Anomaly Discovery and Resolution in Web Access Control Policies

Hongxin Hu, Gail-Joon Ahn and Ketan Kulkarni
Arizona State University
Tempe, AZ 85287, USA
{hxhu,gahn,kakulkar}@asu.edu

ABSTRACT
The advent of emerging 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. We focus on XACML (eXtensible Access Control Markup Language) policy since XACML has become the de facto standard for specifying and enforcing access control policies for various Web-based applications and services. We introduce a policy-based segmentation technique to accurately identify policy anomalies and derive effective anomaly resolutions. We also discuss a proof-of-concept implementation of our method called XAnalyzer and demonstrate how efficiently our approach can discover and resolve policy anomalies.

Categories and Subject Descriptors
D.4.6 [Security and Protection]: Access controls

General Terms
Security, Management

Keywords
Access control policies, XACML, anomaly management

1. INTRODUCTION
With the explosive growth of Web applications and Web services deployed on the Internet, the use of a policy-based approach has received considerable attention to accommodate the security requirements covering large, open, distributed and heterogeneous computing environments. XACML (eXtensible Access Control Markup Language) [27], 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 [3], especially Web services. 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 remarkably 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 [27]: 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. With the significant growth of Web applications deployed on the Internet, XACML policies grow rapidly in size and complexity. Hence, redundancy elimination can be treated as one of effective solutions for optimizing XACML policies and improving the performance of XACML evaluation.

Recently, policy anomaly detection has received a great deal of attention [7, 10, 22, 28], especially, in firewall policy analysis. Corresponding policy analysis tools, such as Firewall Policy Advisor [7] and FIREMAN [28], with the goal of discovering firewall policy anomalies have been developed. However, we cannot directly adopt those prior 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 analysis approaches are still needed to be improved [8]. 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. Last but not the least, 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 those 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 overlap 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 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 [15, 18, 20], but cannot be directly supported by XACML. Thus, we additionally propose a flexible and extensible policy conflict resolution in this paper. Besides, we implement a policy analysis tool XAnalyzer based on our approach. To evaluate the practicality of our tool, our experiments deal with both real-life and synthetic XACML policies.

The rest of this paper is organized as follows. Section 2 overviews the XACML policy and 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. PRELIMINARIES

2.1 XACML Overview

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. For an applicable policy or policy set, the corresponding target should be evaluated to be true; otherwise, the policy or policy set is skipped when evaluating an access request. A rule set is a sequence of rules. Each rule consists of a target, a condition, and an effect. The target of a rule decides 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; the condition is a boolean expression to specify restrictions on the attributes in the target and refine the applicability of the rule; and the effect is either permit or deny. If an access request satisfies both the target and condition of a rule, the response is sent with the decision specified by the effect element in the rule. Otherwise, the response yields NotApplicable which is typically considered as deny.

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 [27]. Figure 1 shows an example XACML policy. The root policy set \( P_{S1} \) 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. Thus, 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 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 \( P_{S1} \) 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, \( P_2 \) 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 \( r_3 \) 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
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) [13] 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 [9]. 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. $\text{Subject} = \text{"student"}$) or on conditions (e.g. $8:00 \leq \text{Time} \leq 17:00$).

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

\[
(\text{Subject} = \text{"Designer"} \vee \text{Subject} = \text{"Tester"}) \wedge \\
(\text{Resource} = \text{"Codes"}) \wedge \\
(\text{Action} = \text{"Change"})
\]

**The Boolean expression for rule $r_3$ is:**

\[
(\text{Subject} = \text{"Designer"} \vee \text{Subject} = \text{"Tester"}) \wedge \\
(\text{Resource} = \text{"Reports"} \vee \text{Resource} = \text{"Codes"}) \wedge \\
(\text{Action} = \text{"Read"} \vee \text{Action} = \text{"Change"}) \wedge \\
(8:00 \leq \text{Time} \leq 17:00)
\]

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. [5] 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.

**Table 1: Atomic Boolean expressions and corresponding Boolean variables for $P_1$.**

<table>
<thead>
<tr>
<th>Unique Atomic Boolean Expression</th>
<th>Boolean Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Subject} = \text{&quot;Designer&quot;}$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$\text{Subject} = \text{&quot;Tester&quot;}$</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$\text{Subject} = \text{&quot;Developer&quot;}$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$\text{Subject} = \text{&quot;Manager&quot;}$</td>
<td>$S_4$</td>
</tr>
<tr>
<td>$\text{Resource} = \text{&quot;Reports&quot;}$</td>
<td>$R_1$</td>
</tr>
<tr>
<td>$\text{Resource} = \text{&quot;Codes&quot;}$</td>
<td>$R_2$</td>
</tr>
<tr>
<td>$\text{Action} = \text{&quot;Read&quot;}$</td>
<td>$A_1$</td>
</tr>
<tr>
<td>$\text{Action} = \text{&quot;Change&quot;}$</td>
<td>$A_2$</td>
</tr>
<tr>
<td>$8:00 \leq \text{Time} &lt; 12:00$</td>
<td>$C_1$</td>
</tr>
<tr>
<td>$12:00 \leq \text{Time} &lt; 13:00$</td>
<td>$C_2$</td>
</tr>
<tr>
<td>$13:00 \leq \text{Time} \leq 17:00$</td>
<td>$C_3$</td>
</tr>
</tbody>
</table>

We encode each of the atomic Boolean expression as a Boolean variable. For example, an atomic Boolean expression $\text{Subject} = \text{"Designer"}$ 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.

**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 $r_3$ is:**

\[
(S_1 \vee S_2) \wedge (R_1 \vee R_2) \wedge (A_1 \vee A_2) \wedge (C_1 \vee C_2 \vee C_3)
\]
Each conflicting segment indicates a policy conflict. Components with different effects are identified.

### 4.1 Conflict Detection Approach

$\mathcal{R}$, $\mathcal{P}$, and $\mathcal{P}_S$ be the set of rules, policies and policy sets, respectively, of an XACML policy $\pi$. An authorization space for an XACML policy component $c \in \mathcal{R}_x \cup \mathcal{P}_x \cup \mathcal{P}_S$ represents a collection of all access requests $Q_c$ to which a policy component $c$ is applicable.

#### 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.

Conflicting segments are identified as shown in lines 6-9 in Algorithm 1. A set of conflicting segments $\mathcal{C}^s : \{c_{s1}, c_{s2}, \ldots, c_{sn}\}$ from conflicting rules has the following three properties:

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

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 16-32 in Algorithm 1, a function called $\text{Partition}()$ 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 31). Therefore, one can utilize set operations to separate the overlapped spaces into disjoint spaces.

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 $\mathcal{C}^s$ for $P$.
1 * Partition the entire authorization space of $P$ into disjoint spaces /*
2 $P \leftarrow \text{Partition}(P)$;
3 $\mathcal{C}^s \leftarrow \mathcal{P}(P)$;
4 /* Identify the conflicting segments */
5 $\mathcal{C}^s \leftarrow \text{Partition}()$;
6 foreach $s \in S$ do
7 /* Get all rules associated with a segment $s$ */
8 $R' \leftarrow \text{GetRule}(s)$;
9 if $\exists r \in R, r \notin R', \forall i, \forall j, \forall Effect \neq i, Effect$ then
10 $\mathcal{C}^s \leftarrow \text{SApp}(s)$;
11 $\mathcal{C}^s \leftarrow \text{Partition}(P)$;
12 $R \leftarrow \text{GetRule}(P)$;
13 foreach $r \in R$ do
14 $s_r \leftarrow \text{AuthorizationSpace}(r)$;
15 $S \leftarrow \text{Partition}(S, s_r)$;
16 return $S$;
17 $\mathcal{C}^s \leftarrow \text{Partition}(S, s_r)$;
18 foreach $s \in S$ do
19 /* $s_r$ is a subset of $s$ */
20 if $s_r \subseteq s$ then
21 $S \leftarrow S \cup s_r$;
22 return $S$;
23 /* $s_r$ is a superset of $s$ */
24 if $s_r \supseteq s$ then
25 $S \leftarrow S \cup \text{SApp}(s \setminus s_r)$;
26 return $S$;
27 /* $s_r$ partially matches $s$ */
28 if $s_r \cap s \neq \emptyset$ then
29 $S \leftarrow S \cup \text{SApp}(s \setminus s_r)$;
30 $s \leftarrow s_r \cap s$;
31 $s_r \leftarrow s_r \setminus s$;
32 return $S$;
```

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

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 intersection of BDDs of $r_1$ and $r_2$. Note that the resulting BDD from the intersection operation 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 $\mathcal{R}_x$, $\mathcal{P}_x$ and $\mathcal{P}_S$ be the set of rules, policies and policy sets, respectively, of an XACML policy $\pi$. An authorization space for an XACML policy component $c \in \mathcal{R}_x \cup \mathcal{P}_x \cup \mathcal{P}_S$ represents a collection of all access requests $Q_c$ to which a policy component $c$ is applicable.

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 True (1) or False (0). Figures 2(a) and 2(b) give BDD representations of two rules $r_1$ and $r_2$, respectively.

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 16-32 in Algorithm 1, a function called $\text{Partition}()$ 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 31). Therefore, one can utilize set operations to separate the overlapped spaces into disjoint spaces.

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 $\mathcal{C}^s$ for $P$.
1 * Partition the entire authorization space of $P$ into disjoint spaces /*
2 $P \leftarrow \text{Partition}(P)$;
3 $\mathcal{C}^s \leftarrow \text{Partition}(P)$;
4 /* Identify the conflicting segments */
5 $\mathcal{C}^s \leftarrow \text{Partition}(P)$;
6 foreach $s \in S$ do
7 /* Get all rules associated with a segment $s$ */
8 $R' \leftarrow \text{GetRule}(s)$;
9 if $\exists r \in R, r \notin R', \forall i, \forall j, \forall Effect \neq i, Effect$ then
10 $\mathcal{C}^s \leftarrow \text{SApp}(s)$;
11 $\mathcal{C}^s \leftarrow \text{Partition}(P)$;
12 $R \leftarrow \text{GetRule}(P)$;
13 foreach $r \in R$ do
14 $s_r \leftarrow \text{AuthorizationSpace}(r)$;
15 $S \leftarrow \text{Partition}(S, s_r)$;
16 return $S$;
17 $\mathcal{C}^s \leftarrow \text{Partition}(S, s_r)$;
18 foreach $s \in S$ do
19 /* $s_r$ is a subset of $s$ */
20 if $s_r \subseteq s$ then
21 $S \leftarrow S \cup s_r$;
22 return $S$;
23 /* $s_r$ is a superset of $s$ */
24 if $s_r \supseteq s$ then
25 $S \leftarrow S \cup \text{SApp}(s \setminus s_r)$;
26 return $S$;
27 /* $s_r$ partially matches $s$ */
28 if $s_r \cap s \neq \emptyset$ then
29 $S \leftarrow S \cup \text{SApp}(s \setminus s_r)$;
30 $s \leftarrow s_r \cap s$;
31 $s_r \leftarrow s_r \setminus s$;
32 return $S$;
```

Figure 3 gives a representation of the segments of authorization space derived from the policy $P_1$ in the XACML example policy.
shown in Figure 1 \(^1\). We can notice that five unique disjoint segments are generated. In addition, three conflicting segments \(c_{S1}, c_{S2}\) and \(c_{S3}\) are identified. They represent three policy conflicts, where conflicting segment \(c_{S1}\) is associated with a rule set consisting of two rules \(r_1\) and \(r_2\), conflicting segment \(c_{S2}\) is related to a rule set including three rules \(r_1, r_2\) and \(r_3\), and conflicting segment \(c_{S3}\) is associated with a rule set containing two rules \(r_2\) and \(r_3\).

### 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 decided by XACML components covered by this segment. For non-conflicting 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 (CA), which is used by the owner (a policy or a policy set) of the segment.

1. **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. **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. **CA=Deny-Overrides**: The effect of a conflicting segment always equals to "Deny."
4. **CA=Only-One-Applicable**: The effect of a conflicting segment equals to the effect of only-applicable component.

To support the recursive specifications of XACML policies, we parse and model an XACML policy as a tree structure, where each terminal node represents an individual rule, each nonterminal node whose children are all terminal nodes represents a policy, and each nonterminal node whose children are all nonterminal nodes represents a policy set. At each nonterminal node, we store the target and combining algorithm. At each terminal node, the target and effect of the corresponding rule are stored.

Algorithm 2 shows the pseudocode of identifying disjoint conflicting authorization spaces for a policy set \(P_S\) based on the tree structure. In order to partition authorization spaces of all nodes contained in a policy set tree, this algorithm recursively calls the partition functions, \(\text{Partition}_P()\) and \(\text{Partition}_PS()\), 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 \(\text{permitted subspace} (E^P)\) and \(\text{denied subspace} (E^D)\) by aggregating all "Permit" segments and "Deny" segments, respectively, as follows:

\[
\begin{align*}
E^P &= \bigcup_{s_i \in E} \text{Effect}(s_i) = \text{Permit} \\
E^D &= \bigcup_{s_i \in E} \text{Effect}(s_i) = \text{Deny}
\end{align*}
\]

(1)

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

For example, Figure 4 shows the result of aggregating authorization spaces shown in Figure 3. Two subspaces \(P_S^P\) and \(P_S^D\) are constructed for the policy \(P_1\), which is a child node of the policy set \(P_{S1}\) in our example XACML policy.

In order to generate segments for the policy set \(P_S\), we can then leverage two subspaces \((E^P, E^D)\) of each child node \(E\) to partition existing authorization space set belonging to \(P_S\) (lines 28-29). Figure 5 represents an example of the segments of authorization space derived from policy set \(P_{S1}\) in our example policy (Figure 1). We can observe that seven unique disjoint segments are generated, and two of them \(c_{S1}\) and \(c_{S2}\) are conflicting segments, where \(c_{S1}\) is related to \(P_1^P\) and \(P_2^P\), and \(c_{S2}\) is associated with \(P_1^D\) and \(P_2^D\). They indicate two conflicts occurring in \(P_S\) at policy set level.

### 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 conflicts.
Algorithm 2: Identify Disjoint Conflicting Authorization Spaces of Policy Set \( PS \)

\[
\begin{array}{l}
\text{Input:} \text{ A policy set } PS \text{ with a set of policies or other policy sets.} \\
\text{Output:} \text{ A set of disjoint conflicting authorization spaces } CS \text{ for } PS. \\
1. P \leftarrow \text{Partition the entire authorization space of } PS \text{ into disjoint spaces} \forall \\
2. S.\text{New}(); \\
3. S \leftarrow \text{Partition}_PS(PS); \\
4. P \leftarrow \text{Identify the conflicting segments} \forall \\
5. CS.\text{New}(); \\
6. \text{foreach } s \in S \text{ do} \\
7. E \leftarrow \text{GetElement}(s); \\
8. \text{if } \exists e_i \in E, e_j \in E, e_i \neq e_j \text{ and } e_i.\text{Effect} \neq e_j.\text{Effect} \text{ then} \\
9. \text{CS.\text{Append}}(s); \\
10. \text{Partition}_PS(PS) \\
11. S'.\text{New}(); \\
12. C \leftarrow \text{GetChild}(PS); \\
13. \text{foreach } c \in C \text{ do} \\
14. S'.\text{New}(); \\
15. P \leftarrow \text{Partition}_P(c); \\
16. \text{if } \exists \text{Policy}(c) = \text{true} \text{ then} \\
17. S' \leftarrow \text{Partition}_P(c); \\
18. \text{else if } \exists \text{PolicySet}(c) = \text{true} \text{ then} \\
19. S' \leftarrow \text{Partition}_P(c); \\
20. E'\text{.New}(); \\
21. E'\text{.New}(); \\
22. \text{foreach } s' \in S' \text{ do} \\
23. \text{if } \text{Effect}(s') = \text{Permit} \text{ then} \\
24. E' \leftarrow E' \cup s'; \\
25. \text{else if } \text{Effect}(s') = \text{Deny} \text{ then} \\
26. E' \leftarrow E' \cup s'; \\
27. S'' \leftarrow \text{Partition}(S', E'); \\
28. S'' \leftarrow \text{Partition}(S', E'); \\
29. \text{return } S'';
\end{array}
\]

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 [15, 18, 20], 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.

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 associated with the conflicting segment. 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 [20], such as Recency-Overrides, Specificity-Overrides and High-Authority-Overrides, can be implied in our framework as well.

Figure 6: Fine-grained conflict resolution framework.

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. Therefore, the first-applicable component in each conflicting segment has the same effect with corresponding effect constraint.

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. We give a definition of rule redundancy as follows, which serves as a foundation of our redundancy elimination approach.

**Definition 2. (Rule Redundancy).** A rule \( r \) is redundant in an XACML policy \( p \) iff the authorization space derived from the resulting policy \( p' \) after removing \( r \) is equivalent to the authorization space defined by \( p \).

5.1 Redundancy Elimination at Policy Level

We employ following four steps to identify and eliminate rule redundancies at policy level: authorization space segmentation, property assignment for rule subspaces, rule correlation break, and redundant rule removal.

This example, two policy segments
authorization space segmentation for a policy with eight rules. In
conflicting overlapping other (conflicting 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 7(a) illustrates an authorization space segmentation for a policy with eight rules. In this example, two policy segments \( s_4 \) and \( s_5 \) are non-overlapping segments. Other policy segments are overlapping segments, including two conflicting overlapping segments \( s_1 \) and \( s_3 \), and two non-conflicting overlapping segments \( s_2 \) and \( s_5 \).

5.1.1 Authorization Space Segmentation

We first perform the policy segmentation function \( \text{Partition}_P \) defined in Algorithm 1 to divide the entire authorization space of a policy into disjoint segments. We classify the policy segments into 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 7(a) illustrates an authorization space segmentation for a policy with eight rules. In this example, two policy segments \( s_4 \) and \( s_5 \) are non-overlapping segments. Other policy segments are overlapping segments, including two conflicting overlapping segments \( s_1 \) and \( s_3 \), and two non-conflicting overlapping segments \( s_2 \) and \( s_5 \).

5.1.2 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 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 three processes to perform the property assignments to all of rule subspaces within the segments of a policy, considering different categories of policy segments.

**Process1:** 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.

**Process2:** 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 (CA).

1. \( \text{CA} = \text{First-Applicable} \): 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. \( \text{CA} = \text{Permit-Overrides} \): 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 weak irremovable property, other rule subspaces without irremovable property are assigned with removable property. Otherwise, all “permit” rule subspaces are assigned with correlated property.

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

**Process3:** Property assignment for rule subspaces covered by a non-conflicting overlapping segment. If any rule subspace has been assigned with weak 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 7(b) shows the result of applying our property assignment mechanism, which performs three property assignment processes in sequence, to the example presented in Figure 7(a). We can easily identify that \( r_3 \) and \( r_5 \) are removable rules, where all subspaces are with removable property. However, we need to further examine the correlated rules \( r_2 \), \( r_4 \) or \( r_7 \), which contain some subspaces with correlated property. The extension with correlation break algorithm remains in our future work. Figure 7(c) depicts the result of redundancy removal for the example.

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 8 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 $s_j$ of PS covers three policy segments $P_1.s_1$, $P_2.s_1$ and $P_3.s_1$, where $P_i.s_j$ denotes that a segment $s_j$ belongs to a policy $P_i$.

Figure 8: Example of authorization space segmentation at policy set level for redundancy discovery and removal.

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 [4] to parse the XACML policies and construct Boolean encoding. JavaBDD [2], which is based on BuDDy package [1], 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.

| Table 2: XACML policies used for evaluation. |
|-----------------|-----------------|-----------------|
| Policy          | Rule (#) | Policy (#) | Policy Set (#) |
| 1 (CodeA)       | 4        | 2       | 1              |
| 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)  | 316      | 305     | 111            |
| 8 (SyntheticPolicy-2) | 456    | 65      | 40             |
| 9 (SyntheticPolicy-3) | 572   | 114     | 75             |
| 10 (SyntheticPolicy-4) | 685   | 188     | 84             |

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 [14]; among them, Continue-a and Continue-b are designed for a real-world Web application supporting a conference management. GradeSheet is utilized in [11]. The Pluto policy is employed in ARCHON 2 system, which is a digital library that federates the collections of physics with multiple degrees of meta data richness. Since it is hard to get a large volume of real-world policies due to the reason that they are often considered to be highly confidential, 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. 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 3 summarizes our evaluation results.

[Evaluation of Conflict Detection]: Time required by XAnalyzer

http://archon.cs.odu.edu/
7. RELATED WORK

Many research efforts have been devoted to XACML. However, most existing research work focus on modeling and verification of XACML policies [6, 12, 17, 14, 19]. None of them dealt with anomaly analysis in XACML policies. We discuss a few of those work here.

In [12], 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. [14] introduced an approach to represent XACML policies with Multi-Terminal Binary Decision Diagrams (MTBDDs). A policy analysis tool called Margrave was developed. Margrave can verify XACML policies against the given properties and perform change-impact analysis based on the semantic differences between the MTBDDs representing the policies. Ahn et al. [6] 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.

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. [7] designed a tool called Firewall Policy Advisor which can only detect pairwise anomalies in firewall rules. Yuan et al. [28] 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. However, the anomaly detection procedures of FIREMAN are still incomplete [8]. Our tool, XAnalyzer, could conduct a complete examination of policy anomaly and provide more accurate anomaly diagnosis information for policy analysis. On the other hand, as we discussed previously, XACML policy and firewall policy have some significant distinctions. Hence, directly applying prior policy anomaly analysis approaches to XACML are not suitable.

Some XACML policy evaluation engines, such as Sun PDP [4] and XEngine [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 including Fundulaki et al. [15], Fisler et al. [14] and Jajodia et al. [18]. Especially, Li et al. [20] 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.

Other related work includes XACML policy integration [24, 25, 26] and XACML policy optimization [21, 23]. Since anomaly discovery and resolution are challenging issues in policy integration and redundancy elimination can contribute in policy optimization, all of those related work are orthogonal to our work.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Partitions (g)</th>
<th>BDD Nodes (g)</th>
<th>Conflict Detection</th>
<th>Redundant Removal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (CodeA)</td>
<td>6</td>
<td>16</td>
<td>1</td>
<td>0.082</td>
</tr>
<tr>
<td>2 (SamplePolicy)</td>
<td>8</td>
<td>34</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3 (GradeSheet)</td>
<td>18</td>
<td>45</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4 (Plato)</td>
<td>34</td>
<td>78</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5 (SyntheticPolicy-1)</td>
<td>205</td>
<td>112</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>6 (Continue-a)</td>
<td>439</td>
<td>135</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>7 (Continue-b)</td>
<td>486</td>
<td>146</td>
<td>10</td>
<td>21</td>
</tr>
<tr>
<td>8 (SyntheticPolicy-2)</td>
<td>523</td>
<td>209</td>
<td>29</td>
<td>17</td>
</tr>
<tr>
<td>9 (SyntheticPolicy-3)</td>
<td>614</td>
<td>227</td>
<td>39</td>
<td>19</td>
</tr>
<tr>
<td>10 (SyntheticPolicy-4)</td>
<td>814</td>
<td>265</td>
<td>56</td>
<td>19</td>
</tr>
</tbody>
</table>

Figure 9: Evaluation of redundancy removal approach.
8. CONCLUSION

We have proposed an innovative mechanism that facilitates systematic detection and resolution of XACML policy anomalies. A policy-based segmentation technique was 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 assured 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. In addition, we would like to extend our tool with information visualization techniques [16], providing an intuitive cognitive sense for policy anomaly to facilitate a more effective policy management. Moreover, we would explore how our anomaly analysis mechanism can be applied to other existing access control policy languages.

Acknowledgments

This work was partially supported by the grants from National Science Foundation (NSF-IIS-0900970 and NSF-CNS-0831360) and Department of Energy (DE-SC0004308 and DE-FG02-03ER25565).

9. REFERENCES