Systematic Policy Analysis and Management

by

Ketan Kulkarni

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2011 by the
Graduate Supervisory Committee:

Gail-Joon Ahn, Chair
Stephen S. Yau
Dijiang Huang

ARIZONA STATE UNIVERSITY

May 2011

ABSTRACT

With the advent of technologies such as web services, service oriented architecture and cloud computing, modern organizations have to deal with policies such as Firewall policies to secure the networks, XACML (eXtensible Access Control Markup Language) policies for controlling the access to critical information as well as resources. Management of these policies is an extremely important task in order to avoid unintended security leakages via illegal accesses, while maintaining proper access to services for legitimate users. Managing and maintaining access control policies manually over long period of time is an error prone task due to their inherent complex nature. Existing tools and mechanisms for policy management use different approaches for different types of policies. This research thesis represents a generic framework to provide an unified approach for policy analysis and management of different types of policies. Generic approach captures the common semantics and structure of different access control policies with the notion of policy ontology. Policy ontology representation is then utilized for effectively analyzing and managing the policies. This thesis also discusses a proof-of-concept implementation of the proposed generic framework and demonstrates how efficiently this unified approach can be used for analysis and management of different types of access control policies.

To my parents, my brother, and all my friends.

ACKNOWLEDGEMENTS

My graduate study in Computer Science has been a great journey as well as learning experience for me. It has contributed significantly towards my abilities, to approach and solve different research problems. I would like to offer my sincere appreciation and regards to my advisor Dr. Gail-Joon Ahn, who has been a constant source of motivation throughout this journey. He has always provided me with valuable guidance in my research area. I would like to specially offer sincere regards to my colleague Mr. Hongxin Hu, who has always inspired and helped me in my research work. I would also like to extend my gratitude to Dr. Stephen S. Yau and Dr. Dijiang Huang for serving on my committee. It has been great experience working in the Secure Engineering For Future Computing (SEFCOM) Lab. I would like to thank each and every member of this lab for their valuable inputs during my research work. I would also like to thank all of my friends who had always been a constant support for me during this wonderful journey. I would like to extend sincere regards and love to my family for being a constant source of motivation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

We have witnessed explosive growth of cloud computing technologies, service oriented architecture (SOA) as well as social networks. Cloud computing and SOA have enabled the on-demand network access to a shared pool of configurable computing resources such as networks, servers, storage, and application services [3]. Social networks also allow users to share personal data and interact with each other. To achieve this, these technologies heavily rely upon different levels of interaction between remote systems/platforms, networks, data, services and users.

The advent of these technologies have provided different challenges for controlling the access to critical resources as well as information. Previously client applications and data would generally reside on dedicated servers. Hence access control was mainly focused on shell defenses like Firewall, intrusion detection systems and policy based mechanisms for critical resources. In addition, Cloud computing and SOA technologies brought the concept of multi-tenancy for serving various subscribers through a common pool of resources. In such an environment, it is necessary to have strong access control mechanism to prevent unintended use of shared common resources and private user data. These concerns have lead to considerable attention towards the research area covering policy-based approach for access control in large, open, distributed and heterogeneous environment.

A *policy*, the basic building block of policy-based system, is a set of rules that control the behavior of system. Policy-based approach controls complex system behavior by separating policies from system implementation and enabling dynamic adaptability of system behavior by changing policy configurations without changing system implementation. As discussed above, in the era of cloud, web-oriented computing policy-based mechanism have gained importance for controlling access to shared common resources as well as private data. Different types of access control policies are used at different levels. For example, application level access control policies like XACML are used to control access to user information and documents. While firewall policies are used to control the access to network devices which can be categorized into network level access control.

Research in the policy-based systems has been divided into two major areas. One area considers enhancement of policy decision point (PDP) and policy enforcement point (PEP). Other area covers policy management and analysis mechanism. Policies that control the behavior of modern systems are exponentially growing in size and complexity. In typical policies, multiple rules may overlap, which mean one access request may match several rules. These multiple rules within same policy may conflict, implying that those rules not only overlap each other but also yield different decisions. Conflicts in these policies may lead to security issues (e.g. allowing unintended access) as well as availability issues (e.g. denying legitimate access). On the other hand, there might be some rules that are redundant meaning that access request matching one rule also matches other rules with the same effect. In such cases, performance of access control system might be degraded since it directly depends on the number of rules evaluated per access request. An intuitive means for policy designer or manager to resolve these conflicts is to identify and remove all policy conflicts and redundancies by analyzing the policies.

Identifying and resolving conflicts as well as redundancies by modifying policies is remarkably difficult and nearly impossible task in practice from many aspects. First, the number of conflicts in policies is typically large considering that they generally have thousands of rules. Second, one conflict might be associated with several rules. Hence it is important to consider policy as whole and not in parts while resolving those conflicts. With the large number of rules and conflicts, it is almost impossible to identify and analyze these conflicts manually. Besides, policies are generally maintained by multiple administrators over their lifetime. Without the prior knowledge of actual policy specification intentions, modifications may lead to incorrect policy semantics.

Research in this area has resulted into policy analysis tools, such as Firewall Policy Advisor [16] and FIREMAN [32], with the goal of discovering firewall policy conflicts. For XACML policies, tools like XAnalyzer [25] have been developed for helping policy administrators to identify conflicts as well as redundancies in XACML policies. Most of these prior approaches handle management of a particular type of policy. Because of this, policy administrator has to face the difficult task of getting familiar with each of these tools or approaches for every different type of policies and may get confused with their different methods. Therefore, a unified policy analysis and management mechanism is desirable which can manage different types of access control policies.

In this thesis, we present innovative approach of representing different access control policies using ontology, which can then be used for our unified policy analysis and management approach for different type of access control policies. This approach uses policy-based segmentation technique for policy anomaly detection. Accurate policy anomaly information can be used for policy administrators to resolve policy anomalies. As part of this research, we have also implemented this unified policy analysis and management approach as a proof-of-concept tool for policy anomaly detection and resolution.

The remainder of document is organized as follows. Chapter 2 overviews policy anomalies with example and need for generic policy management framework. Chapter 3 presents the requirements of generic access control policy management framework and its components. We describe realization of generic policy representation approach using policy ontology in Chapter 4. In Chapter 5, we discuss the realization of policy analysis and management approach. Chapter 6 discusses implementation details along with evaluation of our approach. This thesis concludes with directions for possible future work in Chapter 7.

Chapter 2

BACKGROUND

The aim of policy-based management is to apply an organization wide integrated management that includes system management, network management and application management. Different types of policies are used at different levels of system. For network operations point of view, policy-based network management is about minimizing complexity of end-to-end management and provides security. In case of application management, policy-based systems are generally used to control access to shared resources. Figure 2.1 shows general components of a policy-based management system. It has mainly three components:

1. *Policy Decision Point (PDP)*: It evaluates and authorizes the policy decisions.
2. *Policy Enforcement Point (PEP)*: This component intercepts access request and enforces the decisions made by PDP.
3. *Policy Information Point (PIP)*: It provides external information to PDP such as LDAP attributes for a user requesting access to the system.

Different types of polices exist. To enumerate few, Firewall policies are used for network based access control. XACML and Ponder [8] policies are used for application level access control. Effectiveness of policy-based mechanism depends

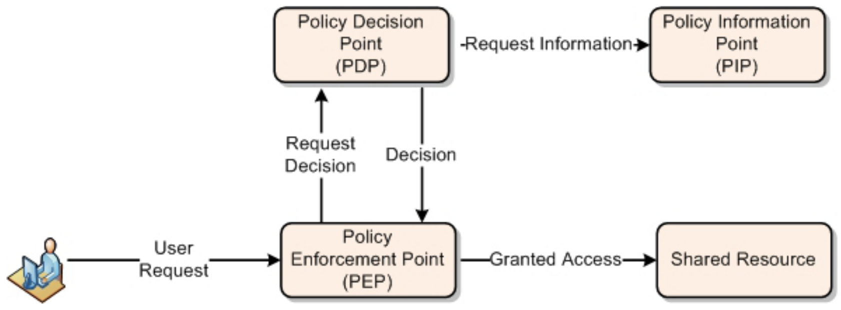
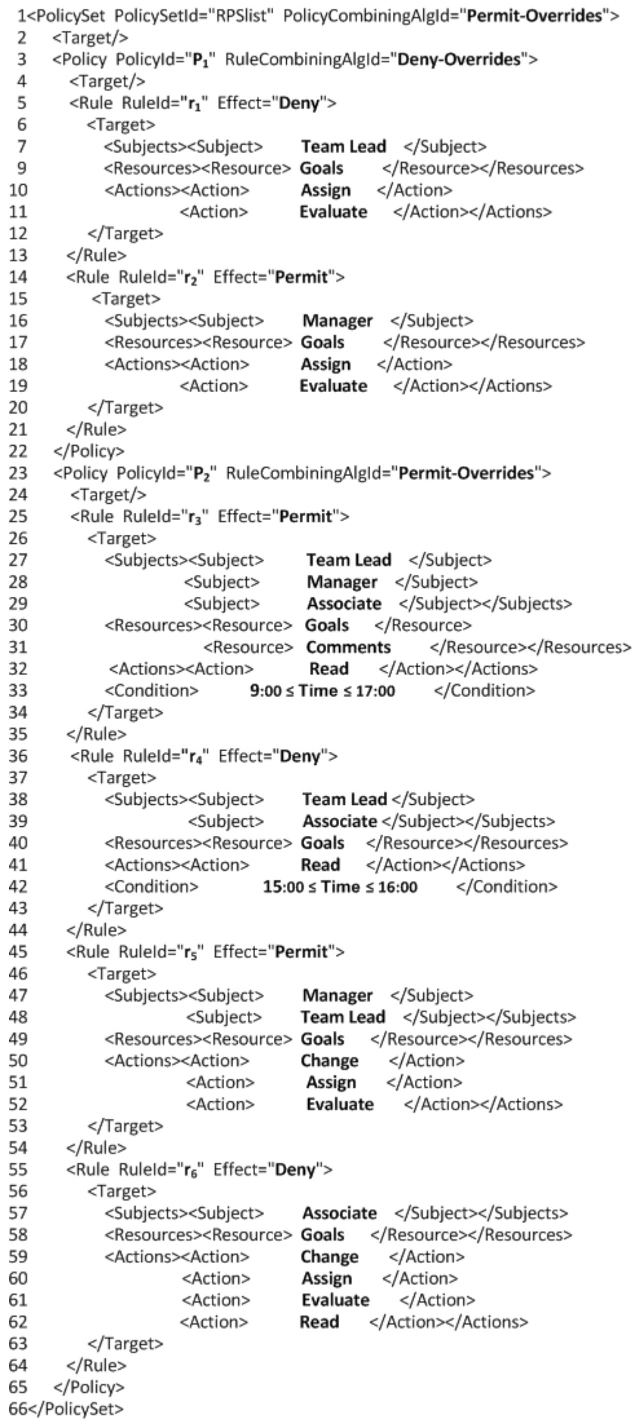

Policy Decision Point (PDP)
Request Information
Policy Information Point (PIP)
Request Decision
Decision
User Request
Policy Enforcement Point (PEP)
Granted Access
Shared Resource

Figure 2.1: Policy Based Management System

upon quality and correctness of the policy used. As discussed in Chapter 1, policy anomalies always exist due to various reasons. Policy anomalies are mainly of two types: (a) *Policy Conflicts* (b) *Policy Redundancies*. To consider how these policy anomalies exist in the real world, let us consider an example of the *Employee Appraisal System*. In this system, we have different *actors* such as *Associate, Team Lead, Manager* who can perform different activities such as *Assign, Evaluate, Change* the *Goals* as well as *Read or Change* some *Comments or Feedback* for assigned goals. System should enforce certain behaviors defined at an organization-level. For example, Goals can be assigned by Team Lead or Manager only and not by an Associate. Figure 2.2 shows an example XACML policy, which defines the behavior of the employee appraisal system.

At root level, example XACML policy consists of a *Policy set*, which further contain two *policies*. Each policy consists of *rule set*. 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`. An XACML policy often has conflicting rules or policies, which are resolved by four different *combining algorithms* defined in XACML specifications. The root policy set $RPSlist$ contains two policies, P_1 and P_2, which are combined using *Permit-Overrides* combining algorithm. The policy P_1 has two rules, r_1 and r_2, and its rule combining algorithm is *Deny-Overrides*. The policy P_2 includes four rules r_3, r_4, r_5 and r_6 with *Permit-Overrides* as a combining algorithm. In this example, there are three subjects: *Manager*, *Team Lead* and *Associate*; two resources: *Goals* and *Comments*; and four actions: *Read*, *Change*, *Assign* and *Evaluate*. Note that both r_3 and r_4 define conditions over the *Time* attribute.



```
<PolicySet  PolicySetId="RPSlist"  PolicyCombiningAlgId="Permit-Overrides">
  <Target/>
  <Policy  PolicyId="P1"  RuleCombiningAlgId="Deny-Overrides">
    <Target/>
    <Rule  RuleId="r1"  Effect="Deny">
      <Target>
        <Subjects><Subject>       Team Lead   </Subject>
        <Resources><Resource>  Goals       </Resource></Resources>
        <Actions><Action>         Assign     </Action>
                    <Action>         Evaluate     </Action></Actions>
      </Target>
    </Rule>
    <Rule  RuleId="r2"  Effect="Permit">
       <Target>
        <Subjects><Subject>       Manager    </Subject>
        <Resources><Resource>  Goals       </Resource></Resources>
        <Actions><Action>         Assign     </Action>
                    <Action>         Evaluate     </Action></Actions>
      </Target>
    </Rule>
  </Policy>
  <Policy  PolicyId="P2"  RuleCombiningAlgId="Permit-Overrides">
    <Target/>
    <Rule  RuleId="r3"  Effect="Permit">
      <Target>
        <Subjects><Subject>       Team Lead    </Subject>
                    <Subject>        Manager    </Subject>
                    <Subject>        Associate    </Subject></Subjects>
        <Resources><Resource>  Goals     </Resource>
                      <Resource>  Comments       </Resource></Resources>
         <Actions><Action>          Read       </Action></Actions>
        <Condition>          9:00 ≤ Time ≤ 17:00        </Condition>
      </Target>
    </Rule>
     <Rule  RuleId="r4"  Effect="Deny">
      <Target>
        <Subjects><Subject>       Team Lead </Subject>
                     <Subject>       Associate </Subject></Subjects>
        <Resources><Resource>  Goals     </Resource></Resources>
        <Actions><Action>         Read      </Action></Actions>
        <Condition>            15:00 ≤ Time ≤ 16:00        </Condition>
      </Target>
    </Rule>
    <Rule  RuleId="r5"  Effect="Permit">
      <Target>
        <Subjects><Subject>       Manager    </Subject>
                     <Subject>       Team Lead    </Subject></Subjects>
        <Resources><Resource>  Goals      </Resource></Resources>
        <Actions><Action>         Change      </Action>
                     <Action>         Assign      </Action>
                     <Action>         Evaluate      </Action></Actions>
      </Target>
    </Rule>
    <Rule  RuleId="r6"  Effect="Deny">
      <Target>
        <Subjects><Subject>       Associate    </Subject></Subjects>
        <Resources><Resource>  Goals      </Resource></Resources>
        <Actions><Action>         Change      </Action>
                    <Action>         Assign      </Action>
                    <Action>         Evaluate      </Action>
                    <Action>         Read      </Action></Actions>
      </Target>
    </Rule>
  </Policy>
</PolicySet>
```

Figure 2.2: An example XACML Policy

Anomalies in Example Policy

In case of policies having hierarchical structure (i.e. root level policy set may contain one or more policy sets or policies) like XACML policy, anomalies occur at two different levels. First, anomalies occur at policy level because of conflicting or redundant policy rules. Secondly, anomalies may exist at policy set or group level because of conflicting or redundant policy or policy set components. Following anomalies could be found in the example XACML policy:

- **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_4 is in conflict with the *permit* rule r_3 in Figure 2.2 because rule r_3 allows the access requests from a team lead and associate to read goals in the time interval [9:00, 17:00], which are supposed to be denied by r_4 during the time interval [15:00, 16:00]; 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_4 to *Permit*, r_4 becomes redundant since r_3 will also permits a team lead and associate to read goals in the time interval [15:00, 16:00].
- **Anomalies at Policy Set Level:** Anomalies may also occur across policies and/or policy sets. For example, considering two policy components P_1 and P_2 of the policy set $RPSlist$ in Figure 2.2, P_1 is ***conflicting*** with P_2, because P_1 permits the access requests that a team lead assigns or evaluates the goals, which are denied by P_2. r_2 is ***redundant*** with respect to r_5, even though r_2 and r_5 are placed in different policies P_1 and P_2, respectively.

Considering above example and inherent complex nature of the real world access control policies, it is safe to assume that identifying and removing the policy anomalies manually, is a very difficult and error prone task. Hence, it is necessary to automate the policy analysis and management approach for policy administrators in order to ensure the quality and correctness of access control policies. Many research efforts have been devoted to identify policy analysis and management approach for different types of policies. Fisler et al. [22] 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. Several work presenting policy analysis tools with the goal of detecting policy anomalies in firewall are closely related to this work. Al-Shaer et al. [16] designed a tool called Firewall Policy Advisor which can only detect *pairwise* anomalies in firewall rules. Yuan et al. [32] 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 [17]. Generic policy analysis and management framework designed as a part of this research thesis is the result of experience gained from our previous work [24] [25].

However, each of the above approach considers specific type of policy such as XACML or Firewall only. Policy administrator in the real world deals with different types of access control policies. It is very difficult for administrators to use different tools and approaches for each different type of policy. Sometimes it might be confusing to use different tools and may lead to errors in the policy analysis and management tasks. Hence, it is desirable to have a generic approach for the policy analysis and management, which can consider different types of policies. It will help to maintain consistency in the policy analysis and management tasks. In this work, we attempt to design a generic approach along with the corresponding tool for accurate anomaly detection as well as effective anomaly resolution in the access control policies.

Chapter 3

FRAMEWORK FOR GENERIC POLICY MANAGEMENT

Chapters 1 and 2 discussed about the policy anomalies, need for the automation as well as benefits of generic framework for the policy analysis and management tasks. The motivation behind the generic framework is to support policy analysis and management of different types of access control polices under a single unified approach. These different types of policies may include network level policies such as Firewall or application level policies like XACML. We analyzed the policy analysis and management tasks performed by the policy administrators in order to identify the components required for a generic policy analysis and management framework. We also studied policy specifications for the access control policies like Firewall, XACML and Ponder, to extract common semantics and structure of the access control policies. Based upon this literature survey, we have come up with the generic policy analysis and management framework for access control policies. Figure 3.1 shows the components involved in this generic framework and their interaction with each other to achieve effective policy analysis and management of access control policies under a single unified approach.

Generic access control policy management framework is divided into three major parts:

Part 1: *Generic Access Control Policy Representation*. This part deals with the conversion of different types of access control policies into generic internal representation, which can be further utilized by the policy analysis and management tasks. Generic policy representation enables policy analysis and management to be independent of the policy type. Hence, it is an important component in the generic policy management framework.

Part 2: *Policy Analysis and Management*. This part of generic framework encom-

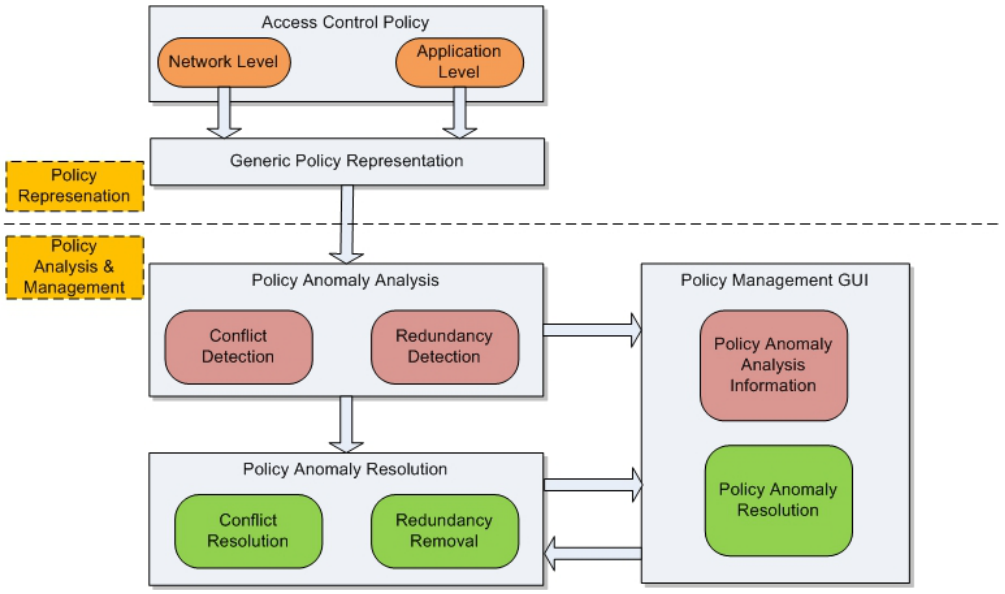


Figure 3.1: Generic Access Control Policy Management Framework

passes the policy analysis and management tasks. These tasks include policy anomaly detection and resolution. It uses the generic policy representation.

Part 3: *Policy Administration User Interface*. We consider user interface for policy analysis and management tasks to be an integral part of the framework. It is extremely important to provide policy administrators with an intuitive user interface that will help them to easily understand the policy management process as well as information about policy anomalies and their resolution.

Following subsections describe the details and requirements for each of the subcomponent of a generic policy analysis and management framework.

3.1 Generic Access Control Policy Representation

Generic access control policy representation enables the policy analysis and management components to be independent of different access control policy types. Hence, for the design of a generic policy management framework, it can be considered as a central point of focus. In order to design generic access control policy representation we analyzed the specifications of access control policies like Firewall [2] and XACML [4] as well as some of the available test set policies. We observed that access control policy representation has three important characteristics:

1. *Policy Domain Concepts*: Access control policy domain concepts can be considered as the terms that are generally used to describe the access control policies.
2. *Policy Structure*: Structure can be considered as the depiction of how policy components are arranged within policy with respect to each other.
3. *Policy Semantics*: Semantics capture the relationship between policy components and also describe the behavior of a policy.

In order to achieve effective and comprehensive policy analysis and management, generic policy representation should successfully capture above characteristics from different types of policies. Following subsections describe how to capture above mentioned characteristics from different types of access control policies to design a generic policy representation. The process of capturing these characteristics is a continuous process. These characteristics should be revised whenever changes are made to the policy specifications or new policy types are considered for generic approach of policy management.

3.1.1 Capturing Policy Domain Concepts

Capturing access control policy domain concepts can be considered as a first step towards the capturing generic policy representation. We analyzed specifications of different access control policies and enlisted the terms used for expressing those policies. After that, we tried to classify these terms from different type of policies under common classes that can be considered as access control policy domain concepts. For example, we enlisted terms like rule, policy, policy set, access control list, subject, action ,resource, combination algorithm, conflict resolution strategy and many more. After that we classified them under common concepts such as Policy, Policy Group, PolicyRule. To give an example of our classification process, consider XACML policy has the notion of *Combination Algorithm*, which describes the policy behavior in case of conflicts. Firewall policy uses default *first-matched* strategy for conflict resolution. Both can be classified under one class *Meta-Policy*, which can be considered as policy about the behavior of an access control policy in case of a decision conflict.

3.1.2 Capturing Generic Policy Structure

We analyzed the structure of different access control policies. XACML policy structure has the notion of *<Rule>, <Policy> and <PolicySet>*. PolicySet is the container for other policies as well as policy sets. Policy defines the list of rules. Consider other access control policy such as firewall, it has the concept of *Access Control List (ACL)* which includes the list of firewall rules. Typical firewall policy may contain number of ACLs. We can consider each ACL similar to *policy* node containing group of rules. While firewall policy can be considered as a group of number of ACLs or *Policy* nodes.

Based on our observation from the test set policies and study of different policy specifications, we designed generic access control policy structure. Access control policy defines what activities a member of the *subject* domain can perform on the set of objects in the *Resource* domain. Basic node or unit for defining policy is *Policy Rule*. These rules either permit or deny access to the resource objects and hence can be classified into two types: *Positive Rule* and *Negative Rule* respectively. Rules that are applicable to the same subject or resource objects can

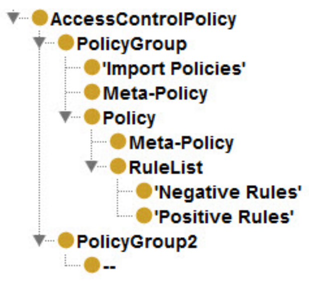


Figure 3.2: Access Control Policy Structure

be arranged into the *Policy* node as a *Rule List*. Each *Policy* node may have meta-policies associated with it, which specify policy about policy behavior. For example, conflict resolution strategy specified for policy node defines the behavior of a policy in case of conflict. Further access control policies are grouped together to generate composite policies. Related policies may be grouped together into one *Policy Group*. For example, policies related to same department or same application may grouped together for better policy organization and management. This policy group may also have meta-policies associated with it. Policy group may choose to refer to the existing policy groups or polices instead of defining them again. These referred policies are called as *Import Policies*. Access control policy might contain *Policy* as root node or it might have a hierarchical structure where root node is *Policy Group* containing other policies or policy groups. Figure 3.2, depicts the generic access control policy structure.

3.1.3 Capturing Generic Policy Semantics

Access control policies are defined in terms of the policy attributes. Attributes are named values of known types and are characteristics of the *Subject*, *Resource*, *Action* in which the access requests are made. Access control policy rules are defined based on these attributes. Generally high-level semantics of a rule in an access control policy can be described as *which subject(s) has access* *to which resource(s) and with what action(s) permitted or denied?* Thus at the unit level of policy semantics, access control policies are typically based on the policy rules expressed in terms of attributes of type *Subject*, *Resource*, *Action* and *Effect*. Additionally rules might have conditions, that need to be satisfied for making access control decisions.

To capture the high-level semantics mentioned above in generic policy representation, we need to identify attributes of type *Subject*, *Resource*, *Action* and *Effect* from different types of policies. Additionally, we also need to identify attributes such as *condition*, *conflict resolution strategy* for comprehensive access control policy representation.

Consider rule r_3 from the example XACML policy in Chapter 2. We can easily extract following attributes using XACML attribute designators:

Subject *Team Lead, Manager, Associate*.
Resource *Goals, Comments*.
Action *Read*.
Condition *Between 9 AM to 5 PM*
Effect *Permit*.

Now, consider following Firewall policy rule:

```
deny udp 149.169.112.73 eq 20 192.168.99.61 eq 137
```

In this case it not easy to extract the required attributes from this rule. However, consider the high level semantics of a firewall policy which says: *source-ip (or network space) wants to access resource-ip (or network space) with specific protocol has effect deny or allow*. According to this we can identify required policy attributes as follows:

Subject *149.169.112.73 : 20*.

Resource *192.168.99.61 : 137*.

Action *access (dummy action)*.

Condition *(protocol==udp)*

Effect *Deny*.

Apart from the above policy rule level semantics, a policy also specifies attributes such as *conflict resolution strategy* which define the behavior of a policy in case of conflicting policy decisions. For example, XACML defines *permit-override, deny-override, first-applicable and only-one applicable* strategies, while firewall policy uses default *first-matched* strategy. Attributes like *conflict resolution strategy* which define the policy behavior, need to be identified and associated with proper policy structure components such as *Policy* or *Policy Group* node for effective and comprehensive generic policy representation.

3.1.4 Framework for Capturing Generic Policy Structure and Semantics

Figure 3.3 shows a framework for capturing generic policy structure and semantics mentioned previously. *Policy information extractor* component captures the required policy structure information such as *Policy, Policy Set or Rules* as well as policy semantics information such as policy attributes from a given policy. Policy information extractor is specific for each type of access control policy. *Policy Instance Generator* component creates a generic access control policy instance based upon the information captured by a policy information extractor and previously identified *generic policy structure and semantics*. This generic policy instance can be directly used as a generic policy representation for policy analysis and management tasks or may be converted to an efficient data structure representation based upon the nature of operations required for the policy analysis and management.

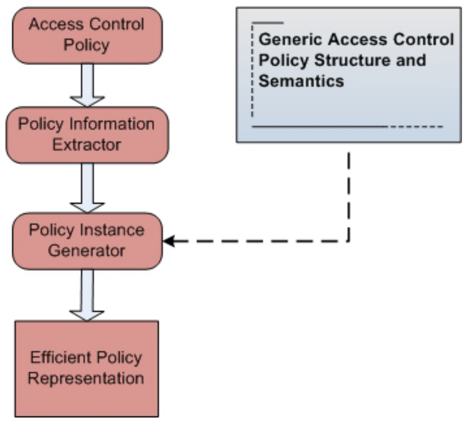

Access Control Policy
Policy Information Extractor
Policy Instance Generator
Efficient Policy Representation
Generic Access Control Policy Structure and Semantics

Figure 3.3: Framework for capturing access control policy structure and semantics

3.2 Policy Analysis and Management

Policy anomalies are always present due to the inherent complex nature of access control policies. Policy anomalies may lead to an unintended access to critical resources or denial of a legitimate access request in a policy-based system. To remove these anomalies, policies must be analyzed periodically or whenever new changes are introduced. Considering large number of components involved in policies and their inherent complex nature, manual analysis of policies for anomaly detection is almost impossible or error prone task. Hence, an automated approach is necessary for consistent and effective policy analysis and management. In our approach, we consider two types of policy anomalies detection and resolution:

1. *Policy Conflicts*: Chapter 2 discussed about the conflicts in an access control policy. Several policy components may be involved in the policy conflicts. Also due to the composite or hierarchical policies, conflicts may be present at different levels of policy structure. Thus, conflicts must be identified by considering a policy as a whole and not just comparing the two policy components. In this way, approach should be able to detect the policy conflicts present at different levels of a policy structure. Once conflicts are identified,

existing mechanisms for the conflict resolution can be applied to resolve the conflicts. However, we observed that existing conflict resolution mechanisms are too restrictive and hence administrator should be able to apply more fine grained mechanisms for conflict resolution.

2. *Policy Redundancies*: Chapter 2 also discussed about the redundancies present in an access control policy. Redundant components may also be present at different levels of a policy structure. Hence for comprehensive redundancy analysis, policies must be considered as a whole similar to the conflict detection approach. Redundant components may be overlapping with the other policy components. In that case, redundancy removal approach should make sure that all subspaces of redundant components are *removable* without changing the existing policy semantics.

Figure 3.1 shows the framework components involved in the policy analysis and management tasks. Chapter 5 discusses our comprehensive policy anomaly detection and removal approach using generic policy representation.

3.3 Policy Administration User Interface

Policy administration user interface is an essential part of the generic policy management framework. It provides administrator with the detailed information about policy anomalies such as policy conflicts and redundancies. Important feature of user interface for generic policy management framework is the depiction of information does not change with the change in policy type. It helps in maintaining unified and unambiguous approach for policy management. Figure 3.4 shows components of the user interface for policy analysis and management. It has mainly two components:

1. *Conflict Viewer*: This component provides administrators with the precise in-

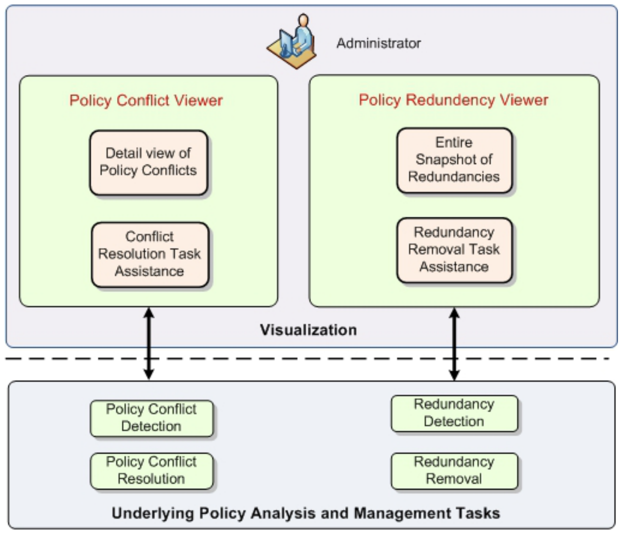

Administrator
Policy Conflict Viewer
Detail view of Policy Conflicts
Conflict Resolution Task Assistance
Policy Redundancy Viewer
Entire Snapshot of Redundancies
Redundancy Removal Task Assistance
Visualization
Policy Conflict Detection
Policy Conflict Resolution
Redundancy Detection
Redundancy Removal
Underlying Policy Analysis and Management Tasks

Figure 3.4: Policy Administration User Interface Components

formation about policy components involved in the policy conflicts. It helps to guide them through conflict resolution process and provides them an opportunity to interfere in the resolution process at the precise points such as selection of a conflict resolution strategy.

2. *Redundancy Viewer*: It provides administrators with the intuitive information about redundant policy components and help them in policy analysis. It also guides them through a redundancy removal process.

Chapter 4

REALIZATION OF GENERIC REPRESENTATION FOR ACCESS CONTROL POLICIES

Chapter 3 discussed about the framework for generic approach of access control policy analysis and management. It discussed about the required components and their interaction with each other. Generic framework for access control policy analysis and management is divided into three major parts: (a) *Generic Representation of Access Control Policy* (b) *Policy Analysis and Management* and (c) *Policy Administration User Interface*. This chapter discusses the realization of a system architecture and workflow for *Generic Representation of Access Control Policy*.

Process of creating generic representation of access control policy involves three major steps:

Step 1: *Generic Ontology for access control policies*. Chapter 3 discussed about the capturing access control policy domain concepts, structure and semantics for generic policy representation. Generic characteristics identified through the analysis of different types of policies needs to be defined somewhere and we use generic access control policy ontology for this purpose. Policy ontology is then used to create the generic policy representation from different types of access control policies.

Step 2: *Generating policy specific instance*. Based upon the policy ontology, required structure and semantics information needs to be extracted from the access control policy under consideration. Ontology instance instantiated with the information extracted from a policy represents the policy specific instance.

Step 3: *Generating efficient representation*. Realization of the policy analysis and management involves complex algorithms. These algorithms typically involve a lot of *set* operations. To carry out these tasks efficiently, we need to convert

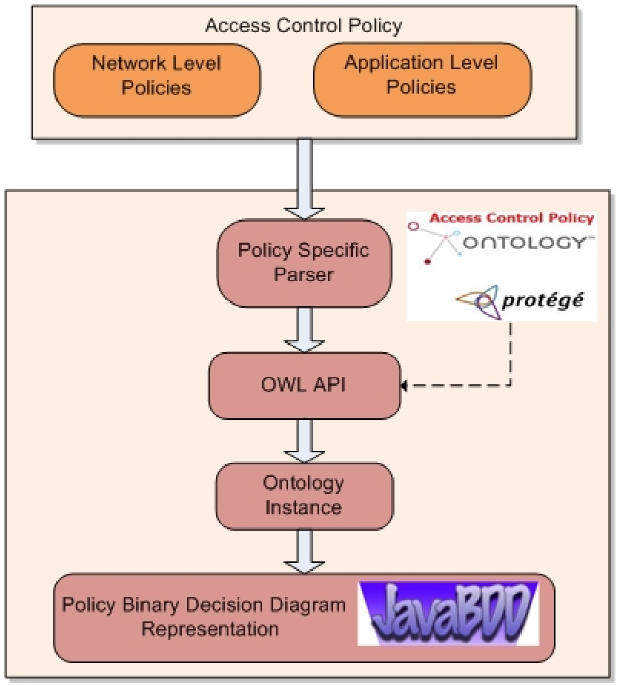


Figure 4.1: Generic Access Control Policy Representation Process

this policy specific ontology instance to an efficient data structure suitable for the required operations. In our implementation, we utilize *binary decision diagrams* (BDD) [20] as an efficient data structure.

Figure 4.1 depicts the system components involved in the process of generic representation of an access control policy. Following subsections describe the details about the realization of each step in a generic policy representation process.

4.1 Generic Ontology for Access Control Policies

Generic representation of policy requires identifying the domain concepts, policy structure and semantics shared by different types of policies. Chapter 3 discussed about the process of capturing these characteristics. However, a format or template is needed which can store this generic information about policy domain concepts, structure and semantics effectively and accurately. Also, this format or template must be easily populated or instantiated with information obtained from a policy in

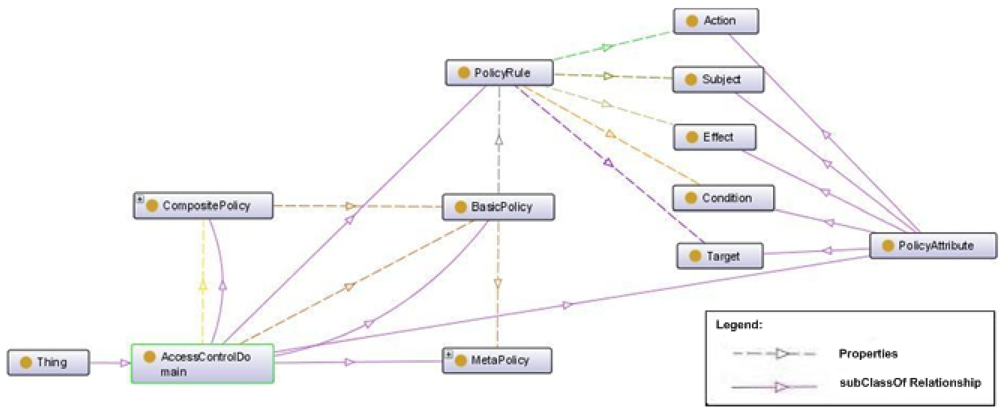


Figure 4.2: Access Control Policy Ontology

order to create generic representation instance for a specific policy. We considered different approaches such as template based information extraction [21]. However, we found them cumbersome and insufficiently expressive for describing the access control policy domain. Then, we came across the method of modeling a domain using an *Ontology*.

Ontology is a model or language for describing the world that consists of a set of types, properties, and relationships. It provides us the shared vocabulary for modeling a domain, which includes the types of concepts in a particular domain, their properties and relationships. Ontology is formally described as follows:

Definition 1 (***Ontology***). *It is a formal representation of knowledge as a set of concepts within a particular domain and relationships between those concepts.*

Creating access control policy ontology enables us to model the access control policy domain by defining its vocabulary, objects and their relations along with the properties. Policy ontology which represents shared domain knowledge, provides us the template which can be instantiated with the information extracted from different types of access control policies to provide a generic policy representation.

Figure 4.2 shows the generic ontology created for the access control policy domain. We used the access control policy domain concepts, structure and semantics obtained from policy specifications as well as hand analysis of a test set of access control policies, to generate the policy ontology. We utilize web ontology language (OWL) [7], which represents the family of knowledge representation languages for authoring an ontology. To create the policy ontology using OWL, we used protege [9] tool. OWL allows us to describe the domain in terms of `individuals`, `classes` and `properties`.

`Class` represents the collection of objects, usually sharing some common properties. Tree hierarchy in the Figure 4.3 shows the classes representing the concepts which describes the *Access Control Domain*. We created the base classes for domain concepts such as *subject*, *action*. `Properties` represents the relationship between individual concepts. There are two main types of OWL properties used to define the relationship:

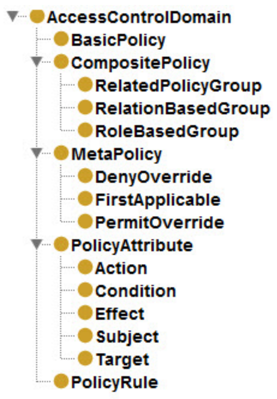


Figure 4.3: Access Control Policy Ontology Concepts

1. *Object Properties*: Object property is a relationship between the two individuals or concepts of ontology domain. We defined the different object properties to capture generic structure as well as semantic information of access control policy. For e.g. object property like *hasRule* between concepts *AccessControlPolicy* and *PolicyRule* captures the structure information that, access

control policy node may contain one or more rules.

2. *Data Properties*: Data properties link an individual concept to its literal value. We used data properties such as *hasValue* to assign a literal value to a concept such as *Subject*.

Defining policy ontology enables us to capture the access control policy domain concepts, structure as well as semantic information. Policy ontology can be easily extended to support new characteristics introduced due to the changes in the access control policy specifications. We can add data properties or object properties along with the new domain concepts to reflect these newly introduced changes in our policy ontology.

4.2 Generating Policy Specific Instance

For creating the generic representation of an access control policy of particular type, we use policy ontology defined for access control policy domain as a template. Module for creating the generic representation of a policy starts with a base policy ontology containing object classes for the concepts of the access control domain and properties as well as relationships between them. We instantiate this base ontology with the structure and semantic information extracted from a particular policy. This method of instantiating the base ontology with the specific attributes, properties and relationships extracted from a policy, is called *Ontology Population*. Basically, we create a copy of the base ontology populated with the attributes, structure and semantic information from a given policy. Figure 4.4 shows the ontology instance representation. Nodes are either labeled with the ontology concept or information instance obtained from a particular policy such as XACML or firewall. A link labeled *instance Of* from an information instance to ontology concept represents an instance generated for the corresponding policy ontology concept.

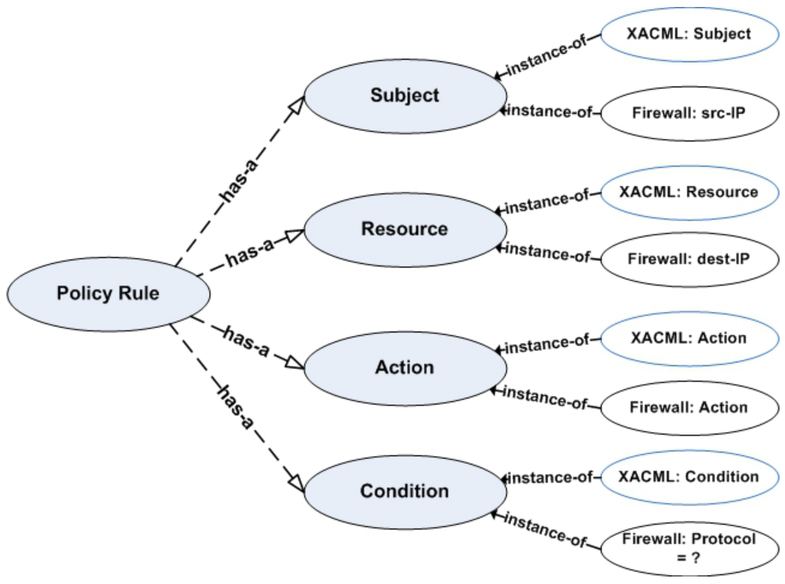


Figure 4.4: Generic Access Control Policy Ontology Instance Representation

We use policy specific parser that extracts the information required for ontology population. Parser understands the semantic as well structure information for a particular policy type and parses out the information required by policy ontology such as *Subject*, *Resource*, *Action* and *PolicyRule*. For example, parser for the firewall policy parses out `source-ip` information in a rule as the *Subject* for that particular rule as per the firewall policy semantics. Process of creation of an ontology instance from example XACML can be described briefly as follows:

Step 1: When XACML parser extracts the information about policy set `RPSlist`, we create the instance of class *PolicyGroup* defined in a base ontology. We also populate the data properties such as *hasId*, *hasStrategy* to assign the policy set *id*, *combining algorithm* information extracted by the parser to a newly created *PolicyGroup* instance.

Step 2: When parser extracts information about policy nodes *P1* and *P2*, we create two instances of class *BasicPolicy* defined in the base ontology. Similarly, instances for policy rules and rule attributes like subject, action, resource are created, which are in turn related to the appropriate policy and rule instances according to the base ontology structure and semantic information. Data properties of each instance such as *hasRuleId*, *hasId* are populated with appropriate information obtained by the parser.

Our current implementation supports the parsers for XACML and firewall policies. Creating the instance for information extracted by a parser is performed using Java based OWL API [14]. It provides the API functions for importing the base ontology, creating the instance for the classes, object properties and data properties defined in the base ontology as well as saving the ontology with newly created instances as ".owl" file in the XML format.

4.3 Generating Efficient Representation

Our policy analysis and management approach uses policy-based partitioning which requires a well-formed representation of policies for performing a variety of set operations. Considering the size and complexity of the real world access control policies, using the ontology instance created in XML format from a given policy is extremely inefficient for performing those set operations. Binary decision diagram (BDD) [20] is a data structure that has been widely used for formal verification and simplification of digital circuits. In our approach, we leverage BDD as the underlying data structure for generic representation of access control policies and facilitate effective policy analysis.

As described in Chapter 3, at the unit level of policy semantics, access control policies are typically based on the policy rules expressed in terms of the attributes of type *Subject*, *Resource*, *Action* and *Effect*. Additionally rules might have conditions, that need to be satisfied for making access control decisions. Given the ontology instance corresponding to a particular policy, we can use `OWLOntologyWalker` and `OWLOntologyWalkerVisitor` provided by OWL API, to walk the asserted structure of the policy ontology. As the *walker* object created using `OWLOntologyWalker` walks over the ontology structure, visitor object created using `OWLOntologyWalkerVisitor` gets visited by the different objects encountered by the *walker*. We can override *visit* method of an `OWLOntologyWalkerVisitor` to obtain the information about required attributes such as *Subject*, *Resource*, *Action* and *Effect* for each policy rule object in an ontology instance. Once these attributes are identified, all policy rule instances can be transformed into boolean expressions [18]. Each boolean expression of a rule is composed of atomic boolean expressions combined by the logical operators $\vee$ and $\wedge$. Atomic boolean expressions are treated as equality constraints or range constraints on attributes (e.g. $Subject = \text{“}Manager\text{”}$) or conditions (e.g. $9:00 \leq Time \leq 17:00$).

Example 1 *Based on the attribute information extracted from an ontology instance of the example XACML policy in Figure 2.2, rule r_3 can be expressed in terms of* atomic boolean expressions *as follows:*

$$(Subject = \text{“}TeamLead\text{”} \vee Subject = \text{“}Associate\text{”} \vee Subject = \text{“}Manager\text{”}) \wedge$$
$$(Resource = \text{“}Goals\text{”} \vee Resource = \text{“}Comments\text{”}) \wedge (Action = \text{“}Read\text{”}) \wedge$$
$$(9:00 \leq Time \leq 17:00)$$

The boolean expression for rule r_5 is:

$$(Subject = \text{“}TeamLead\text{”} \vee Subject = \text{“}Manager\text{”}) \wedge (Resource = \text{“}Goals\text{”}) \wedge$$
$$(Action = \text{“}Assign\text{”} \vee Action = \text{“}Evaluate\text{”} \vee Action = \text{“}Change\text{”})$$

Similarly, some firewall rule from an ontology instance of a firewall policy can be represented in boolean expression as follows:

$$(Subject = \text{“}190.128.72.10 : 80\text{”}) \wedge (Resource = \text{“}10.8.50.115 : 1098\text{”}) \wedge$$
$$(Action = \text{“}Access\text{”}) \wedge (Protocol = \text{”}TCP\text{”})$$

Boolean expressions for policy rules consists of an atomic boolean expressions with an overlapping value ranges. In such cases, those atomic boolean expressions are needed to be transformed into a sequence of new atomic boolean expressions with the disjoint value ranges. Agrawal et al. [15] 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 policy rules.

Table 4.1: Atomic boolean expressions and corresponding boolean variables mapping.

Unique atomic Boolean Expression	**Boolean Variable**
$Subject = \text{“}TeamLead\text{”}$	S_1
$Subject = \text{“}Manager\text{”}$	S_2
$Subject = \text{“}Associate\text{”}$	S_3
$Resource = \text{“}Goals\text{”}$	R_1
$Resource = \text{“}Comments\text{”}$	R_2
$Action = \text{“}Read\text{”}$	A_1
$Action = \text{“}Change\text{”}$	A_2
$Action = \text{“}Assign\text{”}$	A_3
$Action = \text{“}Evaluate\text{”}$	A_4
$9:00 \leq Time < 12:00$	C_1
$15:00 \leq Time < 16:00$	C_2

We encode each of the atomic boolean expression as a boolean variable. For example, an atomic boolean expression

Subject=“Team Lead" is encoded into a boolean variable S_1. A complete list of boolean encoding for the ontology instance of the example XACML policy in Figure 2.2 is shown in the Table 4.1. We then utilize the boolean encoding to construct boolean expressions in terms of boolean variables for the policy rules.

Example 2 *Consider the ontology instance of the example XACML policy in Figure 2.2 in terms of* boolean variables*. The boolean expression for rule r_3 is:*

$$(S1 \vee S2 \vee S3) \wedge (R1 \vee R2) \wedge (A1) \wedge (C1)$$

The boolean expression for rule r_5 is:

$$(S1 \vee S2) \wedge (R1) \wedge (A2 \vee A3 \vee A4)$$

BDD is a acyclic directed graph which represents the 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 the boolean values T (True) or F (False). Figures 4.5(a) and 4.5(b) give BDD representations of two rules r_3 and r_5, respectively.

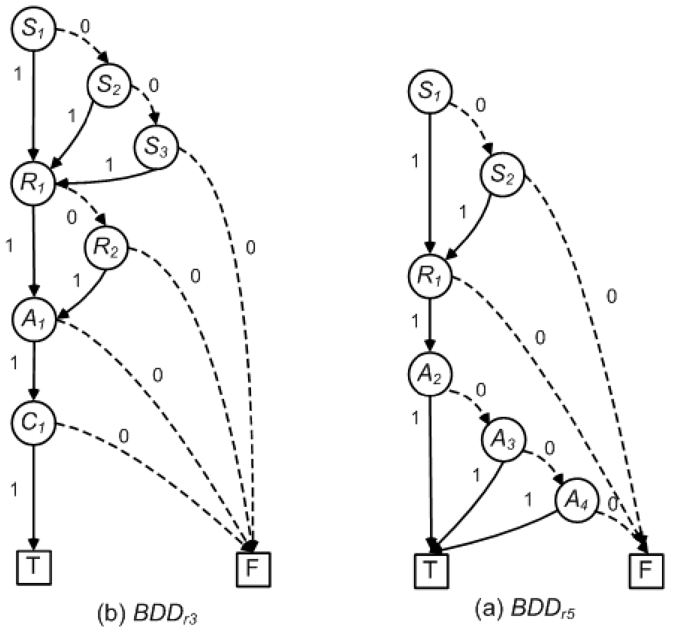


Figure 4.5: Representing the rules of XACML policy ontology instance with BDD.

Once the BDDs are constructed for policy rules, performing the set operations, such as union ($\cup$), intersection ($\cap$) and set difference ($\setminus$), required by our policy-based segmentation algorithms (see Algorithm 1 and Algorithm 2) is efficient as well as straightforward. Our BDD representation is based on boolean expressions generated from the boolean variables mapped to the policy ontology attributes. If new policy attributes are introduced to the generic policy ontology due to changes in policy specifications, then our BDD representation can be easily extended to represent these new attributes from corresponding boolean expressions.

Chapter 5

REALIZATION OF POLICY ANALYSIS AND MANAGEMENT

In the Chapter 4, we discussed about the realizing the generic representation of an access control policies. We used policy ontology to create the instance of given policy type and then convert it to BDD-based policy representation for efficient policy analysis and management. Generic representation of policy gives us the advantage of implementing unified approach for analysis and management for any type of access control policy. This chapter discusses the realization of policy analysis and management module. It consists of two main tasks: (a) Anomaly Detection and (b) Anomaly Resolution. Following subsections describe the implementation approach for analyzing policy anomalies of mainly two types: *policy conflicts* and *policy redundancies*. It also includes the section for the graphical user interface, which is an integral part of the policy analysis and management framework. Figure 5.1 depicts the system components of our policy analysis and management approach.

5.1 Conflict Detection and Resolution

Policy conflicts are the part of policy anomalies which may lead to allowing unauthorized access or denial of legitimate access to a critical resource. To identify the policy conflicts, we define the *authorization space* for each policy component based on BDD representation. We first define the *authorization space* and then determine our conflict detection and resolution approach in following subsections.

Definition 2 *(**Authorization Space**). An authorization space for a policy component can be defined as collection of access requests to which policy component is applicable. Policy component can be considered as policy rule, policy or policy set that collectively define the access control policy.*

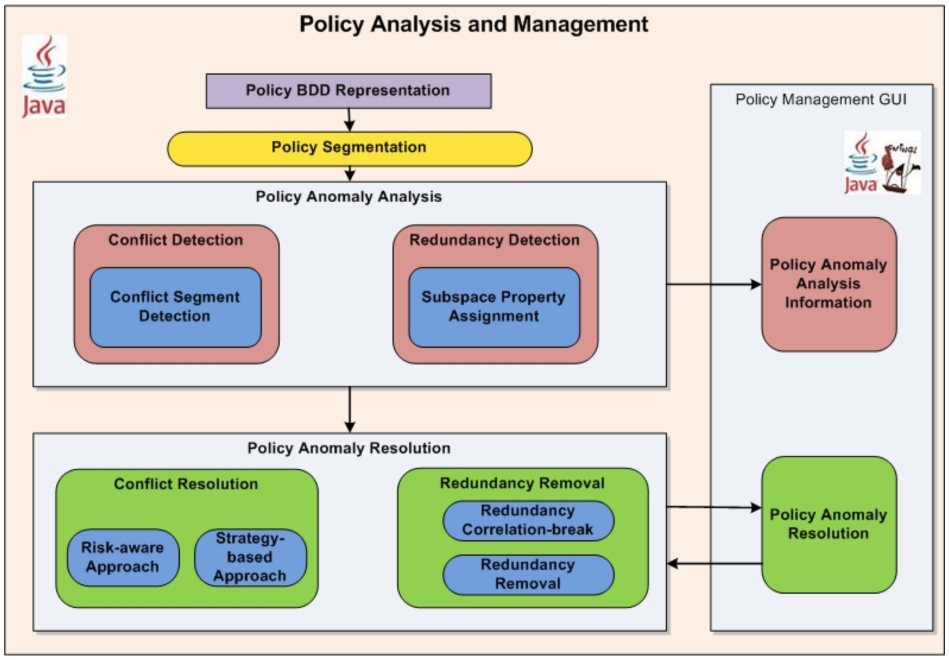

Policy Analysis and Management
Java
Policy BDD Representation
Policy Segmentation
Policy Anomaly Analysis
Conflict Detection
Conflict Segment Detection
Redundancy Detection
Subspace Property Assignment
Policy Management GUI
Java
Policy Anomaly Analysis Information
Policy Anomaly Resolution
Conflict Resolution
Risk-aware Approach
Strategy-based Approach
Redundancy Removal
Redundancy Correlation-break
Redundancy Removal
Policy Anomaly Resolution

Figure 5.1: Generic Analysis and Management Approach for Access Control Policies

5.1.1 Conflict Detection Approach

In our conflict detection approach, we used policy-based segmentation technique [25], in which we partition entire authorization space defined by policy components into disjoint authorization space segments. Then conflicting segments, which contain policy components with different effects, are identified. Each of those conflicting segment represents a policy conflict. We use generic representation of access control policy for policy analysis and management and hence we consider generic structure of policy for conflict detection. For simple access control policies, which are defined using list of rules, we identify conflicts at policy level. While for complex or composite policies, which are defined using more than one policy or group of policies, we identify conflicts at policy as well as policy group level.

Figure 5.2 depicts a tree data structure for generic policy in which the leaf node represents the *policy* node and intermediate nodes on the path from root

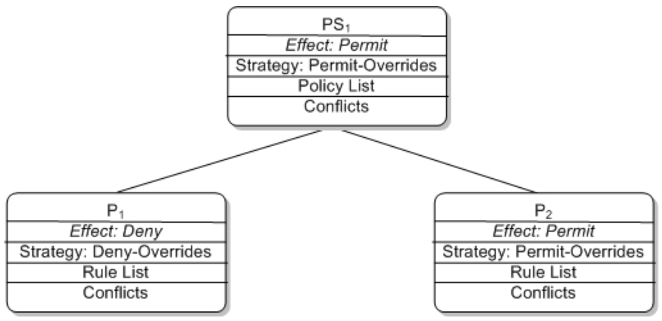


Figure 5.2: Tree structure for example XACML policy

node to leaf node represents a *policy set* or *policy group*. Each node stores the information about the combining algorithm and effect of a policy component. We identify conflicts at each node and store the details in each node to provide the detailed analysis of each policy conflict to the administrator. In case of a simple policy where policy do not have policy groups or hierarchical structure, only one node is present for e.g. firewall policy containing only one ACL. Following subsections discuss the approach and algorithm for detecting conflicts at policy and policy group level.

Conflict Detection at Policy Level

A policy component in an access control policy consists of list of rules. Each rule specifies an authorization space with the effect of either `permit` or `deny`. We call an authorization space with the effect of `permit` as *permitted space* and an authorization space with the effect of `deny` as *denied space*.

Algorithm 1 shows the pseudocode for generating conflicting segments of a policy component P. An entire authorization space derived from a policy component is first partitioned into the set of disjoint segments. As shown in lines 16-32 in Algorithm 1, a method called `Partition` accomplishes this procedure. This method works by adding an authorization space s derived from a rule r to an authorization space list denoted by $partList$. 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, we use the *set* operations to separate the overlapped spaces into disjoint spaces.

Algorithm 1: Identify Disjoint Conflicting Authorization Spaces of Policy

```
    Input: A policy node $PNode$ with a set of rules.
    Output: A set of disjoint conflicting authorization space for $PNode$.
 1  /* Partition the entire authorization space of $PNode$ into disjoint spaces*/
 2  PolicyPartition($PNode$) $partList.New()$;
 3  $ruleList \longleftarrow GetRules(PNode)$;
 4  foreach $r \in ruleList$ do
 5      $s_r \longleftarrow GetAuthorizationSpace(r)$;
 6      $partList \longleftarrow$ Partition($partList, s_r$);
 7  $StorePartitions(PNode, partList)$;
 8  /* Identify the conflicting segments */
 9  $conflictList.New()$;
10  foreach $s \in partList$ do
11      $R^{'} \longleftarrow GetRule(s)$;
12      if $\exists r_i \in R^{'}, r_j \in R^{'}, r_i \neq r_j$ and $r_i.Effect \neq r_j.Effect$ then
13          $conflictList.Append(s)$;
14  $StoreConflicts(PNode, conflictList)$;
15  return $PNode$;
16  Partition($partList, s_r$)
17  foreach $s \in partList$ do
18      /* $s_r$ is a subset of $s$*/
19      if $s_r \subset s$ then
20          $partList.Append(s \setminus s_r)$;
21          $s \longleftarrow s_r$;
22          $Break$;
23      /* $s_r$ is a superset of $s$*/
24      else if $s_r \supset s$ then
25          $s_r \longleftarrow s_r \setminus s$;
26      /* $s_r$ partially matches $s$*/
27      else if $s_r \cap s \neq \emptyset$ then
28          $partList.Append(s \setminus s_r)$;
29          $s \longleftarrow s_r \cap s$;
30          $s_r \longleftarrow s_r \setminus s$;
31  $partList.Append(s_r)$;
32  return $partList$;
```

Conflicting segments are identified as shown in lines 9-13 in Algorithm 1. Figure 5.3 gives representation of the segments of authorization space derived from the policy P_2 of the example policy shown in Figure 2.2 [1]. Five unique disjoint

[1]For the purpose of easy analysis and understandability, we use a two dimensional geometric representation for each authorization space segment. Note that a rule in a policy typically has multiple fields such as subject, action and resource. Thus a complete representation of authorization space must be multi-dimensional.

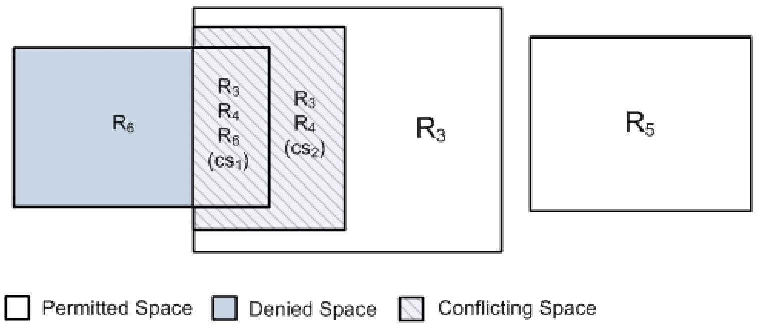


Figure 5.3: Disjoint partition segment for authorization space of policy P_2 in example policy

segments are generated. From five segments, two conflicting segments cs_1 and cs_2 are identified. Each of them represents a policy conflict, where conflicting segment cs_1 is associated with a rule set consisting of three rules r_3, r_4 and r_6, and conflicting segment cs_2 is related to a rule set including two rules r_3 and r_4.

Conflict Detection at Policy Group Level

Our generic approach of policy analysis and management considers conflict detection at a policy group level for the types of policies having hierarchical structure (e.g. XACML) or have notion of policy groups (e.g. Ponder). Algorithm 2 shows the pseudocode for identifying disjoint conflicting authorization spaces of a policy group based on the tree structure of policy as identified in Figure 5.2. In order to partition authorization spaces of all nodes contained in a policy tree, this algorithm recursively calls the partition functions, `PolicyPartition()` and `PolicySetPartition()` to deal with the policy nodes (lines 5-6) and the policy group nodes (lines 9-10), respectively. We used *bottom-to-top* approach to identify policy group conflicts. Once all children nodes of a policy group are partitioned, we 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:

$$\begin{cases} E^P = \bigcup_{s_i \in S_E} s_i & \text{if } Effect(s_i) = Permit \\ E^D = \bigcup_{s_i \in S_E} s_i & \text{if } Effect(s_i) = Deny \end{cases} \tag{5.1}$$

where S_E denotes the set of authorization space segments of the child node E. For aggregating all “`Permit`" segments and “`Deny`" segments, we need to identify an *effect* for each authorization space. For non-conflicting segments, the effect of a segment equals to the effect of components covered by the segment. However, for conflicting segments, effect of segment depends upon the combining algorithm, which is used by the owner (a policy or policy group) of the segment. Different policy types use various combining algorithms (*CA*). For example, Firewall policy use by default *First-applicable*. XACML policy uses four combining algorithms: *First-applicable*, *Permit-override*, *Deny-override* and *Only-one-applicable*. We currently provide the effect generation for conflict segment using above four combining algorithms in following way:

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 to be “`Permit`”.
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.

Our approach is flexible enough to support other combining algorithms for various types of policies as well. In order to generate segments for the policy group node $PSNode$, we can then leverage two subspaces (E^P and E^D) of each child node (E) to partition the existing authorization space set belonging to $PSNode$ (lines 12-13).

Algorithm 2: Identify Disjoint Conflicting Authorization Spaces of Policy Set

```
Input: A policy set $PSNode$ with a set of policies or other policy sets.
Output: A set of disjoint conflicting authorization spaces $CS$ for $PSNode$.
1  /* Partition the entire authorization space of $PSNode$ into disjoint spaces*/
   PolicySetPartition($PSNode$) $psParts.New()$;
2  $C \longleftarrow GetChildNodes(PSNode)$;
3  foreach $c \in C$ do
4      /* c is a policy*/
5      if $IsPolicy(c) = true$ then
6          $c \longleftarrow$ PolicyPartition($c$);
7          $AggregatePermitDenySpace(c)$;
8      /* c is a policy set*/
9      else if $IsPolicySet(c) = true$ then
10         $c \longleftarrow$ PolicySetPartition($c$)
11 $childNodeParts \longleftarrow$ GetChildNodePartitions($PSNode$);
12 foreach $childPart \in childNodeParts$ do
13     $psParts \longleftarrow$ Partition($psParts, childPart$);
14 $StorePartitions(PSNode, psParts)$;
15 $AggregatePermitDenySpace(PSNode)$;
16 /* Identify the conflicting segments */ $CS.New()$;
17 foreach $s \in psParts$ do
18     $E \longleftarrow GetElement(s)$;
19     if $\exists e_i \in E, e_j \in E, e_i \neq e_j$ and $e_i.Effect \neq e_j.Effect$ then
20         $CS.Append(s)$;
21 $StoreConflicts(PSNode, CS)$;
22 return $psNode$;
23 AggregatePermitDenySpace($node$)
24 $S^{'} \longleftarrow GetNodePartitions(node)$;
25 $E^P.New()$;
26 $E^D.New()$;
27 foreach $s^{'} \in S^{'}$ do
28     if Effect($s^{'}$) = Permit then
29         $E^P \longleftarrow E^P \cup s^{'}$;
30     else if Effect($s^{'}$) = Deny then
31         $E^D \longleftarrow E^D \cup s^{'}$;
32 $StorePermitSpace(node, E^P)$;
33 $StoreDenySpace(node, E^D)$;
34 return $node$;
```

Figure 5.4 represents an example of the segments of authorization spaces derived from policy set PS_1 in our example policy (Figure 2.2). We can observe that six unique disjoint segments are generated, and one of them cs_1 is a conflicting segment. cs_1 is related to P_1^D and P_2^P. It indicates a conflict occurring at policy set level.

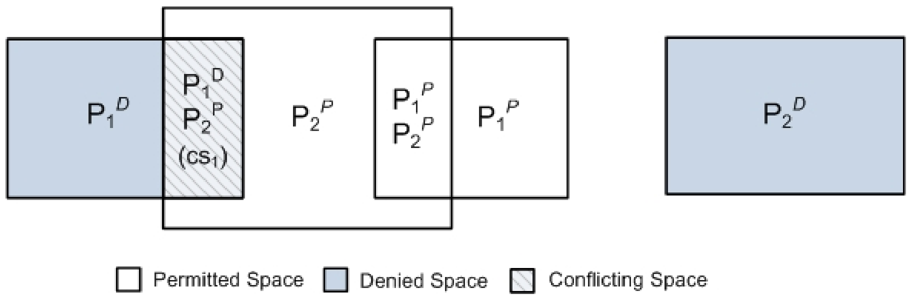


Figure 5.4: Disjoint partition segment for authorization space of policy P_2 in example policy

5.1.2 Conflict Resolution Approach

As discussed previously, conflicts always exist in access control policies. Every access control policy specification defines the mechanism to resolve the conflict. For example, firewall policy deals with conflicts by using default *First-match* mechanism. XACML policy specification provides four combining algorithms: *Permit-override, Deny-override, First-applicable* and *Only-one-applicable*. Once conflicts are identified between policy components using our approach, administrator can choose appropriate strategies according to policy specifications to resolve them. However, conflict resolution strategies or mechanisms provided by existing policy specifications are too restrictive and allow policy administrator to choose only one combining algorithm for all conflicts identified within policy. It is common case that administrator might want to resolve each component individually using different resolution strategies. Many conflict resolution strategies exists [23, 26, 27], but cannot be used due to restrictions of policy specifications. Our approach to conflict resolution provides comprehensive conflict resolution framework as shown in Figure 5.5. Our conflict resolution approach has two steps: (A) Effect Constraint Generation (B) Conflict Resolution based on Effect Constraints. Following subsections describe each step in detail.

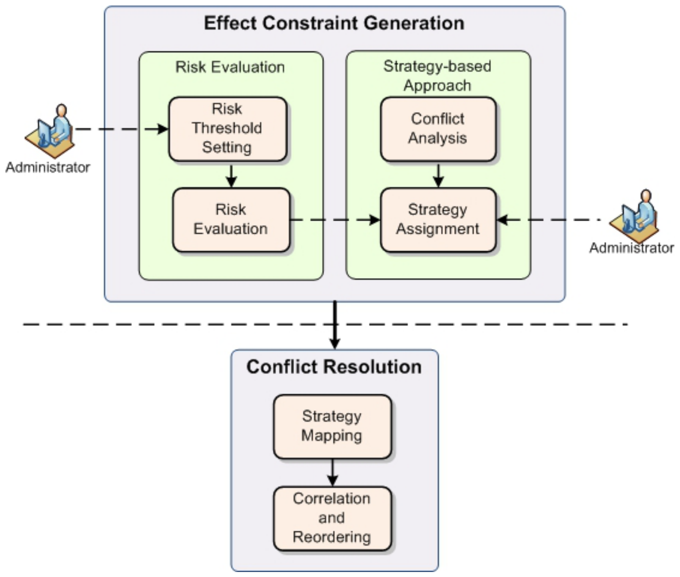


Figure 5.5: Conflict Resolution Framework

Step A. Effect Constraint Generation

In order to resolve the policy conflicts, policy components involved in conflict should take expected action for any access request within the conflicting authorization segment. In order to decide expected action, policy administrator needs to assign desirable effect constraint for conflict segment. *Effect Constraint* for conflict segment can be formally defined as follows:

Definition 3 (***Effect Constraint***). *An effect constraint for a conflicting segment defines an expected action that the policy should take when any authorization request within the conflicting segment comes to the policy.*

To generate effect constraints for conflict segments within policy, we use *Strategy-based* approach described below.

Strategy-based approach In this approach, an effect constraint is derived from the conflict resolution strategy associated with the conflicting segment. A policy administrator chooses an appropriate conflict resolution strategy for each identified conflict by examining the features of conflicting segment and policy components involved. In our conflict resolution framework, a policy administrator is able to adopt different strategies to resolve the conflicts indicated by different conflicting segments. In addition to standard conflict resolution strategies provided by different policy specifications, user-defined strategies [27], described below can be used:

- *Recency-Overrides*: This strategy indicates that in case of policy conflicts the recency of policy components involved in a conflict is checked. The decision provided by more recently introduced policy component takes precedence over other components involved in a conflict.
- *High-Majority-Overrides*: This strategy permits (or denies) a request if the number of policy components taking "Permit" (or "Deny") action is greater than the number of policy components taking "Deny" (or "Permit") action respectively.
- *High-Authority-Overrides*: This strategy states that a policy component defined by a policy administrator with the highest authority takes precedence.

In addition, we also articulate another resolution strategy with the notion of the *risk*. This strategy can be applied to the network security policies such as firewall policy. A basic idea of using risk-aware strategy for effect constraint generation is that a risk level of a conflicting segment could be used to determine the expected action taken for the access requests within a conflicting segment. Risk evaluation can be performed using specific tools. For example in case of network level policies, we can use Common Vulnerability Scoring System (CVSS) [29, 30] as an underlying security metrics for the risk evaluation. Network venerability scanning tools such as Nessus [13] and Qualys [10] adopt CVSS *base score* to provide crucial baseline information for automated security analysis.

In conventional risk measurement, two major factors contribute to the calculation of *risk value* (R_v). One factor is *probability of damage* (P_d), which indicates the chance that an event happens to incur the damage. Another is *cost of damage* (C_d), which is a quantified measurement of the damage. Beside those two factors, another important factor in determining the criticality of an identified security problem is the *asset importance value* (A_v). Normally, policy administrators place a higher priority on protecting the critical resources over non-critical once. Equation 5.2 shows the general method for *Risk Value* measurement.

$$\textit{Risk Value} = P_d \times C_d \times A_v \tag{5.2}$$

To calculate the *risk level* $RL(cs)$ of each conflicting segment, we use *average* of all risk values for the policy components covered by a conflicting segment using the equation 5.3.

$$RL(cs) = \frac{\sum_{R_v}}{|C(cs)|} \tag{5.3}$$

Where, $C(cs)$ is a function to return all policy components involved in a conflicting segment cs. Once the risk level of a conflicting segment is computed, an action constraint can be generated based on the risk level and a threshold (*TH*), which is often set by a policy administrator.

$$Action(cs_i) = \begin{cases} \text{“}allow\text{”} & \text{if } RL(cs_i) < TH. \\ \text{“}deny\text{”} & \text{if } RL(cs_i) >= TH; \end{cases} \tag{5.4}$$

Essentially, the threshold represents a tradeoff between security and availability as shown in Figure 5.6. We assume the range of risk level is from 0.0 to 10.0 [2]. We can see, if the choice of the threshold is less than 5.0, more conflicting

[2]For simplicity, we will just adopt *average* risk value (from 0.0 to 10.0) as the risk level of a conflict segment for the demonstration of our approach in the rest of this document.

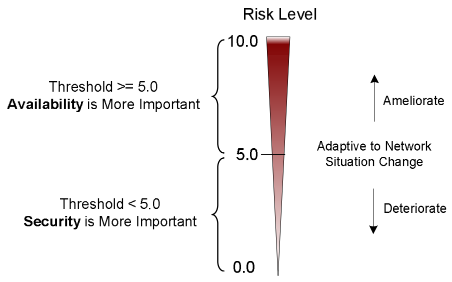

Risk Level
10.0
Threshold >= 5.0
Availability is More Important
5.0
Threshold < 5.0
Security is More Important
0.0
Ameliorate
Adaptive to Network Situation Change
Deteriorate

Figure 5.6: Tradeoff between security and availability using threshold.

segments will be assigned a *deny* action constraints. That means more access requests will be blocked due to the reason that the policy administrator considers securing the system is more important than system availability. On the other hand, if the policy administrator selects a large number for threshold ($>= 5.0$), more *permit* action constraints will be generated for the conflicting segments, which means more access requests can access system resources.

Risk-aware approach provides automatic constraint generation for conflict segments based on the risk evaluation. It does not require policy administrators intervention once risk level threshold is set. In future, additional strategies can be easily supported by our conflict resolution framework for effect constraint generation.

Step B. Conflict Resolution Based On Effect Constraint

Once *effect constraints* are generated for conflict segments, it is important to map them to the existing conflict resolution strategies determined by the policy specifications. In our framework, conflict resolution strategies assigned to resolve different conflicts by a policy administrator can be *automatically* mapped to the standard conflict resolution strategies defined in policy specifications without changing the way current policy implementations perform. For example, XACML policies provides *permit-override, deny-override* and *first-applicable*. If all effect constraints are

“Permit" for all conflict segments, then *permit-override* is selected for the target policy component in XACML policy. Similarly, *deny-override* is assigned to the target policy component, if all effect constraints are “Deny". In other cases, *first-applicable* is selected as conflict resolution strategy. Most of the access control policies support basic conflict resolution mechanism like *first-applicable*. However, in order to resolve all conflicts within the target component by applying *first-Applicable*, the process of reordering the conflicting components is compulsory. Process of reordering the conflicting components makes sure that first component applied to access request is the correct component as intended by the policy administrator.

Practically, one policy component may get involved in multiple conflicts. The most ideal solution for conflict resolution by using *first-applicable* strategy is that all action constraints for conflicting segments can be satisfied by reordering conflicting components. In other words, if we can find out an ordering of conflicting components that satisfies all action constraints, this ordering must be the optimal solution for conflict resolution. Unfortunately, in practice action constraints for conflicting segments can only be satisfied partially in some cases. It also implies that, we cannot resolve the conflicts by reordering policy components according to *first-applicable* strategy for each conflict segment individually. Thus we use conflict correlation mechanism proposed by Hongxin et al. [25] to identify dependent relationships among conflict segments. Algorithm 3 shows the process of identifying correlated conflict segments.

Figure 5.7 shows an example for conflicting segment correlation, considering an general policy component P with eight rules. Five conflicting segments are identified in this example. Several rules in this policy component are involved in multiple conflicts. For example, r_2 contributes to the two policy conflicts corresponding to the 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

Algorithm 3: Conflicting Segment Correlation Algorithm.

```
   Input: A set of conflicting segments, $C$.
   Output: A set of groups for correlated segment, $GRP$.
1  $GRP.New()$;
2  foreach $c \in C$ do
3      $PCList \longleftarrow GetComponents(c)$;
4      foreach $g \in GRP$ do
5          foreach $c^{'} \in GetSegment(g)$ do
6              $PCList^{'}.Append(GetComponents(c^{'}))$;
7          if $PCList \cap PCList^{'} \neq \varnothing$ then
8              $g.Append(c)$;
9          else
10             $GRP.NewGroup().Append(c)$;
11 return $GRP$;
```

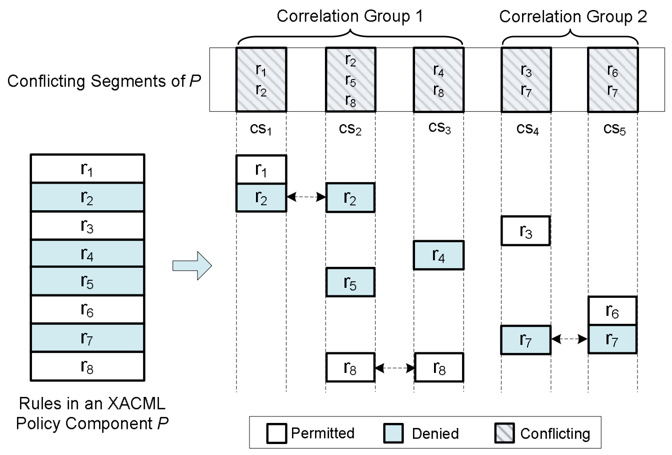


Figure 5.7: Conflicting segment correlation Example.

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*.

Once we identify conflict correlation group, we need to identify ordering of policy components involved in a correlated group that satisfies all effect constraints generated for conflict segments. However as discussed above, all effect constraints cannot be satisfied always. Hence, naive way to find optimal solution is to exhaustively list all permutations of correlated conflicting components. Total *resolve score* is computed for each permutation, which indicates total number of conflicts resolved by particular permutation. Permutation with maximum *resolve score* is selected as near optimal or optimal solution. Resolving score can be computed as shown in equation 5.5.

$$RS(pm) = |CS.Sat(pm)| \tag{5.5}$$

Note that we use a function *Sat()* in this equation to identify whether a permutation can satisfy the effect constraint of a conflicting segment. Implementation of *Sat()* function is fairly straight forward, which checks whether the effect of a first component in particular permutation satisfies the effect constraint for conflict segment or not.

5.2 Redundancy Detection and Removal

We use policy-based segmentation technique in our redundancy removal approach. Hongxin et al. [25] have introduced redundancy detection and removal mechanism for XACML policies. We try to generalize their approach for policy analysis and management. We define the *redundancy* in policy as follows:

Definition 4 *(**Redundancy**). A component c is redundant in a policy p if and only if the authorization space derived from the resulting policy p' after removing c is equivalent to the authorization space defined by p.*

We use three steps to identify and eliminate the redundancies: authorization space segmentation, property assignment for subspaces of segments and correlation break and redundancy removal. We describe each step in details below.

Step 1. Authorization Space Segmentation

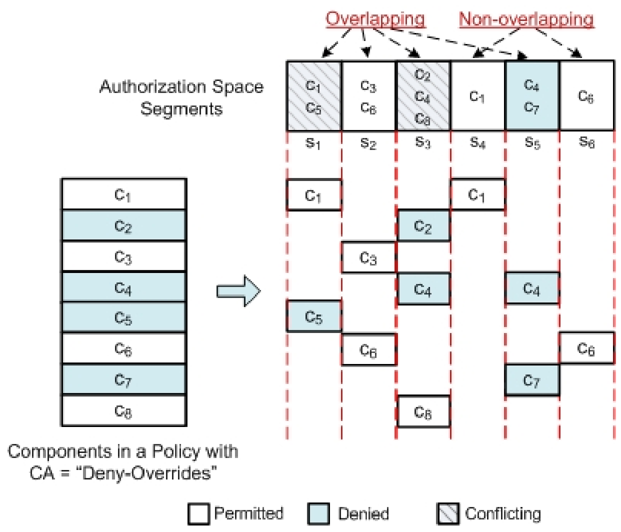


Figure 5.8: Authorization Space Segments Classification.

In this step, we use authorization space segments created for policy level conflict detection. We identify *overlapping, non-overlapping* and *conflicting overlapping* segments. Each non-overlapping segment is associated with a unique component and each overlapping segment is related to a set of components, which may conflict with each other (conflicting overlapping segment) or may have same effect (overlapping segment). Figure 5.8 shows example authorization segments classified in above categories.

Figure 5.8 illustrates an authorization space segmentation for a policy with eight components. In this example, two policy segments s_4 and s_6 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.

Step 2. Property Assignment

To reflect different characteristics of each subspace in authorization segments, we assign four property values: *removable* (R), *strong irremovable* (SI), *weak irremovable* (WI) and *correlated* (C). *Removable* property indicates that a rule subspace is removable. In other words, removing such a component subspace does not make any impact on the original authorization space of an associated policy. *Strong irremovable* property means that a component subspace cannot be removed because the effect of corresponding policy segment can be only decided by this component. *Weak irremovable* property is assigned to a component subspace when any subspace belonging to the same component has *strong irremovable* property. That means a component subspace becomes irremovable due to the reason that other portions of this component cannot be removed. *Correlated* property is assigned to multiple component subspaces covered by a policy segment, if the effect of this policy segment can be determined by any of these components. Following are the rules for the property assignment to each subspace within segments of a policy.

Rule 1: *Property assignment for subspaces of a non-overlapping segment*. A non-overlapping segment contains only one component subspace. Thus, this subspace is assigned with the *strong irremovable* property. Other component subspaces associated with the same component are assigned with the *weak irremovable* property, except the component subspaces that already have the *strong irremovable* property.

Rule 2: *Property assignment for subspaces of a conflicting segment*. Property assignment for this component is based on the combination algorithm used by the owner component of this segment. Combination algorithms differ with each policy type. Currently we support combination algorithms used by XACML and firewall policies. We describe the behavior of the property assignment module for *First-applicable, Permit-override* and *Deny-override* combination algorithms below. However, we need to extend behavior of this module to support new combination algorithms that may be introduced by the new policy types.

1. *First-Applicable*: In this case, the first component subspace covered by the conflicting segment is assigned with the *strong irremovable* property. Other component subspaces in the same segment are assigned with the *removable* property. Meanwhile, other subspaces associated with the same component are assigned with the *weak irremovable* property except the subspaces already having *strong irremovable* property.
2. *Permit-Overrides*: All subspaces of “`deny`" components in this conflicting segment are assigned with the *removable* property. If there is only one “`permit`" subspace, this case is handled similar to the *First-Applicable* case. If any “`permit`" subspace has been assigned with the *weak irremovable* property, other subspaces without *irremovable* property are assigned with the *removable* property. Otherwise, all “`permit`" subspaces are assigned with the *correlated* property.
3. *Deny-Overrides*: This case works the same as for the *Permit-Overrides* case.

Rule 3: *Property assignment for subspaces of non-conflicting overlapping segment*. If any component subspace has been assigned with *weak irremovable* property, other subspaces without *irremovable* property are assigned with the *removable* property. Otherwise, all subspaces within the segment are assigned with the *correlated* property.

Figure 5.9 shows the result of applying our property assignment mechanism, which performs three step property assignment process in the sequence to the example presented in Figure 5.8. We can easily identify that r_3 and r_8 are *removable* rules whose all subspaces have *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.

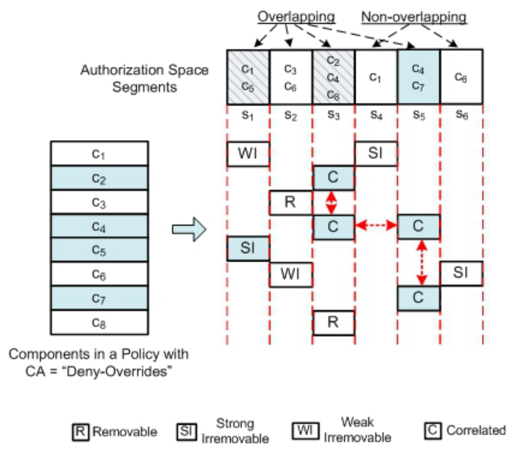


Figure 5.9: Property Assignment.

Step 3. Correlation Break and Redundancy Removal

Subspaces covered by an overlapping segment are correlated with each other when the effect of overlapping segment can be determined by any one of those correlated subspaces. Thus, keeping one correlated subspace and removing others may not change the effect of corresponding segment. We call such a correlated relation as *vertical correlation*, which can be identified in the property assignment step. For example in Figure 5.9, within segment s_3, component subspace of c_4 is in a *vertical correlation* with the component subspace of c_2. In addition, we observe that some components may be involved in several conflicts. For example in Figure 5.9, c_4 has two subspaces that are involved in the correlated relations with c_2 and c_7, respectively. This correlated relation is called as *horizontal correlation*. This case is similar to conflict resolution reordering process where we cannot resolve the correlation individually. We use similar approach of identifying component correlation groups based on the *vertical* and *horizontal* correlations. Figure 5.10(a) shows correlated relationships of component subspaces of c_2, c_4 and c_7 in the correlation group g.

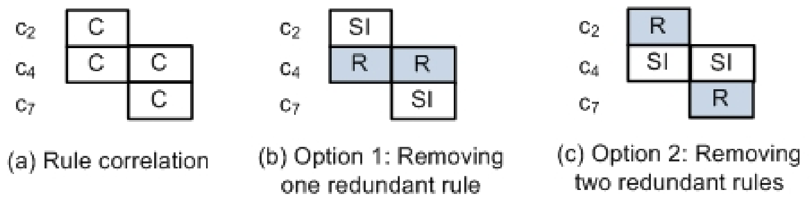

c2
c4
c7
C
C
C
C
(a) Rule correlation
SI
R
R
SI
(b) Option 1: Removing one redundant rule
R
SI
SI
R
(c) Option 2: Removing two redundant rules

Figure 5.10: Example of Correlation Breaking Process for Redundancy Removal.

We additionally observe that different property assignments to break the rule correlations in a correlation group may lead to different results for redundancy removal. Figure 5.10(b) shows one possible solution. If we first assign two subspaces of c_4 with *removable* property, c_4 becomes a *removable* but c_2 and c_7 are turned to an *irremovable* components. In this case only one redundant component can be removed. Consider another solution as shown in Figure 5.10(c), if we first assign the correlated subspace of c_2 with *removable* property, then only c_4 becomes an *irremovable* component. Both c_2 and c_7 are *removable*. Thus, in this case, we can remove two redundant components. Based on this example, we observe that it is necessary to seek optimal solution in order to achieve maximum redundancy removal. To achieve this goal, we can compute a correlation degree (CD) for each correlated component c using the equation 5.6.

$$CD(c) = \sum_{s_i \in CS(c)} \frac{1}{NC(s_i) - 1} \tag{5.6}$$

Note that $CS(c)$ is a function to return all correlated segments of a component c, and $NC(s_i)$ is a function to return the number of correlated components within a segment s_i. Since each policy segment contains multiple correlated components ($NC(s_i) \geq 2$), $\frac{1}{NC(s_i)-1}$ gives the degree of breakable correlation relations associated with a policy segment s_i if we set a component c as *removable*. To maximize the number of removable components for redundancy resolution, our correlation break process selects one component with the minimal CD as the candidate removable component each time. For example, applying this equation to calculate

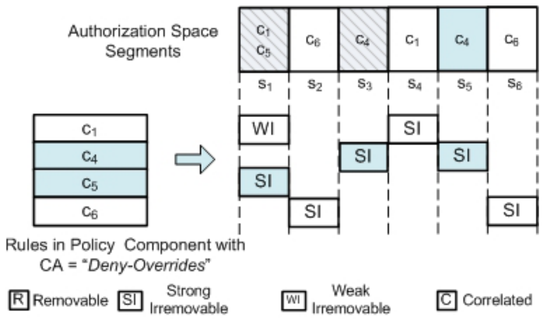


Figure 5.11: Redundancy Removal.

correlation degrees of three components demonstrated in Figure 5.10(a), $CD(c_2)$ and $CD(c_7)$ equals to 1, and $CD(c_4)$ equals to 2. Thus, we can select either c_2 or c_7 as the candidate removable component in the first break step. Finally, two components c_2 and c_7 become removable after breaking all correlations. Applying this correlation break results to given example in Figure 5.9, we observe that four components c_2, c_3, c_7 and c_8 are identified as redundant components.

Figure 5.11 depicts the result of applying our redundancy removal approach to the example given in Figure 5.8.

We identified the generic approach for redundancy identification and removal from policy components. For complex policies having hierarchical structure, it is not sufficient to identify redundancies at policy level. We also need to identify redundancies at policy set or policy group level. Following two subsections describe the redundancy removal at policy as well as policy set level.

5.2.1 Redundancy Removal at Policy Level

To remove the redundancies at policy level, we use the same approach as described above. The redundant components at policy level can be identified in terms of policy rules. Algorithm 4 describes the steps involved in the process of redundancy removal at policy level.

```
Algorithm 4: Redundancy Elimination at Policy Level
    Input: A policy $P$ with a set of rules.
    Output: A redundancy-eliminated policy $P^{'}$.
1   PolicyRedundancyRemoval($P$) /* Partition the entire authorization space of $P$ into disjoint spaces*/
2   $S.New()$;
3   $S \longleftarrow$ PolicyPartition($P$);
4   /* Property assignment for all rule subspaces */
5   PropertyAssgin_P($S$);
6   /* Rule correlation break */
7   $G \longleftarrow$ IdentifyCorrelatonGroup($S$);
8   foreach $g \in G$ do
9       foreach $r \in g$ do
10          $r.CD \longleftarrow \sum_{s_i \in CS(r)} \frac{1}{NC(s_i)-1}$;
11      $SP \longleftarrow GetCorrelatedSubspace(MinCDRule(g))$
12      foreach $sp \in SP$ do
13          $sp.Property \longleftarrow$ R ;
14          if $|GetCorrelatedSubspace(sp)| = 1$ then
15              $SP^{'} \longleftarrow GetCorrelatedSubspace(sp)$ $SP^{'}.Property \longleftarrow$ SI ;
16  /*Redundancy removal */
17  $P^{'} \longleftarrow P$;
18  foreach $r \in P^{'}$ do
19      if $AllRemovalProperty(r) =$ true then
20          $P^{'} \longleftarrow P^{'} \setminus r$;
21  return $P^{'}$;
```

5.2.2 Redundancy Removal at Policy Group Level

We use policy tree representation as shown in Figure 5.2 for elimination of redundancies at policy set level. We remove the redundancies using bottom-to-top approach. Redundancies are removed at policy level first using method described above and then redundancies at policy set level are removed recursively.

For redundancy removal at policy set level, both redundancies among children nodes as well as rule 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. Note that, we do not use aggregated authorization spaces as in the case of conflict detection at policy set level. Figure 5.12 demonstrates an example of authorization space segmentation of a policy set PS with three children

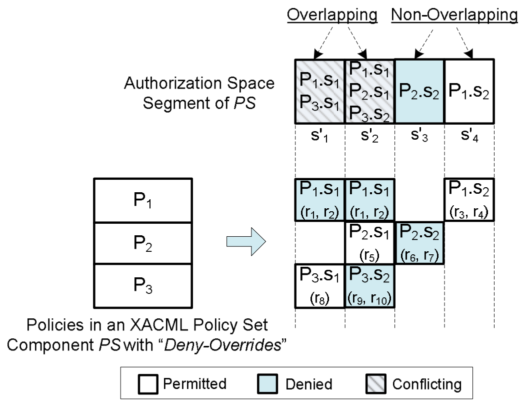


Figure 5.12: Authorization space segmentation at Policy set level for redundancy removal.

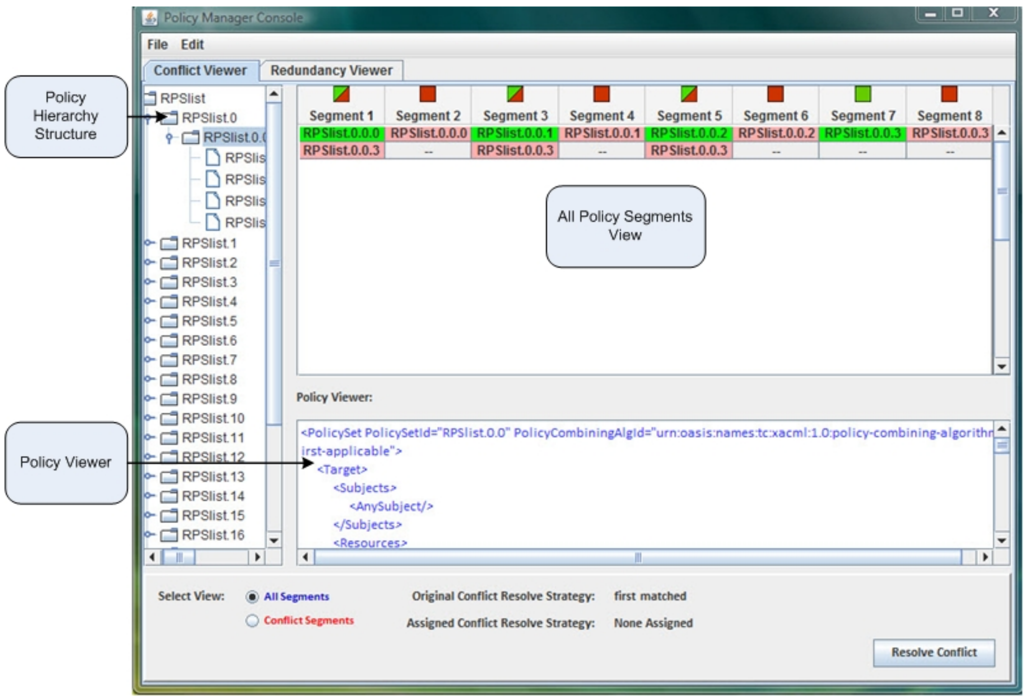
components P_1, P_2 and P_3. The authorization space segments of PS are constructed based on the original segments of each child component. For instance, a segment $s_2^{'}$ of PS covers three policy segments $P_1.s_1$, $P_2.s_1$ and $P_3.s_2$, where $P_x.s_y$ denotes a segment s_y belonging to a policy P_x.

After the property assignment to each component subspace of policy set segments, we 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. Next, 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 for the same rule with dependent *weak irremovable* property need to be changed to *removable*.

5.3 Graphical User Interface

Considering the complexity of tasks involved in the policy anomaly detection and resolution, it is necessary to provide intuitive user interface for policy administrators. Careful study of interactions of policy administrators with policy management tasks provided us with the information about the workflow of policy analysis and management process as well as information required by the administrator at each step. Based on our observation, we designed visualization interfaces for policy analysis and management tasks. Each visualization interface provides administrator with the accurate as well as precise information required for each analysis and management task. It helps in reducing the ambiguity as well as complexity of information that needs to be analyzed by the administrator for performing a particular task. In this section, we describe each visualization interface that constitutes graphical user interface of our implemented tool for policy analysis and management.

Policy analyzer interface of our tool depicts hierarchical structure of policies and allows administrator to view anomalies at different levels of a policy independently. Figure 5.13 shows the policy analyzer interface. Hierarchical structure of policies is depicted by the tree of policy components present on the left side. Administrator can choose a particular policy component such as *Policy* or *PolicySet* node for anomaly analysis. We provide two different tabs for analyzing two types of anomalies: (a)*Policy Conflict* and (b) *Policy Redundancy*. If administrator chooses a *Policy* node for the analysis, then conflicts in that particular *Policy* node are displayed in terms of the rule subspaces involved. Otherwise, if administrator chooses a *PolicySet* or *Policy Group* node, then all conflicts within that particular node are displayed in terms of permit and deny subspaces of policy or policy set components. The *columns* in the right hand space of policy analyzer interface depicts the disjoint segments generated for particular node as a result of policy-based segmentation. Each row of the column represents policy component subspace involved in the disjoint segment and is identified by a component id. Color of each subspace represents the effect of each policy component; *Green* identifies *permit* subspace and *Red* depicts *Deny* subspace. Further, administrator either chooses to visualize all the segments generated for selected policy component or to analyze only



Segment 1	Segment 2	Segment 3	Segment 4	Segment 5	Segment 6	Segment 7	Segment 8
RPSlist.0.0.0	RPSlist.0.0.0	RPSlist.0.0.1	RPSlist.0.0.1	RPSlist.0.0.2	RPSlist.0.0.2	RPSlist.0.0.3	RPSlist.0.0.3
RPSlist.0.0.3	--	RPSlist.0.0.3	--	RPSlist.0.0.3	--	--	--

Policy Viewer:

```
<PolicySet PolicySetId="RPSlist.0.0" PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithn
irst-applicable">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
```

Select View: [✓ All Segments] Conflict Segments

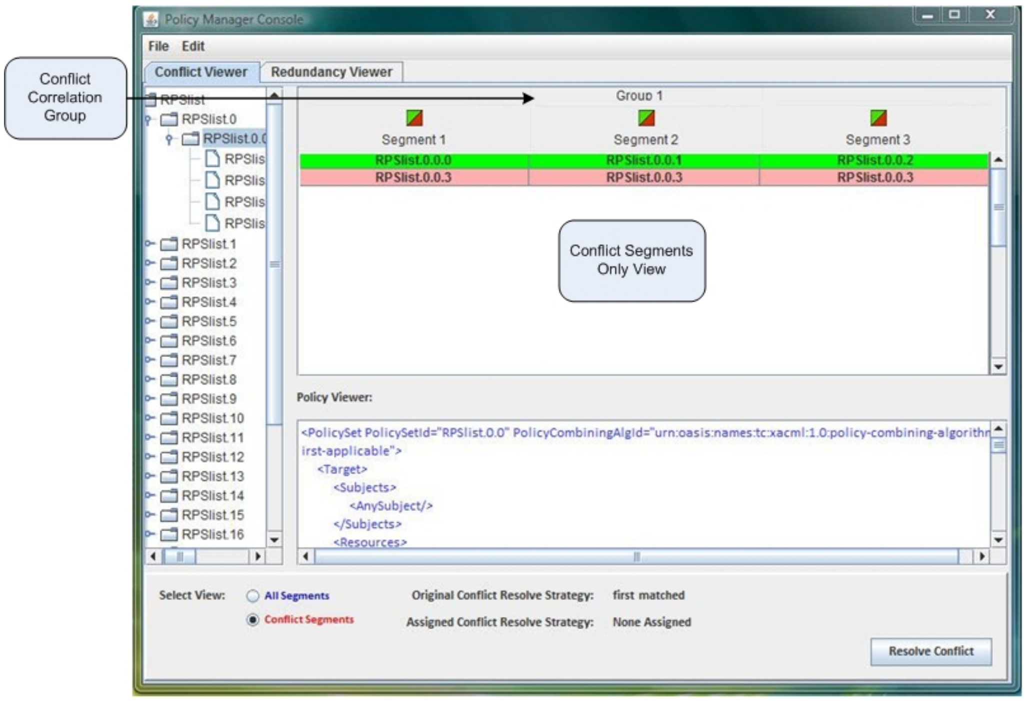
Original Conflict Resolve Strategy: first matched

Assigned Conflict Resolve Strategy: None Assigned

Figure 5.13: Policy Analyzer Interface - All Segments View

conflicting segments as shown in Figure 5.14. This can be done by choosing appropriate radio buttons provided below the policy analyzer interface. Conflicting segments are grouped according to the correlated conflicting segments for easy conflict analysis. In our interface, icons for policy segments indicate three different states with respect to conflict analysis and resolution. One icon with *Green Square* represents *Permit* effect constraint generated for a segment, while other icon with *Red Square* represents *Deny* effect constraint assigned to a particular segment. Third icon with *Half Red and Half Green Square* represents a conflicting segment.

Once administrator clicks on a particular column header representing a policy segment or correlated conflicting group, detailed information about all policy components involved in particular segment or correlated group is displayed in another window as shown in Figure 5.15. Details include information such as risk associated with each component, the date when component was introduced into the policy and by which authority. Administrator analyzes all the information and



Segment 1	Segment 2	Segment 3
RPSlist.0.0.0	RPSlist.0.0.1	RPSlist.0.0.2
RPSlist.0.0.3	RPSlist.0.0.3	RPSlist.0.0.3

Policy Viewer:

```
<PolicySet PolicySetId="RPSlist.0.0" PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithn
irst-applicable">
  <Target>
    <Subjects>
      <AnySubject/>
    </Subjects>
    <Resources>
```

Select View: ( ) All Segments [✓ Conflict Segments]

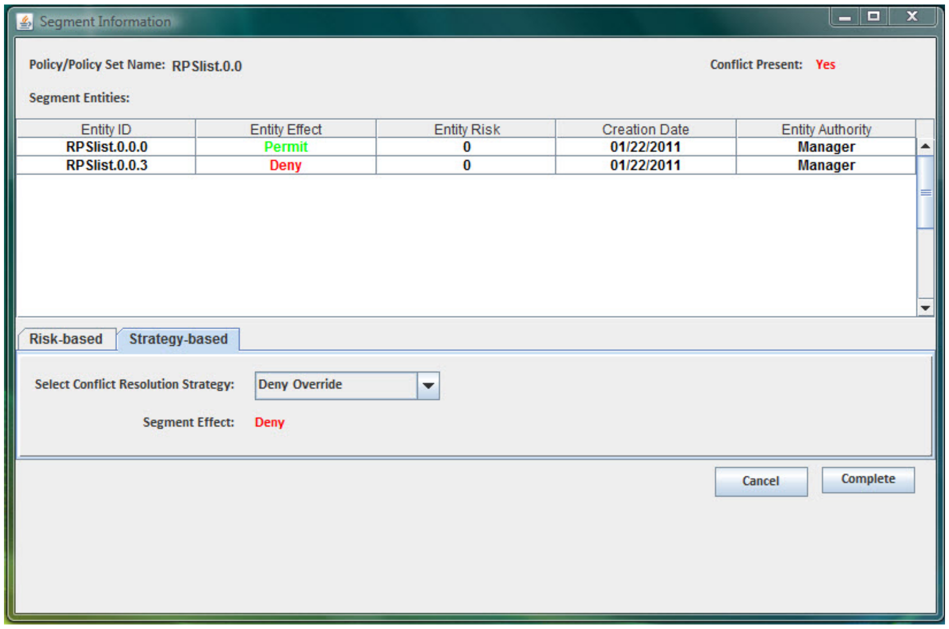
Original Conflict Resolve Strategy: first matched

Assigned Conflict Resolve Strategy: None Assigned

Figure 5.14: Policy Analyzer Interface - Conflict Segments Only View

chooses an *effect constraint* for this conflicting segment based on either *risk* analysis for network security policies or *strategy-based* approach. In case of *risk-aware* approach, administrator analyzes average risk for this conflicting segment and specify the *risk-level threshold* in the tab below. Generated effect-constraint is shown to the administrator. In case of *strategy-based* approach, administrator chooses appropriate strategy based on information provided for each component involved in a conflict segment through the corresponding tab.

After generating effect constraints for each conflict segment, administrator chooses to resolve all conflicts by reordering of policy components and assigning appropriate *combining algorithm* or *conflict resolution strategy* with a particular policy component. This process is initiated by clicking *Resolve* button in the below right corner of policy analyzer interface. Reordering is reflected in the policy analyzer interface and assigned *conflict resolution strategy* is displayed. Policy administrator can also visualize the actual policy component definition in the *Policy Viewer* based



Policy/Policy Set Name: RPSlist.0.0

Conflict Present: Yes

Segment Entities:

Entity ID	Entity Effect	Entity Risk	Creation Date	Entity Authority
RPSlist.0.0.0	Permit	0	01/22/2011	Manager
RPSlist.0.0.3	Deny	0	01/22/2011	Manager

Select Conflict Resolution Strategy: Deny Override

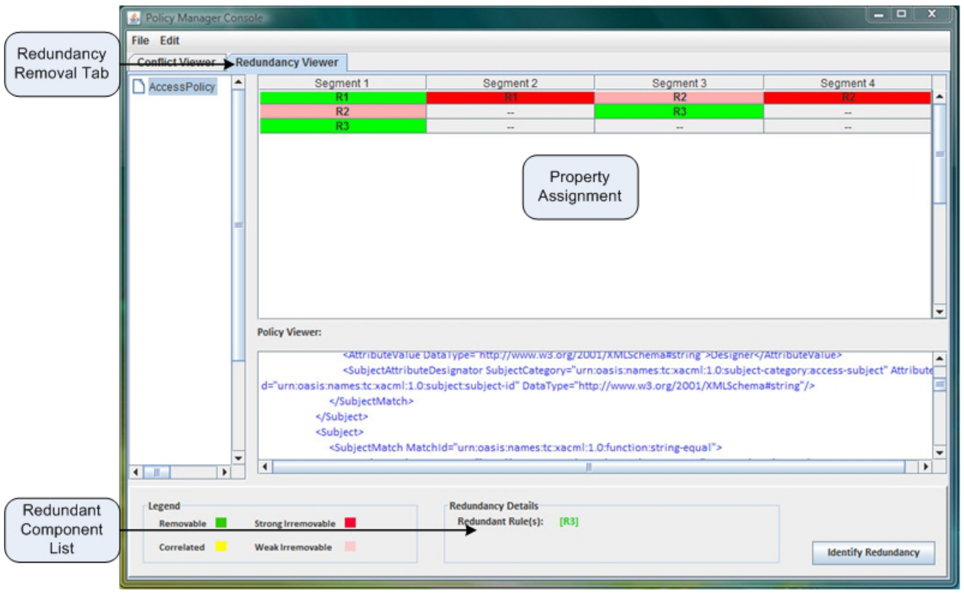
Segment Effect: Deny

Figure 5.15: Policy Analyzer Interface - Segment Detail View

upon the component selected in a policy hierarchy.

Similarly, administrator is provided with the policy-based segments as well as the information about property assignment for each component subspace to perform redundancy analysis. Different color for each subspace represents particular property assignment. *Red* subspace represents *Strong Irremovable*, *Pink* specifies *Weak Irremovable*, *Yellow* represents *Correlated* and *Green* is used to identify *Removable* property. Once administrator chooses to remove all redundancies by clicking *Identify Redundancy* button, correlation breaking process is performed for assigning appropriate properties to the *Correlated* subspaces (if any) and redundant components are displayed to the administrator. Figure 5.16 shows the user interface snapshot of redundancy analysis for an access control policy.

We believe that the visualization interfaces provided in our tool will assist administrators in viewing and analyzing the outputs from policy anomaly analysis and facilitate a more effective and efficient anomaly resolution. It helps administrators by minimizing the portions of the policy information that they need to examine at



Segment 1	Segment 2	Segment 3	Segment 4
R1	R1	R2	R2
R2	--	R3	--
R3	--	--	--

Policy Viewer:

```
                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Designer</AttributeValue>
                <SubjectAttributeDesignator SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject" Attribute
d="urn:oasis:names:tc:xacml:1.0:subject:subject-id" DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </SubjectMatch>
        </Subject>
        <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
```

Legend

Removable	🟩	Strong Irremovable	🟥
Correlated	🟨	Weak Irremovable	▫

Redundancy Details

Redundant Rule(s): [R3]

Figure 5.16: Policy Analyzer Interface - Redundancy Details View

any given time. It also offers an intuitive and succinct view of the conflicting policy components and enables administrator to better understand policy anomalies.

Chapter 6

IMPLEMENTATION AND EVALUATION

We have implemented our generic policy analysis and management approach in a proof of concept tool called *Generic Policy Analyzer* using Java for policy administrators. Based on our generic policy analysis and management mechanism, it consists of three core components: *generic policy representation, policy analysis module* and *graphic user interface (GUI)*. Currently our proof of concept tool supports analysis and management of XACML and Firewall policies only. Here are the implementation details for each module:

1. *Generic Policy Representation*: We used *protege* [9] to define generic access control policy ontology. To create ontology instance for a specific policy, we use OWL API [14]. To populate all required information in ontology instance, we need to parse the information from a given policy. For XACML we use *Sun* XACML implementation [12]. In case of firewall policy, we use the parser based on FIREMAN [32] implementation, to obtain all the required information. Once ontology instance is populated, we convert it into binary decision diagram (BDD) for efficient representation of a policy instance. We used JavaBDD [5], which is based on BuDDy package [1] for this purpose.

2. *Policy Analysis and Management*: This module has two main parts: (a) Conflict Detection and Resolution (b) Redundancy Removal

   Conflict Detection and Resolution We generate disjoint authorization spaces of given policy from its BDD representation. Further we identify different kinds of segments (conflicting and non-conflicting) and corresponding correlation groups using algorithms described in Chapter 5. 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 strategy-based approach. Currently, we provide risk-aware strategy based approach only for the firewall policies. In strategy-based approach, strategies are assigned to each conflicting segment by administrator. The strategy mapping module takes conflict correlation groups and effect constraints of conflicting segments as inputs and then maps assigned strategies to the standard XACML or firewall policy combining algorithms.

Redundancy Removal We use disjoint segments and correlated groups generated during conflict detection and resolution for redundancy removal. The property assignment module takes segment correlation groups as inputs and automatically assigns corresponding property to each subspace covered by the segments of policy components. The assigned properties are in turn utilized to identify redundancies according to algorithms described in Chapter 5.

3. *Graphical User Interface*: We provide the graphical user interface for all steps involved in our policy analysis and management approach. Typical scenario starts with, administrator providing access control policy with its type. Administrator is then provided with the conflict details in one tab and redundancy details in another tab. For conflict details, administrator can choose specific policy or policy set from given policy. Then administrator may choose to analyze either each and every disjoint segment or only conflict segments arranged in correlated groups for conflict analysis. Administrator may individually analyze details of each conflict and resolve it. For redundancy analysis, administrator analyzes the property assignment for each component subspace and then chooses to remove the redundancies in a given policy. For GUI implementation, we use Java Swing [6] API to create various user interface components.

We evaluated the efficiency and effectiveness of `Generic Policy Analyzer` for policy analysis on both real-life and synthetic XACML as well as firewall policies. We performed our experiments on Intel Core 2 Duo CPU 3.00 GHz with 3.25 GB RAM running on Windows XP SP2. Table 6.1 summarizes XACML as well as Firewall policies used for evaluation.

Real-life XACML policies utilized for evaluation were collected from different sources. Two of the policies, *CodeA* and *Continue-a* are XACML policies used in [22]; among them, *Continue-a* is designed for a real-world Web application supporting a conference management. *GradeSheet* is utilized in [19]. It is difficult to get a large volume of real-world policies because they are often considered to be highly confidential. Thus, we generated two large synthetic policies *SyntheticPolicy-1* and *SyntheticPolicy-2* for further evaluating the performance and scalability of our tool. We also used *SamplePolicy*, which is the example XACML policy represented in Figure 2.2, in our experiments. Table 6.1 summarizes the basic information of each XACML policy including the number of rules, the number of policies, and the number of policy sets.

Table 6.1: Summary of Access Control Policies used for evaluation.

Policy	Rule (#)	Policy (#)	Policy Set (#)
XACML Policies			
1 (CodeA)	4	2	5
2 (SamplePolicy)	6	2	1
3 (GradeSheet)	13	1	0
4 (SyntheticPolicy-1)	147	30	11
5 (Continue-a)	312	276	111
6 (SyntheticPolicy-2)	456	65	40
Firewall Policies			
1 (A)	12	1	0
2 (B)	25	1	0
3 (C)	52	1	0
4 (D)	83	1	0
5 (E)	132	2	1
6 (F)	354	3	1

Similarly, firewall polices used for evaluation were obtained from our lab environment or synthetically generated. Table 6.1 summarizes each of the firewall policy used for evaluation and its size in terms of number of rules. ACLs A, B and

Table 6.2: Conflict detection and redundancy removal algorithms evaluation.

Policy	**Partitions (#)**	**Conflict Detection**			**Redundandancy Removal**		
		Policy Level(#)	Policy Set Level(#)	Time (s)	Policy Level(#)	Policy Set Level(#)	Time (s)
XACML Policies							
1 (CodeA)	6	1	1	0.095	1	0	0.096
2 (SamplePolicy)	8	0	2	0.106	0	2	0.109
3 (GradeSheet)	18	0	4	0.125	0	2	0.132
4 (SyntheticPolicy-1)	205	8	14	0.364	7	4	0.359
5 (Continue-a)	439	9	14	0.621	10	7	0.597
6 (SyntheticPolicy-2)	523	29	15	0.914	14	8	0.903
Firewall Policies							
1 (A)	15	3	0	0.119	2	0	0.113
2 (B)	36	5	0	0.153	3	0	0.151
3 (C)	89	11	0	0.196	5	0	0.189
4 (D)	127	18	0	0.224	6	0	0.213
5 (E)	183	23	5	0.417	13	3	0.431
6 (F)	405	41	11	0.589	16	7	0.603

C were synthetically generated from ACLs deployed in lab environment for initial testing purpose. ACLs D, E and F are obtained from our lab environment.

We conducted three separate sets of experiments for the evaluation of generic policy representation, conflict detection and redundancy removal respectively. Table 6.2 summarizes our evaluation results.

Evaluation of Generic Policy Representation For generic approach of policy analysis and management, correct generic representation of access control policies using ontology is extremely important. Correct generic representation of access control policy should have same properties like number of *policy set, policy* or *rules* when compared to the original policy. We calculated number of *policy sets, policies* or *rules* in an ontology instance created for access control policy, before it is converted to a BDD representation. We observed that these numbers for ontology instance of policy match with the number of attributes for original policy as described in Table 6.1. It clearly proves that, generic ontology instance created for XACML as well as Firewall policies correctly depicts the properties of original access control policies.

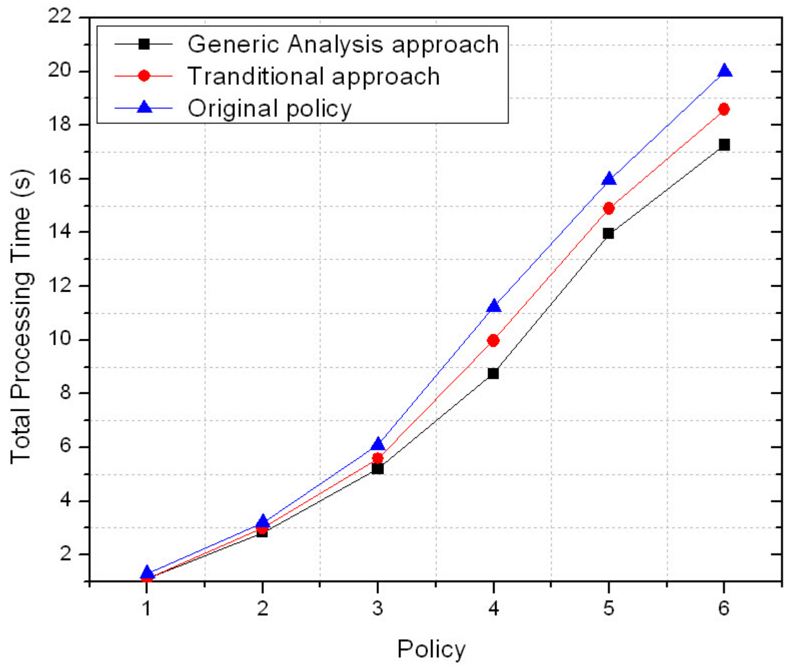


Figure 6.1: Performance improvement after Redundancy Removal

Evaluation of Conflict Detection Time required by `Generic Policy Analyzer` for conflict detection highly depends upon the number of segments generated for each access control policy. The increase of the number of segments is proportional to the number of components contained in an XACML or a firewall policy. From Table 6.2, we observe that `Generic Policy Analyzer` performs conflict detection fast enough to handle larger size XACML as well as firewall policies. We also observe that even for some complex policies with multiple levels of hierarchies along with hundreds of rules such as real-life XACML policy *Continue-a*, two synthetic XACML policies or firewall policy ACLs E and F, it performs conflict detection efficiently. The time trends observed from Table 6.2 are very promising, and hence provides the evidence of efficiency of our conflict detection approach.

Evaluation of Redundancy Removal In the third set of experiments, we evaluated our redundancy analysis approach based on our experimental policies. Table 6.2

shows that, redundancies in both XACML as well as firewall policies can be identified in a very short time. This can be observed even for complex XACML policies, where redundancies are identified at both policy as well as policy set level. Furthermore, when redundancies in a policy are removed, the performance of policy enforcement is improved. To verify this, we choose to evaluate policy enforcement performance for XACML policies in our experiment. We evaluate policy enforcement performance only for XACML policies because *Sun* XACML PDP [12] provides standard API for Policy Decision Point (PDP) implementation as well creating XACML access requests for experimental purpose. We conducted the evaluation of effectiveness by comparing our redundancy analysis approach with *traditional* redundancy analysis approach [16, 28], which can only identify redundancy relations between *two* rules. Figure 6.1 depicts the total processing time for *Sun* XACML PDP [12] in responding 10,000 randomly generated XACML requests. The evaluation results clearly shows that the processing times are reduced after eliminating redundancies in XACML policies applying either *traditional* approach or our approach, and our approach obtains better performance improvement than *traditional* approach.

Chapter 7

CONCLUSION

We have designed an innovative framework for policy analysis and management, which unifies the systematic detection and resolution of policy anomalies for different types of access control policies. This framework presented the generic access control policy representation approach using policy ontology, which helps in realizing policy analysis and management tasks for different types of policies. We used policy based segmentation technique for effective and efficient policy anomaly analysis. As part of this framework, we also presented an intuitive graphical user interface for performing policy analysis and management tasks.

Our contribution in this research thesis is the generic policy representation using policy ontology and the framework for analysis and management of different types of access control policies based on this generic policy representation. We also implemented this framework as a proof of concept tool called *Generic Policy Analyzer*, which currently supports XACML and Firewall policy analysis. Our experimental results show that, *Generic Policy Analyzer* can discover policy anomalies in XACML and Firewall policies efficiently and effectively. We believe that our generic approach for the policy analysis and management will significantly help administrators manage the different types of access control policies such as network level policies, system level policies and web application management policies. It will help enhance the correctness and effectiveness of policy management tasks, by avoiding any potential anomalies.

As a future work, this framework can be extended to support other types of access control policies by extending our policy ontology. Current implementation of storing an ontology as well as ontology instance in a file format can be extended to use Sesame [11] knowledge repositories to store ontological data and SPARQL [31] query language for ontology operations. Also, more fine grained strategies can be supported for effective conflict resolution. Implemented tool can be extended to provide features such as logging, version control as well as rollback for audit purpose, which would help policy administrators in systematically analyzing and managing the policies.

[1] Buddy version 2.4. http://sourceforge.net/projects/buddy.

[2] Cisco IOS Firewall. http://www.singlepointoc.com/products/cisco/docs_security/product_data_sheet09186a0080117962.pdf.

[3] Cloud Computing. http://csrc.nist.gov.

[4] eXtensible Access Control Markup Language (XACML) Version 2.0. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[5] Java BDD. http://javabdd.sourceforge.net.

[6] Java Swing. http://download.oracle.com/javase/6/docs/technotes/guides/swing/index.html.

[7] OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/.

[8] Ponder Policy. http://www-dse.doc.ic.ac.uk/Research/policies/ponder.shtml.

[9] Protege Ontology Editor. http://protege.stanford.edu/.

[10] Qualys. http://www.qualys.com.

[11] Sesame Knowledge Repository. http://www.openrdf.org.

[12] Sun XACML Implementation. http://sunxacml.sourceforge.net.

[13] TENABLE Network Security. http://www.nessus.org/nessus.

[14] The OWL API. http://owlapi.sourceforge.net/.

[15] 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.

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

[17] 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.

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

[19] 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.

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

[21] G. Chowdhury. Template Mining for Information Extraction from Digital Documents. *Library Trends*, 48(1):182–208, 1999.

[22] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM.

[23] I. Fundulaki and M. Marx. Specifying access control policies for XML documents with XPath. In *Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 61–69. ACM New York, NY, USA, 2004.

[24] 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. ACM, 2010.

[25] H. Hu, G.-J. Ahn, and K. Kulkarni. Anomaly Discovery and Resolution in Web Access Control Policies. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, page To Appear. SACMAT, 2011.

[26] 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.

[27] 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.

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

[29] P. Mell, K. Scarfone, and S. Romanosky. A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-Forum of Incident Response and Security Teams, June*. Citeseer, 2007.

[30] M. Schiffman, G. Eschelbeck, D. Ahmad, A. Wright, and S. Romanosky. CVSS: A Common Vulnerability Scoring System. *National Infrastructure Advisory Council (NIAC)*, 2004.

[31] E. Sirin and B. Parsia. Sparql-dl: Sparql query for owl-dl. In *3rd OWL Experiences and Directions Workshop (OWLED-2007)*, volume 4. Citeseer, 2007.

[32] 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*, page 15, 2006.