mathcal{CA}$=*Permit-Overrides*: A multi-valued request may match the examined rule and any other rule(s) in the policy. If the examined rule is a "`permit`" rule and there is any other rule being a "`deny`" rule, the examined rule is *irremovable*.
3) $\mathcal{CA}$=*Deny-Overrides*: If the examined rule is a "`deny`" rule and there is any other rule being a "`permit`" rule, the examined rule is *irremovable*.

Algorithm 4 shows the pseudocode for the definition of a function `IrremovableCheck()`, which will subsequently be used in both property assignment and redundancy removal processes to check if a rule is *irremovable* in a policy considering multi-valued requests.

5.1.3 Property Assignment for Rule Subspaces

In this step, every rule subspace covered by a policy segment is assigned with a property. Four property values, *removable* (R), *strong irremovable* (SI), *weak irremovable* (WI) and *correlated* (C), are defined to reflect different characteristics of rule subspace. *Removable* property is used to indicate that a rule subspace is removable. In other words, removing such a rule subspace does not make any impact on the original authorization space of an associated policy. *Strong irremovable* property means that a rule subspace cannot be removed because (1) this rule subspace belongs to an *irremovable* rule with respect to multi-valued requests; or (2) the effect of corresponding policy segment can be only decided by this rule. *Weak irremovable* property is assigned to a rule subspace when any subspace belonging to the same rule has *strong irremovable* property. That means a rule subspace becomes irremovable due to the reason that other portions of this rule cannot be removed. *Correlated* property is assigned to multiple rule subspaces covered by a policy segment, if the effect of this policy segment can be determined by any of these rules. We next introduce four processes to perform the property assignments to all of rule subspaces within the segments of a policy, considering *irremovable* rules and different categories of policy segments.

Process1: *Property assignment for the rule subspace belonging to an irremovable rule*. An irremovable rule can be identified by calling function `IrremovableCheck()`. All subspaces belonging to an irremovable rule are assigned with *strong irremovable* property.

Process2: *Property assignment for the rule subspace covered by a non-overlapping segment*. A non-overlapping segment contains only one rule subspace. Thus, this rule subspace is assigned with *strong irremovable* property. Other rule subspaces associated with the same rule are assigned with *weak irremovable* property, excepting the rule subspaces that already have *strong irremovable* property.

Process3: *Property assignment for rule subspaces covered by a conflicting segment*. We present this property assignment process based on the following three cases of rule combining algorithm.

1) $\mathcal{CA}$=*First-Applicable*: In this case, the first rule subspace covered by the conflicting segment is assigned with *strong irremovable* property. Other rule subspaces in the same segment are assigned with *removable* property. Meanwhile, other rule subspaces associated with the same rule are assigned with *weak irremovable* property except the rule subspaces already having *strong irremovable* property.
2) $\mathcal{CA}$=*Permit-Overrides*: All subspaces of "`deny`" rules in this conflicting segment are assigned with *removable* property. If there is only one "`permit`" rule subspace, this case is handled which is similar to the *First-Applicable* case. If any "`permit`" rule subspace has been assigned with *irremovable* property, other rule subspaces without *irremovable* property are assigned with *removable* property. Otherwise, all "`permit`" rule subspaces are assigned with *correlated*

property.

3) $\mathcal{CA}$=*Deny-Overrides*: This case is dealt with as the same as *Permit-Overrides* case.

Process4: *Property assignment for rule subspaces covered by a non-conflicting overlapping segment.* If any rule subspace has been assigned with *irremovable* property, other rule subspaces without *irremovable* property are assigned with *removable* property. Otherwise, all subspaces within the segment are assigned with *correlated* property.

Figure 8(b) shows the result of applying our property assignment mechanism to the example presented in Figure 8(a). We can easily identify that r_3 and r_8 are *removable* rules, where all subspaces are with *removable* property. However, we need to further examine the *correlated* rules r_2, r_4 and r_7, which contain subspaces with *correlated* property.

5.1.4 Rule Correlation Break and Redundancy Removal

Rules covered by an overlapping segment are correlated with each other when the effect of the overlapping segment can be determined by any of those rules. Thus, keeping one correlated rule and removing others do not change the effect of the overlapping segment. In addition, some rules may get involved in multiple correlated relations. For example, in Figure 8(b), r_4 has two subspaces that are involved in the correlated relations with r_2 and r_7, respectively. Therefore, similar to the construction of conflict correlation groups, we build rule correlation groups based on these two situations so that dependent relationships among multiple correlated rules within one group can be examined simultaneously. For example, a correlation group consisting of three rules r_2, r_4 and r_7 can be identified in Figure 8(b).

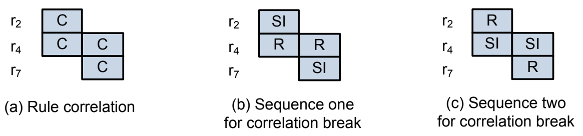

r2
r4
r7
C
C
C
C
(a) Rule correlation
SI
R
R
SI
(b) Sequence one for correlation break
R
SI
SI
R
(c) Sequence two for correlation break

Fig. 9. Example of rule correlation break.

The goal of rule correlation break is to discover as many redundant rules as possible. Different sequences to break rule correlations may lead to different results for redundancy removal. For example, Figure 9(a) shows correlated relations of rules r_2, r_4 and r_7, and we can break their correlated relations into different sequences. As shown in Figure 9(b), if we first choose r_2 as an *irremovable* rule and assign r_2' subspace *strong irremovable* property, only r_4 becomes a *removable* rule and r_7 is turned to be *irremovable*. However, as shown in Figure 9(c), if we first choose r_4 as an *irremovable* rule and assign two subspaces of r_4 with *strong irremovable* property, both r_2 and r_7 then become *removable* rules. To seek an optimal solution for rule correlation break, we measure a *breaking degree* for each correlated rule r, denoted as $BD(r)$, which indicates the number of removable rules if choosing r as

Algorithm 5: Redundancy Elimination of Policy P: RedundancyEliminate_P(P)

```
Input: A policy $P$ with a set of rules.
Output: A redundancy-eliminated policy $P'$.
/* Partition the entire authorization space of P into disjoint spaces*/
$S.New()$;
$S \longleftarrow$ Partition_P($P$);
/* Property assignment for all rule subspaces */
PropertyAssgin_P($S$);
/* Rule correlation break */
$G \longleftarrow$ CorrelatonGroupConstruct($S$);
foreach $g \in G$ do
    foreach $r \in g$ do
        $r.BD \longleftarrow \sum_{s_i \in CS(r)}(NC(s_i) - 1)$;
    $SP \longleftarrow GetCorrelatedSubspace(MaxBDRule(g))$
    foreach $sp \in SP$ do
        $sp.Property \longleftarrow$ R ;
        if $|GetCorrelatedSubspace(sp)| = 1$ then
            $SP' \longleftarrow GetCorrelatedSubspace(sp)$;
            $SP'.Property \longleftarrow$ SI ;
            AssginSI($SP'$);
/*Redundancy removal */
$P' \longleftarrow P$;
foreach $r \in P'$ do
    if $AllRemovalProperty(r) = true$ and
    $IrremovableCheck(r, P') = false$ then
        $P' \longleftarrow P' \setminus r$;
return $P'$;
```

an irremovable rule. $BD(r)$ can be calculated with the following equation:

$$BD(r) = \sum_{s_i \in CS(r)} (NC(s_i) - 1) \quad (2)$$

where, function $CS(r)$ returns the set of all overlapping segments covering correlated subspaces of the rule r, and function $NC(s_i)$ returns the number of correlated rules covered by the segment s_i. For example, $CS(r_4)$ returns a segment set $\{s_3, s_5\}$ and $NC(s_3)$ equals to 2. Since choosing r as an *irremovable* rule turns the subspaces of other rules covered by the segment s_i to *removable*, $NC(s_i) - 1$ donates the number of these removable rules. Consequently, $BD(r)$ aggregates the number of removable rules if setting r as *irremovable*. To maximize the number of removable rules for redundancy elimination, our correlation break process selects the rule with the maximum BD value as the candidate irremovable rule each time. For instance, applying this equation to compute breaking degrees of three rules demonstrated in Figure 9(a), both $BD(r_2)$ and $BD(r_7)$ are equal to 1, and $BD(r_4)$ is equal to 2. Thus, we choose r_4 as the candidate irremovable rule in the first step for rule correlation break. Finally, two rules r_2 and r_7 become removable after breaking all correlations.

The pseudocode of the algorithm for eliminating redundancy at policy level is shown in Algorithm 5. Note that a function `IrremovableCheck()` is called to check if a candidate rule is truly removable considering multi-valued requests before removing the rule from the policy. Figure 8(c) depicts the result of applying this algorithm to the example given in Figure 8(a). Four rules r_2, r_3, r_7 and r_8 were identified as redundant rules and removed from

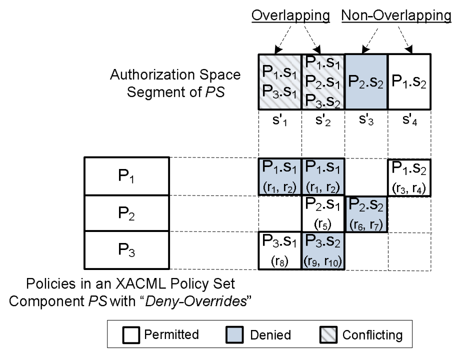


Fig. 10. Example of authorization space segmentation at policy set level for redundancy discovery and removal.

the policy. However, if we leverage *traditional* redundancy detection method [23], [3], which was limited to detect *pairwise* redundancies, to this example, only two redundant rules r_2 and r_7 can be discovered.

5.2 Redundancy Elimination at Policy Set Level

Similar to the solution of conflict detection at policy set level, we handle the redundancy removal for a policy set based on an XACML tree structure representation. If the children nodes of the policy set is a policy node in the tree, we perform `RedundancyEliminate_P()` function to eliminate redundancies. Otherwise, `RedundancyEliminate_PS()` function is excused recursively to eliminate redundancy in a policy set component.

After each component of a policy set PS performs redundancy removal, the authorization space of PS can be then partitioned into disjoint segments by performing `Partition()` function. Note that, in the solution for conflict detection at policy set level, we aggregate authorization subspaces of each child node before performing space partition, because we only need to identify conflicts among children nodes to guide the selection of policy combining algorithms for the policy set. However, for redundancy removal at policy set level, both redundancies among children nodes and rule (leaf node) redundancies, which may exist across multiple policies or policy sets, should be discovered. Therefore, we keep the original segments of each child node and leverage those segments to generate the authorization space segments of PS. Figure 10 demonstrates an example of authorization space segmentation of a policy set PS with three children components P_1, P_2 and P_3. The authorization space segments of PS are constructed based on the original segments of each child component. For instance, a segment $s_2^{'}$ of PS covers three policy segments $P_1.s_1$, $P_2.s_1$ and $P_3.s_2$, where $P_i.s_j$ denotes that a segment s_j belongs to a policy P_i.

The property assignment step at policy set level is similar to the property assignment step at policy level, except that the policy combining algorithm *Only-One-Applicable* needs to be taken into consideration at policy set level. The *Only-One-Applicable* case is handled similar to the *First-Applicable* case. We first check whether the combining algorithm is applicable or not. If the combining algorithm is applicable, the only-applicable subspace is assigned with *strong irremovable* property. Otherwise, all subspaces within the policy set's segment are assigned with *removable* property.

After assigning properties to all segments of children components of PS, we next examine whether any child component is redundant. If a child component is redundant, this child component and all rules contained in the child component are removed from PS. Then, we examine whether there exist any redundant rules. In this process, the properties of all rule subspaces covered by a *removable* segment of a child component of PS needs to be changed to *removable*. Note that when we change the property of a *strong irremovable* rule subspace to *removable*, other subspaces in the same rule with dependent *weak irremovable* property need to be changed to *removable* correspondingly.

6 Implementation and Evaluation

We have implemented a policy analysis tool called `XAnalyzer` in Java. Based on our policy anomaly analysis mechanism, it consists of four core components: segmentation module, effect constraint generation module, strategy mapping module, and property assignment module. The segmentation module takes XACML policies as an input and identifies the authorization space segments by partitioning the authorization space into disjoint subspaces. `XAnalyzer` utilizes APIs provided by *Sun* XACML implementation [27] to parse the XACML policies and construct Boolean encoding. JavaBDD [15], which is based on BuDDy package [10], is employed by `XAnalyzer` to support BDD representation and authorization space operations. The effect constraint generation module takes conflicting segments as an input and generates effect constraints for each conflicting segment. Effect constraints are generated based on strategies assigned to each conflicting segment. The strategy mapping module takes conflict correlation groups and effect constraints of conflicting segments as inputs and then maps assigned strategies to standard XACML combining algorithms for examined XACML policy components. The property assignment module automatically assigns corresponding property to each subspace covered by the segments of XACML policy components. The assigned properties are in turn utilized to identify redundancies.

We evaluated the efficiency and effectiveness of `XAnalyzer` for policy analysis on both real-life and synthetic XACML policies. Our experiments were performed on Intel Core 2 Duo CPU 3.00 GHz with 3.25 GB RAM running on Windows XP SP2. In our evaluation, we utilized five real-life XACML policies, which were collected from different sources. Three of the policies, *CodeA, Continue-a* and *Continue-b* are XACML policies used in [11]; among them, *Continue-a* and *Continue-b* are designed for a real-world Web application supporting a conference management. *GradeSheet* is utilized in [7]. The *Pluto* policy is employed in ARCHON system, [3] which is a digital library that federates the collections of physics with multiple degrees of meta data richness. In addition, we generated four large synthetic policies *SyntheticPolicy-1*, *SyntheticPolicy-2*, *SyntheticPolicy-3* and *SyntheticPolicy-4* for further evaluating the performance and scalability of our tool. These synthetic policies are multi-layered, where each policy component has a randomly selected combining algorithm and each rule has randomly chosen attribute sets from a predefined domain. We also use *SamplePolicy*, which is the example XACML policy represented in Figure 1, in our experiments. Table 2 summarizes the basic information of each policy including the number of rules, the number of policies, and the number of policy sets.

We conducted two separate sets of experiments for the evaluation of conflict detection approach and the evaluation of redundancy removal approach, respectively. Also, we performed evaluations at both policy level and policy set level.

TABLE 2
XACML policies used for evaluation.

Policy	Rule (#)	Policy (#)	Policy Set (#)
1 (CodeA)	4	2	5
2 (SamplePolicy)	6	2	1
3 (GradeSheet)	13	1	0
4 (Pluto)	22	1	0
5 (SyntheticPolicy-1)	147	30	11
6 (Continue-a)	312	276	111
7 (Continue-b)	336	305	111
8 (SyntheticPolicy-2)	456	65	40
9 (SyntheticPolicy-3)	572	114	75
10 (SyntheticPolicy-4)	685	188	84

Evaluation of Conflict Detection: Time required by `XAnalyzer` for conflict detection highly depends upon the number of segments generated for each XACML policy. The increase of the number of segments is proportional to the number of components contained in an XACML policy. From Table 3, we observed that `XAnalyzer` performs fast enough to handle larger size XACML policies, even for some complex policies with multiple levels of hierarchies along with hundreds of rules, such as two real-life XACML policies *Continue-a* and *Continue-b* and four synthetic XACML policies. The time trends observed from Table 3 are promising, and hence provide the evidence of efficiency of our conflict detection approach.

TABLE 3
Conflict detection algorithm evaluation.

Policy	Conflict Detection		
	Policy Level(#)	Policy Set Level(#)	Time (s)
1 (CodeA)	1	1	0.082
2 (SamplePolicy)	0	2	0.090
3 (GradeSheet)	0	4	0.098
4 (Pluto)	0	5	0.136
5 (SyntheticPolicy-1)	8	14	0.329
6 (Continue-a)	9	17	0.583
7 (Continue-b)	10	21	0.635
8 (SyntheticPolicy-2)	29	17	0.896
9 (SyntheticPolicy-3)	39	19	0.948
10 (SyntheticPolicy-4)	56	19	1.123

3. `http://archon.cs.odu.edu/`

Evaluation of Redundancy Removal: In the second set of experiments, we evaluated our redundancy analysis approach based on those experimental XACML policies in terms of two different cases: redundancy removal considering single-valued requests and redundancy removal considering multi-valued requests. From Table 4, we noticed that 140 rules were identified as redundant rules in the experimental XACML policies by our redundancy removal approach considering single-valued requests. By comparison, if multi-valued requests were taken into account in our redundancy removal algorithm, 21 rules became irremovable. Besides, the evaluation results shown in Table 4 indicate the efficiency of our redundancy analysis algorithm as well.

We also conducted the evaluation of effectiveness by comparing our redundancy analysis approach with *traditional* redundancy analysis approach [3], [23], which can only identify redundancy relations between *two* rules. Figure 11(a) depicts the results of our comparison experiments. From Figure 11(a), we observed that `XAnalyzer` could identify that an average of 5.6% of total rules are redundant. However, *traditional* redundancy analysis approach could only detect an average 3.1% of total rules as redundant rules. Therefore, the enhancement for redundancy elimination was clearly observed by our redundancy analysis approach compared to *traditional* redundancy analysis approach in our experiments.

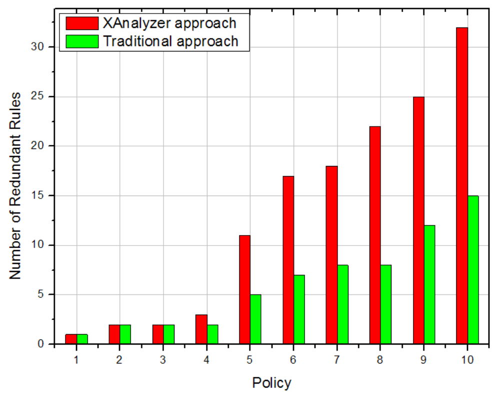
Furthermore, when redundancies in a policy are removed, the performance of policy enforcement is improved generally. For each of XACML policies in our experiments, Figure 11(b) depicts the total processing time in *Sun* XACML PDP [27] for responding 10,000 randomly generated XACML requests. The evaluation results clearly show that the processing times are reduced after eliminating redundancies in XACML policies applying either *traditional* approach or our approach, and our approach can obtain better performance improvement than *traditional* approach.

7 Related Work

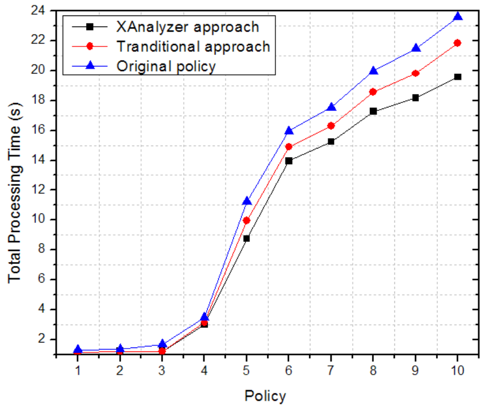
Many research efforts have been devoted to modeling and verification of XACML policies [2], [8], [11]. In [8], the authors formalized XACML policies using a process algebra known as Communicating Sequential Processes (CSP). This work utilizes a model checker to formally verify properties of policies, and to compare access control policies with each other. Fisler et al. [11] introduced an approach to represent XACML policies with Multi-Terminal Binary Decision Diagrams (MTBDDs). They developed a policy analysis tool called `Margrave`, which can verify XACML policies against the given properties and perform change-impact analysis. Ahn et al. [2] presented a formalization of XACML using answer set programming (ASP), which is a recent form of declarative programming, and leveraged existing ASP reasoners to conduct policy verification. However, lacking an exhaustive elicitation of properties, the completeness of the analysis results of policy

TABLE 4
Redundancy removal algorithm evaluation.

Policy	Redundant Removal Considering Single-valued Requests			Redundant Removal Considering Multi-valued Requests		
	Policy Level(#)	Policy Set Level(#)	Time (s)	Policy Level(#)	Policy Set Level(#)	Time (s)
1 (CodeA)	1	0	0.087	1	0	0.088
2 (SamplePolicy)	0	2	0.095	0	2	0.098
3 (GradeSheet)	0	2	0.113	0	2	0.120
4 (Pluto)	0	3	0.147	0	3	0.151
5 (SyntheticPolicy-1)	7	4	0.158	6	4	0.163
6 (Continue-a)	11	7	0.214	8	6	0.223
7 (Continue-b)	12	7	0.585	9	7	0.597
8 (SyntheticPolicy-2)	14	8	0.623	12	7	0.647
9 (SyntheticPolicy-3)	17	10	0.672	15	9	0.695
10 (SyntheticPolicy-4)	23	12	0.803	18	10	0.852



(a) Redundancy elimination rate



(b) Performance improvement

Fig. 11. Evaluation of redundancy removal approach.

verification cannot be guaranteed. In contrast, our approach for policy anomaly analysis can indicate accurate anomaly information without the need of any external properties.

Several work presenting policy analysis tools with the goal of detecting policy anomalies in firewall are closely related to our work. Al-Shaer et al. [3] designed a tool called `Firewall Policy Advisor` which can only detect *pairwise* anomalies in firewall rules. Yuan et al. [29] presented a toolkit, `FIREMAN`, which can detect anomalies among *multiple* firewall rules by analyzing the relationships between *one* rule and the collections of packet spaces derived from all preceding rules. Liu et al. [22] introduced a method for complete redundancy detection in firewall rules using a tree representation of firewalls, call Firewall Decision Trees (FDTs). However, as we discussed previously, XACML policy and firewall policy have some significant distinctions. Hence, directly applying prior firewall policy anomaly analysis approaches to XACML is not suitable.

Some XACML policy evaluation engines, such as *Sun* PDP [27] and `XEngine` [20], [21], have been developed to handle the process of evaluating whether a request satisfies an XACML policy. During the process of policy enforcement, conflicts can be checked if a request matches multiple rules having different effects, and then conflicts are resolved by applying predefined combining algorithms in the policy. In contrast, our tool `XAnalyzer` focuses on policy analysis at policy *design* time. `XAnalyzer` can identify all conflicts within a policy and help policy designers select appropriate combining algorithms for conflict resolution prior to the policy enforcement. Additionally, `XAnalyzer` has the capability of discovering and eliminating policy redundancies that cannot be dealt with by policy evaluation engines.

Some work addressed the general conflict resolution mechanisms for access control [11], [14], [16], [17]. Especially, Li et al. [17] proposed a policy combining language `PCL`, which can be utilized to specify a variety of user-defined combining algorithms for XACML. These conflict resolution mechanisms can be accommodated in our fine-grained conflict resolution framework. In addition, Bauer et al. [6] adopted a data-mining technique to eliminate *inconsistencies* occurring between access control policies and user's intentions. By comparison, our approach detects and resolves anomalies within access control policies caused by overlapping relations.

Other related work includes XACML policy integration [24], [26] and XACML policy similarity analysis [19]. In particular, Lin et al. [18] designed a comprehensive environment called `EXAM` for XACML policy analysis and management. `EXAM` can be used to perform a variety of functions, such as policy property analyses, policy similarity analysis, and policy integration. In contrast, our tool `XAnalyzer` also deals with policy analysis but focuses on policy anomaly detection and resolution.

8 Conclusion

We have proposed an innovative mechanism that facilitates systematic detection and resolution of XACML policy anomalies. A policy-based segmentation mechanism and a grid-based representation technique were introduced to achieve the goals of effective and efficient anomaly analysis. In addition, we have described an implementation of a policy anomaly analysis tool called XAnalyzer. Our experimental results showed that a policy designer could easily discover and resolve anomalies in an XACML policy with the help of XAnalyzer. We believe our systematic mechanism and tool will significantly help policy managers support an assurable Web application management service. As our future work, the coverage of our approach needs to be further extended with respect to obligations and user-defined functions in XACML. Moreover, we would explore how our anomaly analysis mechanism can be applied to other existing access control policy languages. In addition, we plan to conduct formal analysis [2], [12] of policy anomalies, particularly dealing with multi-valued requests.

References

[1] D. Agrawal, J. Giles, K. Lee, and J. Lobo. Policy ratification. In *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks, 2005*, pages 223–232, 2005.

[2] G. Ahn, H. Hu, J. Lee, and Y. Meng. Representing and Reasoning about Web Access Control Policies. In *34th Annual IEEE Computer Software and Applications Conference*, pages 137–146. IEEE, 2010.

[3] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE INFOCOM*, volume 4, pages 2605–2616. Citeseer, 2004.

[4] J. Alfaro, N. Boulahia-Cuppens, and F. Cuppens. Complete analysis of configuration rules to guarantee reliable network security policies. *International Journal of Information Security*, 7(2):103–122, 2008.

[5] A. Anderson. Evaluating xacml as a policy language. In *Technical report*. OASIS, 2003.

[6] L. Bauer, S. Garriss, and M. Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):2, 2011.

[7] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 223–234. ACM New York, NY, USA, 2008.

[8] J. Bryans. Reasoning about XACML policies using CSP. In *Proceedings of the 2005 workshop on Secure web services*, page 35. ACM, 2005.

[9] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on computers*, 100(35):677–691, 1986.

[10] Buddy. Buddy version 2.4, 2010. http://sourceforge.net/projects/buddy.

[11] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, pages 196–205. ACM, 2005.

[12] H. Hu and G. Ahn. Enabling verification and conformance testing for access control model. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 195–204. ACM, 2008.

[13] H. Hu, G. Ahn, and K. Kulkarni. Fame: a firewall anomaly management environment. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 17–26, 2010.

[14] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[15] JavaBDD. Java BDD, 2007. http://javabdd.sourceforge.net.

[16] J. Jin, G. Ahn, H. Hu, M. Covington, and X. Zhang. Patient-centric authorization framework for sharing electronic health records. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 125–134. ACM New York, NY, USA, 2009.

[17] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin. Access control policy combining: theory meets practice. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 135–144. ACM, 2009.

[18] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo. Exam: a comprehensive environment for the analysis of access control policies. *International Journal of Information Security*, 9(4):253–273, 2010.

[19] D. Lin, P. Rao, E. Bertino, and J. Lobo. An approach to evaluate policy similarity. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 1–10. ACM, 2007.

[20] A. Liu, F. Chen, J. Hwang, and T. Xie. XEngine: A Fast and Scalable XACML Policy Evaluation Engine. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):265–276, 2008.

[21] A. Liu, F. Chen, J. Hwang, and T. Xie. Designing Fast and Scalable Policy Evaluation Engines. *IEEE Transactions on Computers*, 60(12):1802–1817, 2011.

[22] A. Liu and M. Gouda. Complete redundancy detection in firewalls. In *Proceedings of 19th Annual IFIP Conference on Data and Applications Security*, 2005.

[23] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on software engineering*, 25(6):852–869, 1999.

[24] P. Mazzoleni, B. Crispo, S. Sivasubramanian, and E. Bertino. XACML Policy Integration Algorithms. *ACM Transactions on Information and System Security*, 11(1), 2008.

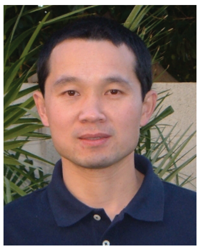
[25] T. Moses et al. Extensible access control markup language (XACML) version 2.0. *Oasis Standard*, 200502, 2005.

[26] P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo. An algebra for fine-grained integration of xacml policies. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 63–72. ACM, 2009.

[27] SunXACML. Sun XACML Implementation, 2006. http://sunxacml.sourceforge.net.

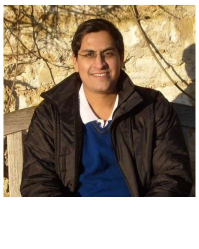
[28] XACML. OASIS XACML committee website. http://www.oasis-open.org/committees/xacml/, 2011.

[29] L. Yuan, H. Chen, J. Mai, C. Chuah, Z. Su, P. Mohapatra, and C. Davis. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy*, pages 199–213, 2006.



Hongxin Hu is an Assistant Professor in the Department of Computer and Information Sciences at Delaware State University. His current research interests include access control models and mechanisms, security and privacy in social networks, security in cloud and mobile computing, network and system security, and secure software engineering. He received the Ph.D. degree in computer science from Arizona State University, Tempe, AZ, in 2012.

Gail-Joon Ahn is an Associate Professor in the School of Computing, Informatics, and Decision Systems Engineering, Ira A. Fulton Schools of Engineering and the Director of Security Engineering for Future Computing Laboratory, Arizona State University. His research has been supported by the U.S. National Science Foundation, National Security Agency, U.S. Department of Defense, U.S. Department of Energy, Bank of America, Hewlett Packard, Microsoft, and Robert Wood Johnson Foundation. Dr. Ahn is a recipient of the U.S. Department of Energy CAREER Award and the Educator of the Year Award from the Federal Information Systems Security Educators Association. He received the Ph.D. degree in information technology from George Mason University, Fairfax, VA, in 2000.



Ketan Kulkarni received the Master degree in computer science from Arizona State University. He was also a member of the Security Engineering for Future Computing Laboratory, Arizona State University. He is currently working as a software engineer at NVIDIA.