Technical Report

ASSURED RESOURCE SHARING IN AD-HOC COLLABORATION

Award No: DE-SC0004308

Project Duration: 2010 - 2015

Principal Investigator: Gail-Joon Ahn, Arizona State University

Executive Summary

The project seeks an innovative framework to enable users to access and selectively share resources in distributed environments, enhancing the scalability of information sharing. We have investigated secure sharing & assurance approaches for ad-hoc collaboration, focused on Grids, Clouds, and ad-hoc network environments. The tasks accomplished by this project are as follows:

1. Secure sharing in Grids and Cloud
Traditional access control mechanisms cannot support dynamic natures of security & privacy requirements derived from Grids and Cloud environments. Also, security and privacy issues should be fully articulated to design and develop effective and innovative security modules and policy management for collaborative environments. To address such challenges, we focused on access control and delegation models for ad-hoc collaboration including composite policies and schema integration. In addition, we attempted to detect unexpected and unauthorized information flows, and visualize our results in an effective manner. Also, we investigated how cryptographic approaches can enhance the integrity measurement in clouds.

2. Policy analysis for assurance
To build trustworthy sharing environments, it is inevitable to ensure that system properties are complied with security policies and requirements. Especially, in ad-hoc collaboration, it is critical to check the integrity measurements of each party. However, existing methods on integrity measurements and policy managements lack effective procedures and mechanisms. Hence, we also studied how integrity measurements of remote sites can be analyzed by policy analysis mechanisms and proposed a remote attestation framework. In addition, we focused on policy anomaly detection and resolution approaches for firewall policies and web-based policies.

3. Attribute-based access control
Furthermore, the collaboration among various facilities in DOE requires proper authentication and authorization approaches to provide assured sharing of data and resources in a real time manner. It is necessary to investigate what characteristics should be considered to support attribute provisioning and policy enforcement for collaborative environments. Even though we witness there exist several solutions, those solutions did not consider potential risks that should be avoided. Therefore, we attempted to propose attribute-based access control in ad-hoc collaboration. Also, we investigated a formal way to describe identified characteristics and properties so that we

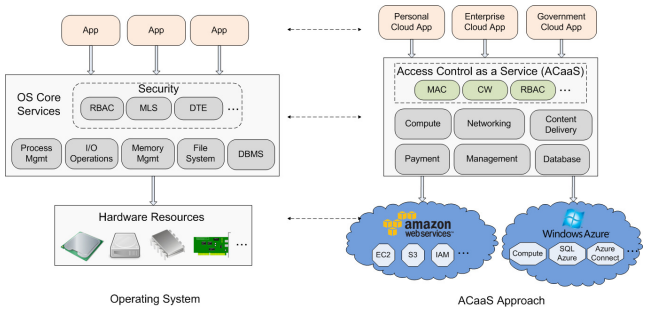


Fig. 1. ACaaS vs. security modules in operating system.

can eventually formulate attribute-based access control model and relevant policy management schemes.

I. Research Task: ACaaS for Clouds

A. Overview

Securely maintaining valuable digital assets in clouds is critical for both cloud service providers and customers. The diversity of cloud services across a wide range of organizations and domains requires various security requirements. Accordingly, a comprehensive and adaptive access control mechanism needs to be in place to support various security policy models for the diverse security needs. However, current cloud computing platforms such as AWS, Windows Azure, Google App Engine, and Eucalyptus all fail to meet such identified needs. Towards this, we propose the concept of access control as a service (ACaaS) with the spirit of pluggable access control modules in modern operating systems. As shown in Figure 1, we draw an analogy between computing, storage, network, and other resources provided by IaaS providers and hardware resources in physical machines such as CPU, disk, and network stack. Cloud provider offerings can be mapped to operating system services such as process management, memory management, scheduling, I/O operations, and networking. For instance, process management conducts basic tasks including starting and suspending processes, CPU allocation, and scheduling for multiple processes. Similarly, computing services in a cloud handle booting and terminating virtual machines instances, allocating resources, and scheduling computing tasks and workflows. For security purposes, authorization modules and policies in traditional operating systems (e.g., Linux) can be dynamically loaded (e.g., SELinux modules), and every access to underlying resources from processes and applications is then be controlled. Similarly, ACaaS can load different access control modules and support various security policy models for different cloud customers, such as mandatory access control (MAC) [36], the Chinese Wall security policy (CW) [9], and role based access control [37]. This plug & play fashion enables parallel evolution of cloud customer's own policy specifications and a cloud provider's security enforcement mechanisms.

B. AWS Access Control Service and Its Limitations

To motivate the design and development of ACaaS, we analyze the access control mechanism in Amazon AWS cloud platform - a service called Identity and Access Management (IAM) [2] which enables an organization to securely control users' access to the AWS services and resources subscribed by the organization. IAM defines security policies with a set of pre-defined components which consists of following components: *Users*, *Groups*, *Actions*, *Objects*, *Permissions*, *Constraints*, *User-Group-Assignment* (UGA), *Permission-User-Assignment* (PUA),

and *Permission-Group-Assignment* (PGA). *Permissions* are defined in the form of *Actions* on *Objects* under certain *Constraints*.

AWS IAM enables cloud customers to manage users and user permissions to secure their resources in clouds. However, we identify several limitations of IAM for enterprise cloud customers as follows.

1) IAM directly assigns permissions to users. With increasing outsourcing computing infrastructures to IaaS, the number of users and permissions can be quite dynamic and in a very large scale. The management cost for the mapping between users and permissions can be extremely high.
2) IAM supports groups to categorize users and let users explicitly obtain permissions assigned to groups they belong to. However, groups are organized in a flat structure, which cannot reflect the hierarchical structures of organizations. For example, a global sales department of a multinational company should have all the permissions of its regional sales departments. Besides, if an IAM policy is removed from a group, the permission associated with the policy is revoked as well, which is not necessary in many cases.
3) IAM allows to specify static constraints on permissions. However, it lacks a systematic support for many other important constraints such as separation of duty (SoD), a well-known principle for preventing the potential fraud. SoD divides the responsibility of a critical task into different people. When many financial and governmental systems are shifting into cloud platforms, SoD issues become even more critical. If permissions with conflict-of-interest issues are assigned to the same user, many valuable assets in clouds can be jeopardized.
4) Session management is missing in IAM such that all permissions of users are effective all the times, which conflicts with the principle of least privilege. Users should be able to manage their sessions for performing tasks. Besides, without session management, dynamic SoD cannot be enforced.
5) IAM does not distinguish administrators and regular users clearly. The root user with the AWS account has both administrative and regular permissions, which also conflicts with the principle of least privilege. Ideally, permissions associated with an AWS account should be split into multiple units.

AWS recently released a new IAM role feature, which enables an EC2 instance running with a predefined IAM role to securely access other AWS service APIs [1]. However, this feature is still too preliminary and coarse-grained to address these limitations. First, it only supports assigning IAM roles to an EC2 instance level but not to user level. Hence, all applications running in an EC2 instance assume the same set of permissions. This violates the least privilege principle, since it is very general that different applications in an EC2 instance run on behalf of different users to have different permissions. Second, an IAM role does not support session management such that an EC2 instance can only run with a single IAM role. Without re-launching the EC2 instance, it is impossible to switch between different IAM roles during the runtime. Furthermore, EC2 role does not support other important RBAC features such as role hierarchy and delegation. Fundamentally, we claim that the EC2 role is more similar to the traditional concept of "group" in access control and there is no RBAC model formulated in AWS.

C. Design of ACaaS$_{RBAC}$ for AWS

In this section, we present the design of ACaaS$_{RBAC}$, a reference architecture of ACaaS that supports RBAC for Amazon AWS cloud platform. There are several reasons that we choose

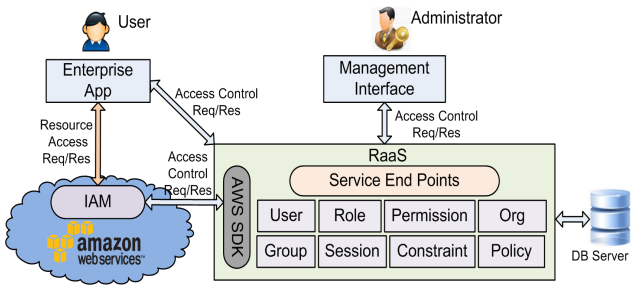


Fig. 2. ACaaS$_{RBAC}$ system architecture for AWS.

to support RBAC for AWS cloud platform. First, RBAC has been widely adopted in enterprise applications. When those applications are moving to clouds, RBAC should be naturally supported in cloud environments. According to a recent cloud market overview [41], RBAC is one of the key criteria to evaluate cloud computing solutions. Second, RBAC is a very generic access control model, and we believe tackling RBAC in clouds requires to address many challenges that will be identically addressed for supporting other access control models with ACaaS.

1) Challenges of Supporting RBAC for AWS: In order to provide RBAC as a service for AWS cloud platform, there are several critical challenges:

Challenge 1: Efficient role hierarchy management. In cloud computing environments, due to dynamic business needs and scalable resource provisioning, the number of roles in an organization could be very large and fluctuated frequently. Accordingly, role hierarchies in the organization could be complex and need to be updated. It is crucial to have an efficient way to manage role hierarchies in terms of both maintenance and update/change management.

Challenge 2: Session management. Session management should be supported to track users' interactions and meet the least privilege principle. Users should be able to activate or deactivate their roles for performing certain tasks in their sessions. With the nature of highly distributed service-oriented computing infrastructure, ACaaS$_{RBAC}$ has to seamlessly support session management without compromising the security property of RBAC.

Challenge 3: SoD support and management of privileged account. SoD constraints should be specified by administrators and each cloud should be able to enforce those constraints for avoiding permission abuse and unexpected fraud. The super user of an AWS account has all privileges upon cloud resources, which should be split into different units or roles in ACaaS$_{RBAC}$.

Challenge 4: System integration and minimal overhead. To leverage the pluggable capabilities provided by ACaaS, ACaaS$_{RBAC}$ services should be easily integrated with customers' applications. Besides, this integration should introduce acceptable performance and network traffic overheads between customers' applications and AWS cloud platform.

2) System Design: Our design of ACaaS$_{RBAC}$ addresses all identified challenges I-C.1 with a service-oriented RBAC for AWS cloud resources. Figure 2 shows the system architecture of ACaaS$_{RBAC}$. In current AWS platform, enterprise applications are able to access cloud resources on behalf of enterprise users, where the AWS IAM enforces security policies defined by enterprise administrators. ACaaS$_{RBAC}$ introduces *RBAC as a service (RaaS)*, which is an RBAC module designed based on NIST RBAC standard [15] and can be hosted by AWS or any third party service provider. This module supports and enforces RBAC configurations by leveraging Amazon IAM service for enterprise administrators. It also provides session capability for enterprise users, e.g., a user or an application can activate and deactivate roles within a single session when accessing resources in AWS *.

*We note that an RBAC session here is usually different from that in AWS services, e.g., a DynamoDB session, although they can be co-related in an implementation.

RaaS provides browser interfaces for enterprise administrators and users to configure RBAC policies. In our prototype (cf. Section I-D), RBAC policies are implemented as relational database entries. RaaS also provides web services APIs such that operations can be integrated into administrative tools or applications from the enterprise side. The results of these configurations are IAM policies that are pushed back to AWS, so that any further access from enterprise applications will be controlled by these policies. Since IAM does not support RBAC, RaaS transforms all role-based policies of a user into AWS permission based policies which can be understandable and enforceable by IAM. The transformation process is to generate direct relationships between users and permissions by removing the role notion between them in the role-based policies. For an enterprise that already has AWS resources in active use, RaaS provisions the information of users, groups, permissions, objects, and actions from AWS via IAM APIs.

RaaS contains eight sub-modules: *Organization*, *User*, *Group*, *Role*, *Permission*, *Session*, *Constraint* and *Policy*, each of which is exposed as web services. The *Organization* sub-module manages (e.g., list, register, and delete) organizations to support the multi-tenant feature of ACaaS$_{RBAC}$. The *Group* sub-module manages user groups of a single organization for administrative users, where the user information is provisioned from the organization's AWS account. The *Permission* sub-module manages the permissions of ACaaS$_{RBAC}$. There are two types of permissions: user permissions (*P*) which are inherited from existing AWS permissions, and administrative permissions (*AP*), which are effective for RaaS only. The *Permission* sub-module maintains both *P* and *AP*, and manages their assignment relations with roles. We elaborate the design of sub-modules in the rest of this section.

User and Permission: This sub-module provides management on regular users (*U*), administrative users (*AU*), and permissions (*P*), and provides interfaces and APIs to create, delete, activate, and deactivate *U*, *AU*, and *P*, and manage their group memberships and role memberships. In RaaS design, both regular users and permissions are provisioned from AWS directly, with the credentials of the AWS account, which is usually the user who can create other users and policies in IAM. Therefore, RaaS does not store any information for *U* and *P*.

Administrative users can be further categorized into two types: root administrative users and regular administrative users. Root administrative users are able to add and delete regular administrative users, and manage their permissions. Regular administrative users are able to perform certain administrative actions (ACT_A) on administrative resources (RES_A) based on their administrative role memberships. By default, a root administrative user is the AWS user that owns the IAM account. With the interfaces provided by RaaS, this user can further create regular administrative users and roles, and their administrative scopes. By controlling the user-role assignments to regular users and administrative users, RaaS can support flexible policies such as splitting privileged accounts, and separation of duty constraints.

Role: This sub-module creates and deletes roles (*R*) and administrative roles (*AR*), and most importantly, it manages role hierarchies (*RH*). RaaS distinguishes *R* and *AR* such that mandatory security policies can be enforced, e.g., by only assigning necessary regular roles to regular users. For least privilege purposes, RaaS may introduce many primitive regular roles, each of which is assigned with atomic permissions. This usually introduces large number of regular roles in a system, where role hierarchy becomes necessary.

Towards efficient role-related operations, we adopt the Nested Set Model [23] for role hierarchy in our implementation, which assigns left and right values to represent a scope of each role in a role hierarchy. If the scope of a role is inside the scope of another role, it means the former role

Algorithm 1: *ComputeActivatePermissions(u, r_a) → P*

```
   Input: A user u wants to activate a role r_a
   Output: A permission set P, of which corresponding IAM policies need to be enforced
1  P, P_all ← ∅;
2  r_is ← getImmediateSeniorRole(r_a);
3  if hasRole(u, r_is) = TRUE AND active(u, r_is) = TRUE then
4      return ∅;
5  else
6      ComputeP(u, r_a);
7      foreach p ∈ Permissions(r_a) do
8          if p ∉ P_all then
9              add p into P;
10     return P;
11 ComputeP(User u, Role r_a)
12 begin
13     R ← getSiblingRoles(r_a) ⋃ getImmediateJuniorRoles(r_a);
14     if R = ∅ then return;
15     ;
16     else foreach r ∈ R do
17         if active(u, r) = TRUE then
18             foreach p ∈ Permissions(r) do
19                 if p ∉ P_all then
20                     add p into P_all;
21         else
22             ComputeP(u, r);
23     ;
```

is junior to the latter role. A big advantage of Nested Set Model for managing role hierarchies is that only one entry for each role needs to be maintained in the database. This avoids storing a lot of redundant role relationship information to maintain role hierarchies. It is also easy to update role hierarchies by updating associated roles' left and right values. One limitation of our current design is that, the Nested Set Model only supports to represent limited role hierarchies in a simple tree structure, i.e., one of two role hierarchies based on NIST RBAC standard [15]. As our future work, we would extend our implementation towards more general role hierarchy management.

Session: This module provides session management including role activation and deactivation for regular users. Usually, a user can be assigned with several roles at the same time. In a session, to meet the least privilege principle, only some of those roles which are needed to perform a certain task should be activated. After finishing this task, relevant activated roles can be deactivated. When activating a role, permissions associated with that role should be effective by enforcing corresponding Amazon IAM policies such that users are able to access cloud resources with needed permissions.

An intuitive way for deactivating a role and activating another role is to remove all permissions of the original role for the user in IAM, and then create new permissions and policies, and push all new policies to IAM. This is the approach we adopt if these two roles have no overlapped permissions. With the existence of role hierarchy, it may not be necessary to generate and enforce Amazon IAM policies for all permissions of the new role since some permissions may have already been effective, e.g., these two roles have common inherited permissions from role hierarchy.

To improve the efficiency of role activation and minimize the communication overheads between RaaS module and AWS cloud platform, we implement an efficient role activation algorithm to compute a minimum permission set when activating a role, as shown in Algorithm 1.

Algorithm 2: *ComputeDeactivatePermissions(u, r_d) → P*

```
   Input: A user u wants to deactivate a role r_d
   Output: A permission set P, of which corresponding IAM policies need to be enforced
1  P, P_all ← ∅;
2  R_senior ← getSeniorRoles(r_d);
3  if R_senior ≠ ∅ then
4      foreach r ∈ R_senior do
5          if active(u, r) = TRUE then
6              return ∅;
7  R_sibling ← getActivatedSiblingRoles(r_d);
8  if R_sibling = ∅ then
9      return Permissions(r_d);
10 else
11     foreach r ∈ R_sibling do
12         if active(u, r) = TRUE then
13             foreach p ∈ Permissions(r) do
14                 if p ∉ P_all then
15                     add p into P_all;
16     foreach p ∈ Permissions(r_d) do
17         if p ∉ P_all then
18             add p into P;
19     return P;
```

This algorithm works as follows: if a user *u* owns an immediate senior and activated role to role r_a which needs to be activated, an empty permission set is returned and no policy needs to be generated and enforced by Amazon IAM. Otherwise, For each sibling role and immediate junior role to role r_a, their senior-most and activated junior roles are identified recursively and a corresponding permission set P_{all} is constructed. Then for each permission associated with the role r_a, if it does not belong to P_{all}, then it will be added to the returned permission set. On the other hand, when deactivating roles, the deactivation should not affect functionalities of other activated roles when they have overlapped permissions.

Correspondingly, we implement a role deactivation algorithm, as shown in Algorithm 2. The algorithm works as follows: if any senior role to a role r_d which needs to be deactivated is activated, an empty permission set is returned and no policy needs to be generated and enforced by Amazon IAM. Otherwise if role r_d does not have any activated sibling roles, a permission set containing all permissions associated with the role r_d is returned. Otherwise, a permission set P_{all} containing all permissions associated with activated sibling roles of the role r_d is constructed. Then for each permission associated with the role r_d, if it does not belong to P_{all}, it is added into the returned permission set.

Constraint: This sub-module provides constraints management services including creating, deleting, updating static constraints as well as separation of duty constraints. Static constraints are specified on permissions in existing Amazon IAM constraints format discussed in Section I-B and enforced by IAM. Only when the static constraints are satisfied, users are able to perform corresponding permissions. When creating an SoD constraint, a set of potential conflicting roles and a cardinality value need to be specified. The cardinality value is a threshold of the total role occurrence in the potential conflicting role set. When it is reached, IAM security policy of the user will be updated and any corresponding request will be denied. SoD constraints are enforced by *Constraint* sub-module itself when administrative users assign users to roles, or users activate their roles. Those static constraints are then converted into IAM policies and pushed into AWS.

Dynamic separation of duty (DSoD) constraints are usually enforced during runtime, e.g.,

conflicting roles cannot be activated in a single session. Similar constraints can be defined and checked by the session module.

Policy: This sub-module provides Amazon IAM policy generation and pushing services to ensure RBAC configurations of an enterprise can be reflected in AWS cloud platform. For example, when a user activates or deactivates roles, corresponding Amazon IAM policies are generated and pushed to the Amazon IAM policy engine for the enforcement. More specifically, for each permission in the permission set computed by Algorithm 1 or Algorithm 2, a corresponding IAM policy is constructed and sent to IAM for the enforcement. Policy transformation and deployment are also triggered by other administration actions that change the regular permissions of a user.

We note that our policy transformation is both complete and sound. That is, each state of the RaaS (the ACaaS$_{RBAC}$ relationships stored in its local database) can be translated to a set of IAM policies, e.g., each regular user has an IAM policy. This is due to the fact that both the users and permissions are provisioned from AWS directly. For each RaaS state, the net result of its configuration is a set of permissions that are authorized for a user, after revolving the constraints such as SoD and DSoD. Similarly, each translation corresponds to a valid IAM policy since the permissions are defined with valid resource names and actions defined by AWS.

D. Implementation

Based on our design, we have implemented a prototype system to provide RBAC services in AWS cloud platform through a web browser interface as well as web services. The core services of the system are implemented in Java based on AWS SDK 1.3.0 and exposed as SOAP-based web services using GlassFish Metro 2.2. A web-based management interface is developed by using JavaServer Pages (JSP) and MySQL Community Server 5.1. Both administrative users and normal users can log into the interface with their usernames and passwords. Administration tools can interact with the system by calling the SOAP-based web services APIs where the body of messages are signed with pre-shared secret keys.

All entities of the major components in ACaaS$_{RBAC}$ are stored in tables of a relational database, which jointly represents the state of the RaaS system. The name spaces of permissions, which are built on resources in AWS, are provisioned through AWS APIs. An administrative operation results in calling one or more AWS APIs, e.g., to create a user, a group, a permission, or add or remove an IAM policy in the root user's AWS account.

II. Research Task: Discovery and Resolution of Anomalies in Access Control Policies

A. Overview of XACML

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

```
1<PolicySet PolicySetId="PS1" PolicyCombiningAlgId="First-Applicable">
2   <Target/>
3   <Policy PolicyId="P1" RuleCombiningAlgId="Deny-Overrides">
4     <Target/>
5     <Rule RuleId="r1" Effect="Deny">
6       <Target>
7         <Subjects><Subject>      Designer  </Subject>
8                   <Subject>      Tester    </Subject></Subjects>
9         <Resources><Resource> Codes     </Resource></Resources>
10        <Actions><Action>       Change    </Action></Actions>
11      </Target>
12    </Rule>
13    <Rule RuleId="r2" Effect="Permit">
14      <Target>
15        <Subjects><Subject>      Designer  </Subject>
16                  <Subject>      Developer </Subject></Subjects>
17        <Resources><Resource> Reports   </Resource>
18                    <Resource> Codes     </Resource></Resources>
19        <Actions><Action>       Read      </Action>
20                  <Action>       Change    </Action></Actions>
21      </Target>
22      <Condition>      8:00 ≤ Time ≤ 17:00      </Condition>
23    </Rule>
24    <Rule RuleId="r3" Effect="Deny">
25      <Target>
26        <Subjects><Subject>      Designer  </Subject></Subjects>
27        <Resources><Resource> Reports   </Resource>
28                    <Resource> Codes     </Resource></Resources>
29        <Actions><Action>       Change    </Action></Actions>
30      </Target>
31      <Condition>      12:00 ≤ Time ≤ 13:00      </Condition>
32    </Rule>
33  </Policy>
34  <Policy PolicyId="P2" RuleCombiningAlgId="Permit-Overrides">
35    <Target/>
36    <Rule RuleId="r4" Effect="Deny">
37      <Target>
38        <Subjects><Subject>      Developer</Subject></Subjects>
39        <Resources><Resource> Reports   </Resource></Resources>
40        <Actions><Action>       Change    </Action></Actions>
41      </Target>
42    </Rule>
43    <Rule RuleId="r5" Effect="Permit">
44      <Target>
45        <Subjects><Subject>      Manager   </Subject>
46                  <Subject>      Designer  </Subject></Subjects>
47        <Resources><Resource> Reports   </Resource>
48                    <Resource> Codes     </Resource></Resources>
49        <Actions><Action>       Change    </Action></Actions>
50      </Target>
51    </Rule>
52  </Policy>
53</PolicySet>
```

Fig. 3. An example XACML policy.

request is applicable to the rule and it has a similar structure as the target of a policy or a policy set; the *condition* is a boolean expression to specify restrictions on the attributes in the target and refine the applicability of the rule; and the *effect* is either `permit` or `deny`. If an access request satisfies both the *target* and *condition* of a rule, the response is sent with the decision specified by the *effect* element in the rule. Otherwise, the response yields `NotApplicable` which is typically considered as `deny`.

An XACML policy often has conflicting rules or policies, which are resolved by four different *combining algorithms*: *Deny-Overrides*, *Permit-Overrides*, *First-Applicable* and *Only-One-Applicable* [33]. Figure 3 shows an example XACML policy. The root policy set PS_1 contains two policies, P_1 and P_2, which are combined using *First-Applicable* combining algorithm. The policy P_1 has three rules, r_1, r_2 and r_3, and its rule combining algorithm is *Deny-Overrides*. The policy P_2 includes two rules r_4 and r_5 with *Deny-Overrides* combining algorithm. In this example, there are four subjects: *Manager*, *Designer*, *Developer* and *Tester*; two resources: *Reports* and *Codes*; and two actions: *Read* and *Change*. Note that both r_2 and r_3 define conditions over the *Time* attribute.

B. Anomalies in XACML Policies

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

- **Anomalies at Policy Level:** A rule is ***conflicting*** with other rules, if this rule overlaps with others but defines a different effect. For example, the *deny* rule r_1 is in conflict with the *permit* rule r_2 in Figure 3 because rule r_2 allows the access requests from a designer to change codes in the time interval [8:00, 17:00], which are supposed to be denied by r_1; and a rule is ***redundant*** if there is other same or more general rules available that have the same effect. For instance, if we change the effect of r_2 to *Deny*, r_3 becomes redundant since r_2 will also deny a designer to change reports or codes in the time interval [12:00, 13:00].
- **Anomalies at Policy Set Level:** Anomalies may also occur across policies or policy sets in an XACML policy. For example, considering two policy components P_1 and P_2 of the policy set PS_1 in Figure 3, P_1 is ***conflicting*** with P_2, because P_1 permits the access requests that a developer changes reports in the time interval [8:00, 17:00], but which are denied by P_2. On the other hand, P_1 denies the requests allowing a designer to change reports or codes in the time interval [12:00, 13:00], which are permitted by P_2. Supposing the effect of r_2 is changed to *Deny* and the condition of r_2 is removed, r_4 is turned to be ***redundant*** with respect to r_2, even though r_2 and r_4 are placed in different policies P_1 and P_2, respectively.

A policy anomaly may involve in multiple rules. For example, in Figure 3, access requests that a designer changes codes in the time interval [12:00, 13:00] are permitted by r_2, but denied by both r_1 and r_3. Thus, this conflict associates with *three* rules. For another example, suppose the effect of r_3 is changed to *Permit* and the subject of r_3 is replaced by *Manager* and *Developer*. If we only examine *pairwise* redundancies, r_3 is not a redundant rule. However, if we check multiple rules simultaneously, we can identify r_3 is redundant considering r_2 and r_5 together. We observe that precise anomaly diagnosis information is crucial for achieving an effective anomaly resolution. In this work, we attempted to design a systematic approach and corresponding tool not only for accurate anomaly detection but also for effective anomaly resolution.

C. Underlying Data Structure

Our policy-based segmentation technique introduced in subsequent sections requires a well-formed representation of policies for performing a variety of set operations. Binary Decision Diagram (BDD) [11] is a data structure that has been widely used for formal verification and simplification of digital circuits. In this work, we leverage BDD as the underlying data structure to represent XACML policies and facilitate effective policy analysis.

Given an XACML policy, it can be parsed to identify subject, action, resource and condition attributes. Once these attributes are identified, all XACML rules can be transformed into Boolean expressions [6]. Each Boolean expression of a rule is composed of atomic Boolean expressions combined by logical operators $\vee$ and $\wedge$. Atomic Boolean expressions are treated as equality constraints or range constraints on attributes (e.g. $Subject =$ "$student$") or on conditions (e.g. $8:00 \leq Time \leq 17:00$).

Example 1: Consider the example XACML policy in Figure 3 in terms of *atomic Boolean expressions*. The Boolean expression for rule r_1 is:

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

The Boolean expression for rule r_2 is:

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

Boolean expressions for XACML rules may consist of atomic Boolean expressions with overlapping value ranges. In such cases, those atomic Boolean expressions are needed to be transformed into a sequence of new atomic Boolean expressions with disjoint value ranges. Agrawal et al. [3] have identified different categories of such atomic Boolean expressions and addressed corresponding solutions for those issues. We adopt similar approach to construct our Boolean expressions for XACML rules.

TABLE I

ATOMIC BOOLEAN EXPRESSIONS AND CORRESPONDING BOOLEAN VARIABLES FOR P_1.

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

We encode each of the atomic Boolean expression as a Boolean variable. For example, an atomic Boolean expression *Subject="Designer"* is encoded into a Boolean variable S_1. A complete list of Boolean encoding for the example XACML policy in Figure 3 is shown in Table I. We then utilize the Boolean encoding to construct Boolean expressions in terms of Boolean variables for XACML rules.

Example 2: Consider the example XACML policy in Figure 3 in terms of *Boolean variables*. The Boolean expression for rule r_1 is:

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

The Boolean expression for rule r_2 is:

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

BDDs are acyclic directed graphs which represent Boolean expressions compactly. Each nonterminal node in a BDD represents a Boolean variable, and has two edges with binary labels, 0 and 1 for *nonexistent* and *existent*, respectively. Terminal nodes represent Boolean value `T` (True) or `F` (False). Figures 4(a) and 4(b) give BDD representations of two rules r_1 and r_2, respectively.

Once the BDDs are constructed for XACML rules, performing set operations, such as unions ($\cup$), intersections ($\cap$) and set differences ($\setminus$), required by our policy-based segmentation algorithms (see Algorithm 1 and Algorithm 2) is efficient as well as straightforward. Figure 4(c) shows an integrated BDD, which is the difference of r_2' BDD from r_1' BDD ($r_2 \setminus r_1$). Note that the resulting BDDs from the set operations may have *less* number of nodes due to the canonical representation of BDD.

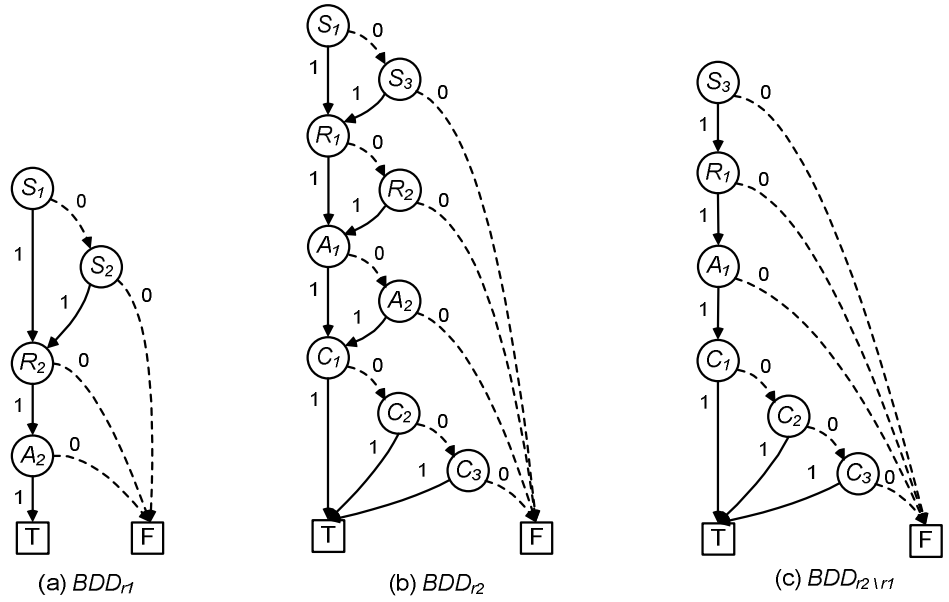


Fig. 4. Representing and operating on rules of XACML policy with BDD.

D. Conflict Detection and Resolution

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

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

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

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

Algorithm 3 shows the pseudocode of generating conflicting segments for a policy component P. An entire authorization space derived from a policy component is first partitioned into a set of disjoint segments. As shown in lines 17-33 in Algorithm 3, a function called `Partition()` accomplishes this procedure. This function works by adding an authorization space s derived from a rule r to an authorization space set S. A pair of authorization spaces must satisfy one of the following relations: *subset* (line 19), *superset* (line 24), *partial match* (line 27), or *disjoint* (line 32). Therefore, one can utilize set operations to separate the overlapped spaces into disjoint spaces.

Conflicting segments are identified as shown in lines 6-10 in Algorithm 3. A set of conflicting segments $CS : \{cs_1, cs_2, \ldots, cs_n\}$ from conflicting rules has the following three properties:

Algorithm 3: Identify Disjoint Conflicting Authorization Spaces of Policy P

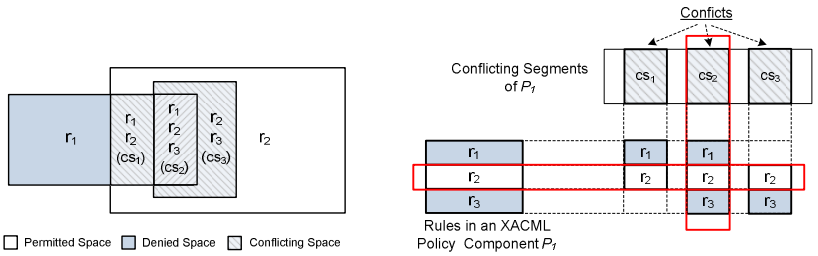
Input: A policy P with a set of rules.
Output: A set of disjoint conflicting authorization spaces CS for P.

```
/* Partition the entire authorization space of P into disjoint spaces*/
S.New();
S ⟵ Partition_P(P);
/* Identify the conflicting segments */
CS.New();
foreach s ∈ S do
    /* Get all rules associated with a segment s */
    R' ⟵ GetRule(s);
    if ∃r_i ∈ R', r_j ∈ R', r_i ≠ r_j and r_i.Effect ≠ r_j.Effect then
        CS.Append(s);

Partition_P(P)
R ⟵ GetRule(P);
foreach r ∈ R do
    s_r ⟵ AuthorizationSpace(r);
    S ⟵ Partition(S, s_r);
return S;

Partition(S, s_r)
foreach s ∈ S do
    /* s_r is a subset of s*/
    if s_r ⊂ s then
        S.Append(s \ s_r);
        s ⟵ s_r;
        Break;
    /* s_r is a superset of s*/
    else if s_r ⊃ s then
        s_r ⟵ s_r \ s;
    /* s_r partially matches s*/
    else if s_r ∩ s ≠ ∅ then
        S.Append(s \ s_r);
        s ⟵ s_r ∩ s;
        s_r ⟵ s_r \ s;
S.Append(s_r);
return S;
```

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

To facilitate the correct interpretation of analysis results, a concise and intuitive representation method is necessary. For the purposes of brevity and understandability, we first employ a two dimensional geometric representation for each authorization space segment. Note that a rule in an XACML policy typically has multiple fields, thus a complete representation of authorization space should be multi-dimensional. Also, we utilize colored rectangles to denote two kinds of authorization spaces: *permitted space* (white color) and *denied space* (grey color), respectively. Figure 5(a) gives a representation of the segments of authorization space derived from the policy P_1 in the XACML example policy shown in Figure 3. We can notice that five unique disjoint segments are generated. In particular, three conflicting segments cs_1, cs_2 and cs_3 are identified, representing three policy conflicts.

$^{\dagger}GetRule()$ is a function that returns all rules matching a request.



(a) Disjoint segments of authorization space for policy P_1

(b) Grid representation of policy conflict in policy P_1

Fig. 5. Authorization space representation for policy P_1 in the example XACML policy.

When a set of XACML rules interacts, one overlapping relation may be associated with several rules. Meanwhile, one rule may overlap with multiple other rules and can be involved in a couple of overlapping relations (overlapping segments). Different kinds of segments and associated rules can be viewed like Figure 5(a). However, it is still difficult for a policy designer or administrator to figure out how many segments one rule is involved in. To address the need of a more precise conflict representation, we additionally introduce a grid representation that is a *matrix-based* visualization of policy conflicts, in which space segments are displayed along the horizontal axis of the matrix, rules are shown along the vertical axis, and the intersection of a segment and a rule is a grid that displays a rule's subspace covered by the segment.

Figure 5(b) shows a grid representation of conflicts in the policy P_1 in in our example policy. We can easily determine which rules are covered by a segment, and which segments are associated with a rule. For example, as shown in Figure 5(b), we can notice that a conflicting segment cs_2, which points out a conflict, is related to a rule set consisting of three rules r_1, r_2 and r_3 (highlighted with a horizontal red rectangle), and a rule r_2 is involved in three conflicting segments cs_1, cs_2 and cs_3 (highlighted with a vertical red rectangle). Our grid representation provides a better understanding of policy conflicts to policy designers and administrators with an overall view of related segments and rules.

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

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

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

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

applicable component.

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

Algorithm 4: Identify Disjoint Conflicting Authorization Spaces of Policy Set PS

Input: A policy set PS with a set of policies or other policy sets.
Output: A set of disjoint conflicting authorization spaces CS for PS.

```
1  /* Partition the entire authorization space of PS into disjoint spaces*/
2  S.New();
3  S ⟵ Partition_PS(PS);
4  /* Identify the conflicting segments */
5  CS.New();
6  foreach s ∈ S do
7      E ⟵ GetElement(s);
8      if ∃e_i ∈ E, e_j ∈ E, e_i ≠ e_j and e_i.Effect ≠ e_j.Effect then
9          CS.Append(s);

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

Algorithm 4 shows the pseudocode of identifying disjoint conflicting authorization spaces for a policy set PS based on the tree structure. In order to partition authorization spaces of all nodes contained in a policy set tree, this algorithm recursively calls the partition functions, `Partition_P()` and `Partition_PS()`, to deal with the policy nodes (lines 16-17) and the policy set nodes (lines 19-20), respectively. Once all children nodes of a policy set are partitioned, we can then represent the authorization space of each child node (E) with two subspaces *permitted subspace* (E^P) and *denied subspace* (E^D) by aggregating all "`Permit`" segments and "`Deny`" segments, respectively, as follows:

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

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

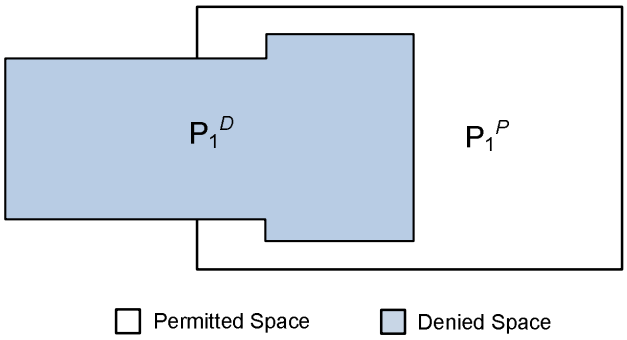
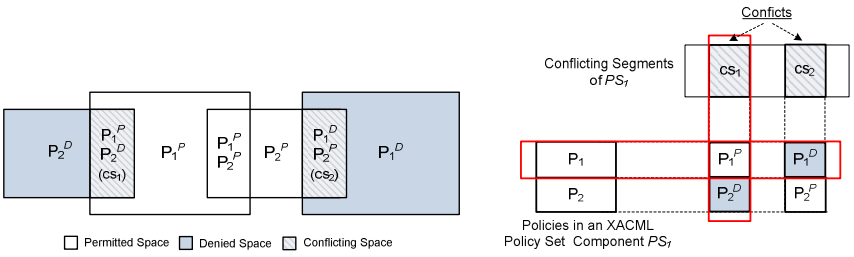


Fig. 6. Aggregation of authorization spaces for policy P_1 in the example XACML policy.

For example, since the combining algorithm ($\mathcal{CA}$) of the policy P_1 in our example XACML policy is *Deney-Overrides*, the effects of three conflicting segments shown in Figure 5 are "`Deny`". Figure 6 shows the result of aggregating authorization spaces of the policy P_1, where two subspaces P_1^P and P_1^D are constructed.

In order to generate segments for the policy set PS, we can then leverage two subspaces (E^P and E^D) of each child node (E) to partition existing authorization space set belonging to PS (lines 28-29). Figure 7(a) represents an example of the segments of authorization space derived from policy set PS_1 in our example policy (Figure 3). We can observe that seven unique disjoint segments are generated, and two of them cs_1 and cs_2 are conflicting segments. We additionally give a grid representation of conflicts in the policy set PS_1 shown in Figure 7(b). Then, we can easily identify that the conflicting segment cs_1 is related to two subspaces: P_1's *permitted* subspace P_1^P and P_2's *denied* subspace P_2^D, and the policy P_1 is associated with two conflicts, where P_1's *permitted* subspace P_1^P is involved in the conflict represented by cs_1 and P_1's *denied* subspace P_1^D is related to the conflict represented by cs_2.



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

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

Fig. 7. Authorization space representation for policy set PS_1 in the example XACML policy.

2) Fine-Grained Conflict Resolution: Once conflicts within a policy component or policy set component are identified, a policy designer can choose appropriate conflict resolution strategies to resolve those identified conflicts. However, current XACML conflict resolution mechanisms have limitations in resolving conflicts effectively. First, existing conflict resolution mechanisms in XACML are too restrictive and only allow a policy designer to select one combining algorithm to resolve all identified conflicts within a policy or policy set component. A policy designer may want to adopt different combining algorithms to resolve different conflicts. Second, XACML offers four conflict resolution strategies. However, many conflict resolution strategies exist [19], [17], [27], but cannot be specified in XACML. Thus, it is necessary to seek a comprehensive

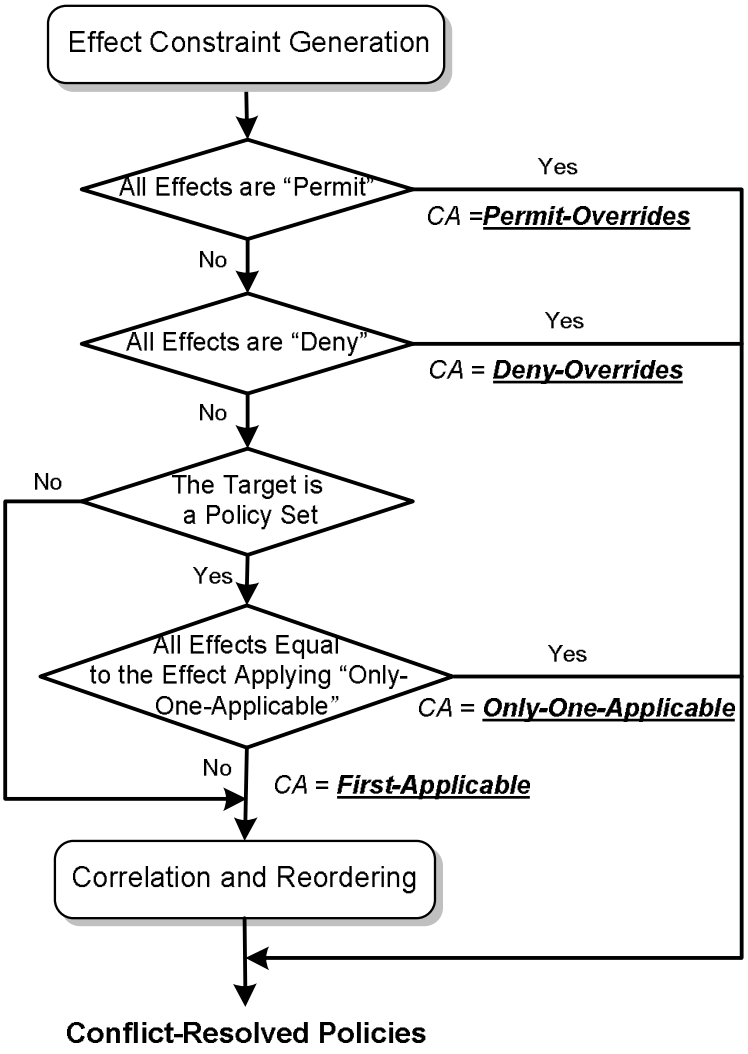


Fig. 8. Fine-grained conflict resolution framework.

conflict resolution mechanism for more effective conflict resolution. Towards this end, we introduce a flexible and extensible conflict resolution framework to achieve a fine-grained conflict resolution as shown in Figure 8.

Effect Constraint Generation from Conflict Resolution Strategy: Our conflict resolution framework introduces an *effect constraint* that is assigned to each conflicting segment. An effect constraint for a conflicting segment defines a desired response (either `permit` or `deny`) that an XACML policy should take when any access request matches the conflicting segment. The effect constraint is derived from the conflict resolution strategy applied to the conflicting segment, using a similar process of determining the effect of a conflicting segment described in Section II-D.1. A policy designer chooses an appropriate conflict resolution strategy for each identified conflict by examining the features of conflicting segment and associated conflicting components. In our conflict resolution framework, a policy designer is able to adopt different strategies to resolve conflicts indicated by different conflicting segments. In addition to four standard XACML conflict resolution strategies, user-defined strategies [27], such as *Recency-Overrides*, *Specificity-Overrides* and *High-Majority-Overrides*, can be implied in our framework

as well. For example, applying a conflict resolution strategy, *High-Majority-Overrides*, to the second conflicting segment cs_2 of policy P_1 depicted in Figure 5, an effect constraint *Effect = "Deny"* will be generated for cs_2.

Conflict Resolution Based on Effect Constraints: A key feature of adopting *effect constraints* in our framework is that other conflict resolution strategies assigned to resolve different conflicts by a policy designer can be *automatically* mapped to standard XACML combining algorithms, without changing the way that current XACML implementations perform. As illustrated in Figure 8, an XACML combining algorithm can be derived for a target component by examining all effect constraints of the conflicting segments. If all effect constraints are "`Permit`," *Permit-Overrides* is selected for the target component to resolve all conflicts. In case that all effect constraints are "`Deny`," *Deny-Overrides* is assigned to the target component. Then, if the target component is a policy set and all effect constraints can be satisfied by applying *Only-One-Applicable* combining algorithm, *Only-One-Applicable* is selected as the combining algorithm of the target component. Otherwise, *First-Applicable* is selected as the combining algorithm of the target component. In order to resolve all conflicts within the target component by applying *First-Applicable*, the process of reordering conflicting components is compulsory to enable that the first-applicable component in each conflicting segment has the same effect with corresponding effect constraint.

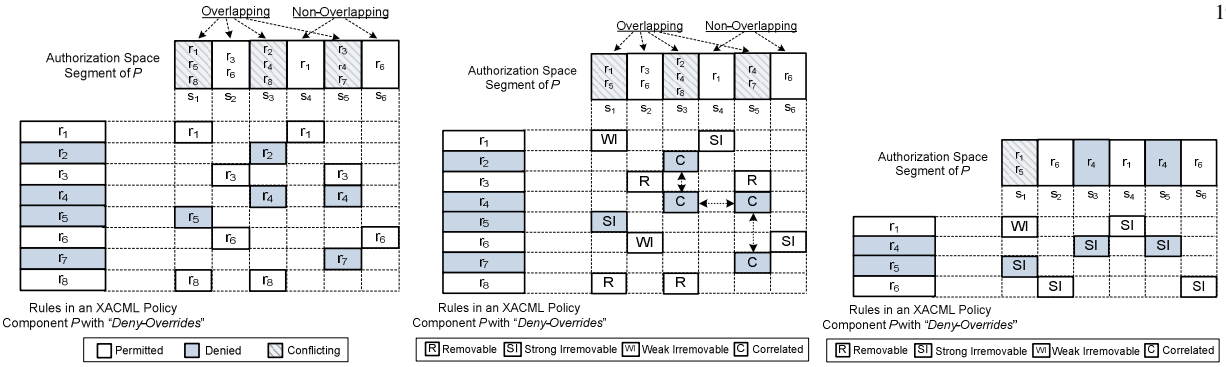
Practically, one XACML component may get involved in multiple conflicts. In this case, removing such a component to satisfy one effect constraint may violate other effect constraints. Therefore, we cannot resolve a conflict individually by reordering a set of conflicting components associated with one conflict. On the other hand, it is also inefficient to deal with all conflicts together by reordering all conflicting components simultaneously. Thus, we next introduce a correlation mechanism to identify dependent relationships among conflicting segments. The major benefit of identifying dependent relationships for a conflict resolution is to lessen the searching space of reordering conflicting components. The pseudocode of an algorithm for identify conflict correlation groups is given in Algorithm 5.

Algorithm 5: Conflicting Segment Correlation

```
   Input: A set of conflicting segments, C.
   Output: A set of groups for correlated segment, G.
1  G.New();
2  g ⟵ G.NewGroup();
3  foreach c ∈ C do
4      R ⟵ GetRule(c);
5      foreach g ∈ G do
6          foreach c' ∈ GetSegment(g) do
7              R'.Append(GetRule(c'));
8          if R ∩ R' ≠ ∅ then
9              g.Append(c);
10         else
11             G.NewGroup().Append(c);
12 return G;
```

E. Redundancy Discovery and Removal

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



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

Fig. 9. Example of eliminating redundancies at policy level.

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

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

1) Redundancy Elimination at Policy Level: :

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

Authorization Space Segmentation: We first perform the policy segmentation function `Partition_P()` defined in Algorithm 3 to divide the entire authorization space of a policy into disjoint segments. We classify the policy segments in following categories: *non-overlapping* segment and *overlapping* segment, which is further divided into *conflicting overlapping* segment and *non-conflicting overlapping* segment. Each *non-overlapping* segment associates with one unique rule and each *overlapping* segment is related to a set of rules, which may conflict with each other (*conflicting overlapping* segment) or have the same effect (*non-conflicting overlapping* segment). Figure 9(a) illustrates a grid representation of authorization space segmentation for a policy with eight rules. In this example, two policy segments s_4 and s_6 are *non-overlapping* segments. Other policy segments are *overlapping* segments, including three *conflicting overlapping* segments s_1, s_3 and s_5, and one *non-conflicting overlapping* segments s_2.

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

Process1: *Property assignment for the rule subspace covered by a non-overlapping segment*. A non-overlapping segment contains only one rule subspace. Thus, this rule subspace is assigned with *strong irremovable* property. Other rule subspaces associated with the same rule are assigned with *weak irremovable* property, excepting the rule subspaces that already have *strong irremovable* property.

Process2: *Property assignment for rule subspaces covered by a conflicting segment*. We present this property assignment process based on the following three cases of rule combining algorithm ($\mathcal{CA}$).

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

Process3: *Property assignment for rule subspaces covered by a non-conflicting overlapping segment*. If any rule subspace has been assigned with *weak irremovable* property, other rule subspaces without *irremovable* property are assigned with *removable* property. Otherwise, all subspaces within the segment are assigned with *correlated* property.

Figure 9(b) shows the result of applying our property assignment mechanism to the example presented in Figure 9(a). We can easily identify that r_3 and r_8 are *removable* rules, where all subspaces are with *removable* property. However, we need to further examine the *correlated* rules r_2, r_4 or r_7, which contain some subspaces with *correlated* property.

Rule Correlation Break and Redundancy Removal: Rule subspaces covered by an overlapping segment are correlated with each other when the effect of overlapping segment can be determined by any of those correlated rules. Thus, keeping one correlated rule and removing others may not change the effect of corresponding segment. We call such a correlated relation *vertical rule correlation*, which can be identified in property assignment step. In addition, we observe that some rule may be involved in several correlated relations. For example, in Figure 9(b), r_4 has two subspaces that are involved in the correlated relations with r_2 and r_7, respectively. We call this kind of correlated relation *horizontal rule correlation*. Obviously, we cannot resolve a correlation individually and those two dimensions of rule correlation should be take into consideration. Therefore, we can further construct rule correlation groups based on those two kinds of rule correlations so that dependent relationships among multiple correlated rules within one group can be examined together. For example, a correlation group g consisting of three rules r_2, r_4 and r_7 can be identified in Figure 9(b) based on two dimensions of rule correlation.

We additionally observe that different sequences to break rule correlations in a correlation

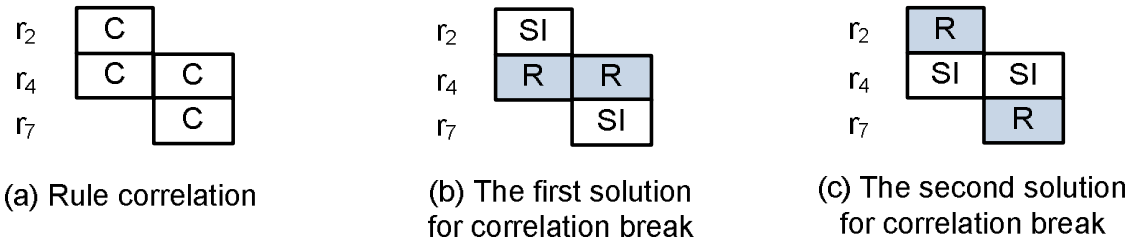


Fig. 10. Example of rule correlation break.

Algorithm 6: Redundancy Elimination of Policy P: RedundancyEliminate_P(P)

Input: A policy P with a set of rules.
Output: A redundancy-eliminated policy $P^{'}$.

```
/* Partition the entire authorization space of P into disjoint spaces*/
S.New();
S ⟵ Partition_P(P);
/* Property assignment for all rule subspaces */
PropertyAssgin_P(S);
/* Rule correlation break */
G ⟵ CorrelatonGroupConstruct(S);
foreach g ∈ G do
    foreach r ∈ g do
        r.CD ⟵ Σ_{s_i ∈ CS(r)} 1/(NC(s_i)−1);
    SP ⟵ GetCorrelatedSubspace(MinCDRule(g))
    foreach sp ∈ SP do
        sp.Property ⟵ R ;
        if |GetCorrelatedSubspace(sp)| = 1 then
            SP' ⟵ GetCorrelatedSubspace(sp);
            SP'.Property ⟵ SI ;
            AssginSI(SP');
/*Redundancy removal */
P' ⟵ P;
foreach r ∈ P' do
    if AllRemovalProperty(r) = true then
        P' ⟵ P' \ r;
return P';
```

group may lead to different results for redundancy removal. Figure 10(a) shows correlated relations of rules r_2, r_4 and r_7 in the correlation group g. We can break their correlation relations via different sequences. Figure 10(b) shows one possible solution. If we first assign two subspaces of r_4 with *removable* property, r_4 becomes a *removable* rule but r_2 and r_7 are turned to an *irremovable* rules. However, regarding another solution represented in Figure 10(c), if we first assign the correlated subspace of r_2 with *removable* property, then only r_4 becomes an *irremovable* rule. Both r_2 and r_7 are *removable*. Thus, it is necessary to seek an optimal solution to obtain *maximum* redundancy removal. To achieve this goal, we can compute a correlation degree (CD) for each correlated rule r using the following equation:

$$CD(r) = \sum_{s_i \in CS(r)} \frac{1}{NC(s_i) - 1} \tag{2}$$

Note that $CS(r)$ is a function to return all correlated segments of a rule r, and $NC(s_i)$ is a function to return the number of correlated rules within a segment s_i. Since each policy segment contains multiple correlated rules ($NC(s_i) \geq 2$), $\frac{1}{NC(s_i)-1}$ gives the degree of breakable correlation relations associated with a policy segment s_i if we set a rule r as *removable*. To maximize the number of removable rules for redundancy resolution, our correlation break process

selects one rule with the minimal CD as the candidate removable rule each time. For instance, applying this equation to calculate correlation degrees of three rules demonstrated in Figure 10(a), $CD(r_2)$ and $CD(r_7)$ equal to 1, and $CD(r_4)$ equals to 2. Thus, we can select either r_2 or r_7 as the candidate removable rule in the first break step. Finally, two rules r_2 and r_7 become removable rules after breaking all correlations.

F. Redundancy Elimination at Policy Set Level

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

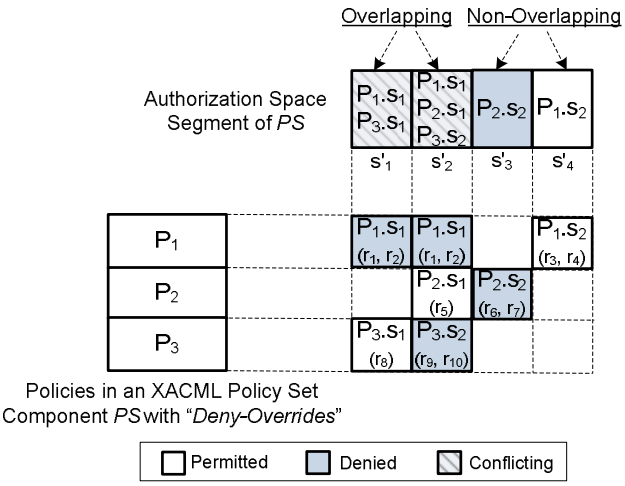


Fig. 11. Example of authorization space segmentation at policy set level for redundancy discovery and removal.

After each component of a policy set PS performs redundancy removal, the authorization space of PS can be then partitioned into disjoint segments by performing `Partition()` function. Note that, in the solution for conflict detection at policy set level, we aggregate authorization subspaces of each child node before performing space partition, because we only need to identify conflicts among children nodes to guide the selection of policy combining algorithms for the policy set. However, for redundancy removal at policy set level, both redundancies among children nodes and rule (leaf node) redundancies, which may exist across multiple policies or policy sets, should be discovered. Therefore, we keep the original segments of each child node and leverage those segments to generate the authorization space segments of PS. Figure 11 demonstrates an example of authorization space segmentation of a policy set PS with three children components P_1, P_2 and P_3. The authorization space segments of PS are constructed based on the original segments of each child component. For instance, a segment $s_2^{'}$ of PS covers three policy segments $P_1.s_1$, $P_2.s_1$ and $P_3.s_2$, where $P_i.s_j$ denotes that a segment s_j belongs to a policy P_i.

The property assignment step at policy set level is similar to the property assignment step at policy level, except that the policy combining algorithm *Only-One-Applicable* needs to be taken into consideration at policy set level. The *Only-One-Applicable* case is handled similar to the *First-Applicable* case. We first check whether the combining algorithm is applicable or not.

Algorithm 7: Eliminate Redundancies of a Policy Set PS: RedundancyEliminate_PS(PS)

Input: A policy set PS with a set of policies or other policy sets.
Output: A redundancy-eliminated policy set $PS^{'}$.

$E \longleftarrow GetChild(PS)$;
foreach $e \in E$ **do**
  /* e *is a policy*/
  if $IsPolicy(e) = true$ **then**
    $e^{'} \longleftarrow$ **RedundancyEliminate_P**(e);
  /* e *is a policy set*/
  else if $IsPolicySet(e) = true$ **then**
    $e^{'} \longleftarrow$ **RedundancyEliminate_PS**(e)
  $E^{'} \longleftarrow E^{'} \cup e^{'}$;
/* *Partition the authorization space of* PS *into disjoint spaces*/
foreach $e^{'} \in E^{'}$ **do**
  foreach $s \in GetSegment(e^{'})$ **do**
    $S \longleftarrow$ **Partition**(S, s);
/* *Property assignment for all subspaces covered by the segments of* PS */
PropertyAssgin_PS(S);
/* *Correlation break* */
CorrelationBreak_PS(S);
/*Redundancy removal for child components of PS */
$PS^{'} \longleftarrow PS$;
foreach $e \in PS$ **do**
  if $AllRemovalProperty(e) =$ *true* **then**
    $PS^{'} \longleftarrow PS^{'} \setminus e$;
/*Redundancy removal for rules of PS */
foreach $e \in PS^{'}$ **do**
  $S \longleftarrow GetSubspace(e)$;
  foreach $s \in S$ **do**
    if $s.Property = R$ **then**
      $S \longleftarrow GetSubspaceWithSI(s)$;
      foreach $s \in S$ **do**
        if $OneSISubspace(GetRule(s)) =$ *true* **then**
          $SP \longleftarrow GetRuleSubspace(GetRule(s)) \setminus s$;
          foreach $sp \in SP$ **do**
            $sp.Property \longleftarrow$ R ;
      $SP^{'} \longleftarrow GetSubspace(s)$;
      foreach $sp' \in SP'$ **do**
        $sp'.Property \longleftarrow$ R ;
foreach $r \in SP^{'}$ **do**
  if $AllRemovalProperty(r) =$ *true* **then**
    $SP^{'} \longleftarrow SP^{'} \setminus r$;
return $PS^{'}$;

If the combining algorithm is applicable, the only-applicable subspace is assigned with *strong irremovable* property. Otherwise, all subspaces within the policy set's segment are assigned with *removable* property.

We utilize a similar correlation break mechanism introduced previously to break the correlation relations among the segments of child components of PS. Since there may exist multiple correlated segments of children components with the minimal correlation degree (CD) value, we additionally compute a removal value (RV), which indicates the possibility of rule redundancy removal if we set a correlated segment with *removable* property, for all candidate segments using following equation:

$$RV(s) = \sum_{r_i \in R_{HI}(s)} \frac{1}{N_{HI}(r_i)} \quad (3)$$

Note that $R_{HI}(s)$ is a function to return all rules having a subspace in the segment s with *strong irremovable* property, and $N_{HI}(r_i)$ is a function to return the number of the rule's subspace with *strong irremovable* property. $\frac{1}{N_{HI}(r_i)}$ measures the possibility of turning a rule to a removable rule if we change a *strong irremovable* rule subspace to be *removable*. The correlated segment with the maximum RV value has the highest priority to be chosen for breaking correlations.

After assigning properties to all segments of children components of PS, we next examine whether any child component is redundant. If a child component is redundant, this child component and all rules contained in the child component are removed from PS. Then, we examine whether there exist any redundant rules. In this process, the properties of all rule subspaces covered by a *removable* segment of a child component of PS needs to be changed to *removable*. Note that when we change the property of a *strong irremovable* rule subspace to *removable*, other subspaces in the same rule with dependent *weak irremovable* property need to be changed to *removable* correspondingly. Algorithm 7 shows the pseudocode of eliminating redundancies for a policy set PS.

G. Implementation and Evaluation

We have implemented a policy analysis tool called `XAnalyzer` in Java. Based on our policy anomaly analysis mechanism, it consists of four core components: segmentation module, effect constraint generation module, strategy mapping module, and property assignment module. The segmentation module takes XACML policies as an input and identifies the authorization space segments by partitioning the authorization space into disjoint subspaces. `XAnalyzer` utilizes APIs provided by *Sun* XACML implementation [43] to parse the XACML policies and construct Boolean encoding. JavaBDD [20], which is based on BuDDy package [12], is employed by `XAnalyzer` to support BDD representation and authorization space operations. The effect constraint generation module takes conflicting segments as an input and generates effect constraints for each conflicting segment. Effect constraints are generated based on strategies assigned to each conflicting segment. The strategy mapping module takes conflict correlation groups and effect constraints of conflicting segments as inputs and then maps assigned strategies to standard XACML combining algorithms for examined XACML policy components. The property assignment module automatically assigns corresponding property to each subspace covered by the segments of XACML policy components. The assigned properties are in turn utilized to identify redundancies.

Considering the complexity of tasks involved in the policy analysis, it is desirable to provide intuitive user interfaces for policy designers or administrators for effective policy anomaly detection and resolution. Since the grid representation of policy anomalies offers a succinct view of the interactions of overlapping rules and enables policy designers or administrators to better understand policy anomalies, we implemented the grid representation of policy anomalies in `XAnalyzer`.

`XAnalyzer` provides two policy viewers, *Conflict Viewer*and *Redundancy Viewer*, to visualize the outputs of policy conflict analysis and policy redundancy analysis, respectively. In addition, `XAnalyzer` offers flexible ways to handle large-scale policies. There are two kinds of visualization interfaces in each viewer: one interface shows an entire snapshot of all anomalies; another interface shows a partial snapshot only containing anomalies within one correlation group. In

addition, XAnalyzer shows a hierarchical structure of policies and allow policy designers or administrators to view policy anomalies at different levels independently. The hierarchical structure of policies is presented by a tree of policy components on the left side of the interfaces. A policy designer or administrator can choose a particular policy component such as *Policy* or *Policy Set* node for anomaly analysis. If an administrator chooses a *Policy* node for the analysis, anomalies in that particular *Policy* node are displayed in terms of the rule subspaces involved. Otherwise, if an administrator chooses a *Policy Set* node, all anomalies within that particular node are displayed in terms of *allowed* and *denied* subspaces of policy or policy set components.

We evaluated the efficiency and effectiveness of XAnalyzer for policy analysis on both real-life and synthetic XACML policies. Our experiments were performed on Intel Core 2 Duo CPU 3.00 GHz with 3.25 GB RAM running on Windows XP SP2. In our evaluation, we utilized five real-life XACML policies, which were collected from different sources. Three of the policies, *CodeA, Continue-a* and *Continue-b* are XACML policies used in [16]; among them, *Continue-a* and *Continue-b* are designed for a real-world Web application supporting a conference management. *GradeSheet* is utilized in [8]. The *Pluto* policy is employed in ARCHON system, [‡] which is a digital library that federates the collections of physics with multiple degrees of meta data richness. Since it is hard to get a large volume of real-world policies due to the reason that they are often considered to be highly confidential, we generated four large synthetic policies *SyntheticPolicy-1*, *SyntheticPolicy-2*, *SyntheticPolicy-3* and *SyntheticPolicy-4* for further evaluating the performance and scalability of our tool. These synthetic policies are multi-layered, where each policy component has a randomly selected combining algorithm and each rule has randomly chosen attribute sets from a predefined domain. We also use *SamplePolicy*, which is the example XACML policy represented in Figure 3, in our experiments. Table II summarizes the basic information of each policy including the number of rules, the number of policies, and the number of policy sets.

TABLE II

XACML POLICIES USED FOR EVALUATION.

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

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

Evaluation of Conflict Detection: Time required by XAnalyzer for conflict detection highly depends upon the number of segments generated for each XACML policy. The increase of the number of segments is proportional to the number of components contained in an XACML policy. From Table III, we observe that XAnalyzer performs fast enough to handle larger size

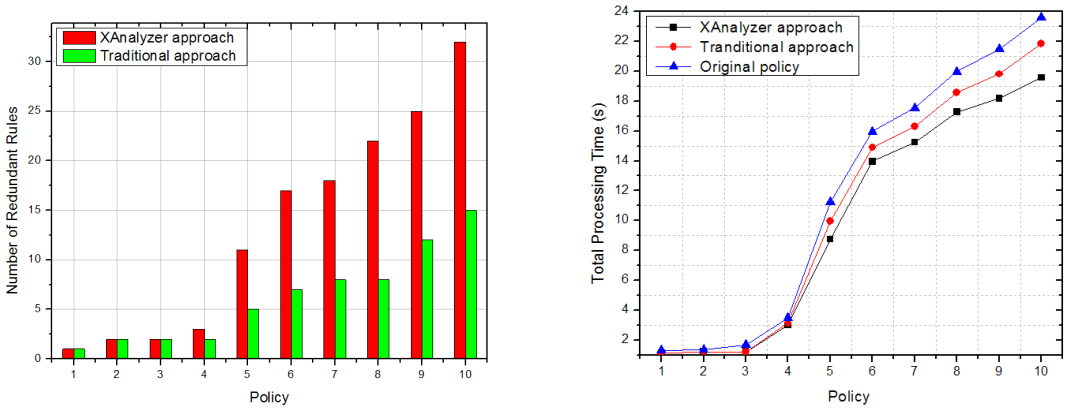
[‡]http://archon.cs.odu.edu/

TABLE III

CONFLICT DETECTION AND REDUNDANCY REMOVAL ALGORITHMS EVALUATION.

Policy	Partitions (#)	BDD Nodes (#)	Conflict Detection			Redundant Removal		
			Policy Level(#)	Policy Set Level(#)	Time (s)	Policy Level(#)	Policy Set Level(#)	Time (s)
1 (CodeA)	6	16	1	1	0.082	1	0	0.087
2 (SamplePolicy)	8	34	0	2	0.090	0	2	0.095
3 (GradeSheet)	18	45	0	4	0.098	0	2	0.113
4 (Pluto)	34	78	0	5	0.136	0	3	0.147
5 (SyntheticPolicy-1)	205	112	8	14	0.329	7	4	0.158
6 (Continue-a)	439	135	9	17	0.583	11	7	0.214
7 (Continue-b)	468	146	10	21	0.635	12	7	0.585
8 (SyntheticPolicy-2)	523	209	29	17	0.896	14	8	0.623
9 (SyntheticPolicy-3)	614	227	39	19	0.948	17	10	0.672
10 (SyntheticPolicy-4)	814	265	56	19	1.123	23	12	0.803

XACML policies, even for some complex policies with multiple levels of hierarchies along with hundreds of rules, such as two real-life XACML policies *Continue-a* and *Continue-b* and four synthetic XACML policies. The time trends observed from Table III are promising, and hence provide the evidence of efficiency of our conflict detection approach.



(a) Redundancy elimination rate

(b) Performance improvement

Fig. 12. Evaluation of redundancy removal approach.

Evaluation of Redundancy Removal: In the second set of experiments, we evaluated our redundancy analysis approach based on those experimental XACML policies. The evaluation results shown in Table III also indicate the efficiency of our redundancy analysis algorithm. Moreover, we conducted the evaluation of effectiveness by comparing our redundancy analysis approach with *traditional* redundancy analysis approach [30], [5], which can only identify redundancy relations between *two* rules. Figure 12(a) depicts the results of our comparison experiments. From Figure 12(a), we observed that `XAnalyzer` could identify that an average of 6.3% of total rules are redundant. However, *traditional* redundancy analysis approach could only detect an average 3.7% of total rules as redundant rules. Therefore, the enhancement for redundancy elimination was clearly observed by our redundancy analysis approach compared to *traditional* redundancy analysis approach in our experiments.

Furthermore, when redundancies in a policy are removed, the performance of policy enforcement is improved generally. For each of XACML policies in our experiments, Figure 12(b) depicts the total processing time in *Sun* XACML PDP [43] for responding 10,000 randomly generated XACML requests. The evaluation results clearly show that the processing times are

reduced after eliminating redundancies in XACML policies applying either *traditional* approach or our approach, and our approach can obtain better performance improvement than *traditional* approach.

H. Related Work

Many research efforts have been devoted to XACML. However, most existing research work focus on modeling and verification of XACML policies [16], [10], [24], [4], [28]. None of them dealt with anomaly analysis in XACML policies. We discuss a few of those work here.

In [10], the authors formalized XACML policies using a process algebra known as Communicating Sequential Processes (CSP). This work utilizes a model checker to formally verify properties of policies, and to compare access control policies with each other. Fisler et al. [16] introduced an approach to represent XACML policies with Multi-Terminal Binary Decision Diagrams (MTBDDs). They developed a policy analysis tool called Margrave, which can verify XACML policies against the given properties and perform change-impact analysis. Lin et al. [28] investigated the problem of similarity analysis for XACML policies, integrating the SAT-solver-based and MTBDD-based techniques. Ahn et al. [4] presented a formalization of XACML using answer set programming (ASP), which is a recent form of declarative programming, and leveraged existing ASP reasoners to conduct policy verification.

Several work presented policy analysis tools with the goal of discovering policy anomalies in firewall [5], [47], [18]. However, we cannot directly apply those analysis approaches for XACML due to several reasons. First, the structure of firewall policies is flat but XACML has a hierarchical structure supporting recursive policy specification. Second, a firewall policy only supports one conflict resolution strategy (*first-match*) but XACML has four rule/policy combining algorithms. Last but not the least, a firewall rule is typically specified with fixed fields, while an XACML rule can be multi-valued.

Some XACML policy evaluation engines, such as *Sun* PDP [43] and XEngine [29], have been developed to handle the process of evaluating whether a request satisfies an XACML policy. During the process of policy enforcement, conflicts can be checked if a request matches multiple rules having different effects, and then conflicts are resolved by applying predefined combining algorithms in the policy. In contrast, our tool `XAnalyzer` focuses on policy analysis at policy *design* time. `XAnalyzer` can identify all conflicts within a policy and help policy designers select appropriate combining algorithms for conflict resolution prior to the policy enforcement. Additionally, `XAnalyzer` has the capability of discovering and eliminating policy redundancies that cannot be dealt with by policy evaluation engines.

Some work addressed the general conflict resolution mechanisms for access control [19], [17], [27], [21], [16]. Especially, Li et al. [27] proposed a policy combining language `PCL`, which can be utilized to specify a variety of user-defined combining algorithms for XACML. These conflict resolution mechanisms can be accommodated in our fine-grained conflict resolution framework. In addition, Bauer et al. [7] adopted a data-mining technique to eliminate *inconsistencies* between access control policies and user's intentions. In comparison, our approach detects and resolves anomalies within access control policies caused by overlapping relations.

Other related work includes XACML policy integration [32], [35] and XACML policy optimization [31]. Since anomaly discovery and resolution are challenging issues in policy integration and redundancy elimination can contribute in policy optimization, all of those related work are orthogonal to our work.

III. Research Task: Federated Access Management for Collaborative Network Environments

In ABAC, a given access control request, e.g. reading a confidential data file, is granted upon the satisfaction of constraints involving some security-relevant properties, also known as *attributes*, that are exhibited by the access control *entities* involved in the request. Commonly, such access control entities include *actors*, i.e., human agents or computer processes running on behalf of them; *targets*, i.e., protected resources such as data files or network ports, and any applicable *context*, i.e., the running environment where a given software system is executed, such as an operative system or a cloud setting. In this section, we introduce a simple use case example in the collaboration domain that involves the specification of access control constraints based on attributes obtained from different entities, in an effort to properly identify how attributes are used for defining constraints and how constraints are in turn related to access rights (permissions). For illustrative purposes, consider a setting when *actors* may request access to a *shared* data file. Such an access right is represented by a permission named *readFilePermission*, which is only granted if the requesting actor happens to be the *owner* of the file, or if the file is labeled for sharing purposes (by naming the file as "*shared.txt*"). Such a constraint depicts a collaboration setting when an access to a shared resource is granted to a collaborative actor only for working hours. A corresponding use case is depicted in Fig. 13: the constraint C_1 restricts access to a given file only if its *name* attribute has a value equal to the "*shared.txt*" and the *time* attribute exhibited by the context environment fits within a certain range. Moreover, the access control constraint C_2 restricts access to files only when the *name* attribute presented by the requesting actor is the same as the *ownername* attribute defined for the requested file. This use case depicts a scenario when attributes are retrieved from all the involved access control entities, and several constraints are defined in order for the *readFilePermission* to be granted. Based on this sample case, some noticeable patterns emerge: first, different attributes exhibited by different access control entities may be used to define constraints, and not all of the entities involved may need to provide attributes in order for an access control decision to be made, as noticed in the C_1 constraint, where the entity requesting access is required to show no attributes for the constraint guarding the requested permission. Second, the same set of attributes exhibited by a given access control entity may be used to define more than one constraint, as shown in C_1 and C_2, where attributes from the *target* are used to define two different constraints. Third, constraints may be in turn composed of different *sub-constraints* and may be evaluated simultaneously while making an access control decision (C_1). Finally, many different constraints, which may be constructed upon different attributes from different access control entities, may be used to guard access to the same permission. The patterns discussed above provide an insight on how flexible an scheme based on attributes and constraints should be for modeling access control requirements: as many different options are available, policy designers may choose the ones that better fit the needs devised for a given collaboration setting. However, at the cost of flexibility, we must also consider such diversity may also cause unexpected challenges when analyzing some non-trivial security properties of a given ABAC setting. As an example, consider the case of performing a *risk analysis* [42] on the ABAC setting depicted in Fig. 13: in order to determine under which situations *readFilePermission* may be granted. Such an analysis may involve the analysis of each constraint guarding the *readFilePermission*, and attributes on each entity, including the range of possible values such attributes may take and the number of possible combinations that may be obtained from them. Such a process gets complicated by the fact that no standardized definition of both attributes and constraints exists in the literature, nor a standardized policy language

for a reference ABAC model exists. These problems may turn the security analysis process cumbersome in real-life systems where diverse constraints may be in place for guarding a given permission.

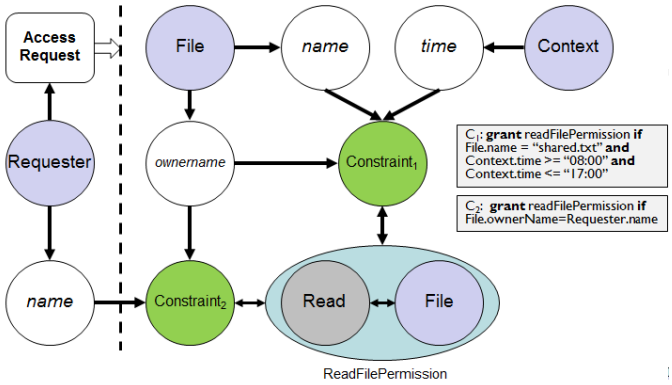


Fig. 13. A combined use case depicting ABAC in the collaboration domain.

A. Model Definition

1) General Description: In order to support access control patterns depicted in Section III, we first define the main components of our proposed model:

- *actors* are users (i.e. human agents) or subjects (i.e. computer processes) acting on behalf of users;
- *targets* are the protected resources within a software system;
- *requests* are initiated by actors when access to a specific target is needed; and
- *context* is the running (executing) environment, e.g. operative system, supporting platform, etc., where a given action is issued and/or served.

Besides the core ABAC component *attributes*, we also introduce the concept of *security tokens*: first-class objects that are intended to provide an abstract representation of a set of security states that may be relevant to the domain-specific properties in collaborative systems. Moreover, these security tokens are helpful to represent the attributes exhibited by access control entities. Attributes are related to security tokens through *token provisioning functions* (TP-Functions), which map each attribute to a corresponding security token, and are intended to model the constraints shown in Section III. They also produce security tokens as a result and allow to define the relationship between permissions and attributes. It must be noticed that the precise definition of such TP-Functions falls in the scope of the collaborative system domain. Fig. 14 shows a visual representation of our proposed model, which leverages a similar representation shown in [38]. In the diagram, the sets of either access control entities or elements (e.g. attributes) are modeled as circles connected by either unidirectional or bidirectional arrows, which can also be single-headed or double-headed. Unidirectional arrows depict functions, whereas bidirectional ones are used for relations. For instance, attributes are related to its corresponding access control entity by means of the *attribute assignment* (AA) relation. Moreover, double-headed arrows depict multiplicity of appearances of members of sets (denoted by circles) in the relation depicted by the arrow, whereas single-headed ones may depict a single appearance. As an example, the AA relation depicts an access control entity is related to several attributes, whereas an attribute may be related only to a single access control entity. Attributes are also related to security tokens

through our proposed TP-Functions, which are depicted as unidirectional arrows in the diagram. Moreover, TP-Functions also relates security tokens with other security tokens. Following this approach, a security token that is produced by means of a TP-Function is *provisioned*. Security tokens are related to access rights (permissions) by means of the *permission assignment* (PA) relation. Permissions are in turn depicted as a combination of a protected source (target) and an operation that can be performed on it. Following the convention for double-headed arrows introduced earlier, a security token may be related to one or more permissions, and a given permission may be related to one or more security tokens. Finally, it must be noticed that our model is intended to be *independent* of any supporting technology or methodology other than the concepts we have enlisted in this section.

2) Definition of Attributes: As described previously, attributes are defined to be *security-relevant* properties that are exhibited by access control entities, namely, actors, targets, policies and any applicable context. Their physical nature and the way those attributes are collected from the access control entities remains dependent on the collaborative application domain. From the use case and patterns discussed in Section III, attributes are expected to have at least the following inner components: an *identifier* (*id* for short), which is later used for defining constraints on them, e.g. the attribute *name* defined for a given file (target); a *value*, which is used when evaluating constraints, e.g. "*shared.txt*"; and a data *type*, which restricts the nature and the range of the value defined for the attribute, e.g. the data type *String*. In addition, for brevity, we also introduce the concept of *attribute families*, which are the sets of attributes where all members share the same *type* and *id* components. With this in mind, we proceed with the following definition:

Definition 1: An *attribute* is a 3-tuple of the form <*type*, *id*, *value*>, where

- *type* is a well-defined data type relevant to the collaborative application domain, e.g. String, Integer, Date, Boolean, etc.
- *id* is a unique identifier, which is required to be non-empty for the purpose of an access control decision. The format of the *id* component, as well as its associated semantics, is based on the scope of the collaborative application domain.
- *value* is a data value of the attribute in the nature and range defined by *type*.

Illustrative examples of attributes are as follows: <String, *file.name*, "*shared.txt*">, <String, *actor.name*, "*Carlos*">, <Integer, *actor.age*, 100>, <Date, *system.date*, "*10-10-3*">, <Boolean, *file.isOpen*, true>, etc.

Definition 2: An *attribute family* is a set of attributes where all elements share the same *type* and *id* components.

The number of possible elements of an attribute family is bounded by the range of values defined by the *type* component. In the rest of this report, we use the term *fam-a*(*id*) to refer to a family of attributes identified by a given *id* component.

3) Definition of Security Tokens: As described previously, we aim to provide a description of the main features of ABAC by introducing the concept of *security tokens*. Informally, such tokens are obtained by *processing* the attributes from access control entities through the aforementioned TP-Functions, and are intended to provide an abstract representation of the security states within a software system (e.g. a collaborative application), on which access rights can be granted *safely*. As an example, TP-Functions may provide functionality intended to validate a given attribute, by inspecting its value component and producing a proper security token as a result. Thus, a validated attribute may put the system in a secure state, and access rights can be safely granted as a result. As described in Section III-A.1, permissions can be assigned to security tokens,

which then serve as a layer of association between attributes and permissions. Such a layer helps identify the attributes that may be ultimately involved in granting a given permission, as well as the set of constraints represented by TP-Functions that may be involved in such process. Moreover, our model also allows for TP-Functions to take security tokens as an input, or may also take both attributes and security tokens as an input altogether to produce security tokens as a result, as depicted in Fig. 14. We define *security token* as follows:

Definition 3: A *security token* is a tuple of the form <*id*, *value*>, where *id* is a unique identifier, whose format as well as associated semantics remain in the scope of the collaborative application domain, and *value* is the devised value of the token, if any, which may also be in the scope of the application domain, and may not necessarily *match* a given value depicted by the attribute (or set of attributes) the token is originated from.

Note that our definition of security tokens does not include the concept of data type, which is included in the definition of attributes, in order to allow for an enhanced flexibility in the definition of the range of values. Examples of security tokens include: <*jobPosition*, manager>, <*jobPosition*, employee>, <*isUnderage*, true>, <*isSuperUser*, false>, etc. With attributes, security tokens are grouped into token families, which have a similar definition to the ones defined for attributes. A token family groups a set of security tokens that share the same *id* component as follows:

Definition 4: A *security token family* is a set of security tokens where all members share the same *id* component.

As with attribute families, we denote a security family identified by *id* as *fam-t*(*id*). Examples of security token families include: *fam-t*(*jobPosition*) = {<*jobPosition*, manager>, <*jobPosition*, employee>} and *fam-t*(*isUnderage*) = {<*isUnderage*, true>, <*isUnderAge*, false>}.

4) Definition of Token-Provisioning Functions: As introduced in Section III-A.1, our TP-Functions represent a core component of our model that is intended to provide a conceptual foundation to the constraints depicted in the use case discussed in Section III. TP-Functions are expected to provide a strong mapping between a given attribute defined in an attribute family and a single security token defined in a security token family. On the other hand, not all tokens in a given security token family are expected to be originated from an attribute through a given TP-Function. Fig. 15 shows a representation of a pair of TP-Functions mapping attributes from the attribute family identified by *fam-a*(*a*) to security tokens in the token families *fam-t*(*t*) (TPF_1) and *fam-t*(*t'*) (TPF_2). Fig. 15 illustrates two interesting properties of our proposed TP-Functions: first, TP-Functions are said to be *non-injective*[§], as two or more elements from a TP-Function input set (domain) may be mapped to the same element in the output set (codomain). As an example, consider the case of attributes a_1 and a_2, from the attribute family *fam-a*(*a*), shown in Fig. 15, which are both mapped to the same element t_1 in *fam-t*(*t*) by TPF_1. Second, TP-Functions are said to be *non-surjective*[¶], as none of elements in the output (codomain) set are required to have a corresponding element in the input (domain) set. As an example, Fig. 15 shows the security token t_3 belonging to the *fam-t*(*t*) family has no originating attribute from *fam-a*(*a*) by TPF_1.

Definition 5: A *token-provisioning function* (TP-Function) is a *non-bijective* as well as *non-surjective*, mapping sets of attribute families and security token families to security token families.

The combination of families of attributes and security tokens as an input to TP-Functions can be achieved by taking the cartesian product of all the families involved. Moreover, as depicted in Section III-A.1, security tokens are associated with access rights (permissions) by means of the

[§]A function f: A $\rightarrow$ B is said to be *injective* or *one-to-one*, if $\forall\ a, a' \in$ A, $f(a) \neq f(a')$.

[¶]A function f: A $\rightarrow$ B is said to be *surjective* or *onto*, if $\forall\ b \in$ B, $\exists\ a \in$ A, $f(a) = b$.

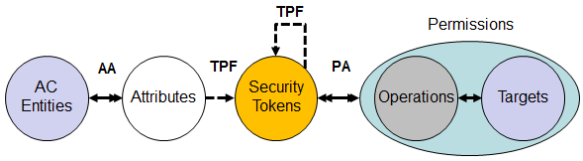


Fig. 14. Graphical depiction of a model based on *security tokens* for ABAC.

PA relation. As shown in Fig. 15, TP-Functions can be chained together to produce a graph-like structure showing how attributes and security tokens are used to produce security tokens. Such a structure is defined as follows:

Definition 6: A *token-provisioning graph* (TP-Graph) is a directed, *weakly* connected, and possibly cyclic graph, whose *vertices* (*nodes*) represent either families of attributes or families of security tokens, and its *edges* (*arcs*) represents token-provisioning functions (TP-Functions).

TP-Graphs are *directed*, since TP-Functions represent unidirectional edges due to their nature. Moreover, TP-Graphs are also *weakly* connected, as there is no requirement for all nodes (families) to be connected to each other. Finally, TP-Graphs are also possibly *cyclic*. Leveraging this definition, security tokens are obtained by *traversing* a given TP-Graph. Fig. 15 shows an example of a TP-Graph that has been annotated with a permission, which is in turn associated with a given security token t_1. The process of granting the depicted *readFilePermission* is summarized as follows: at runtime, the access control entities involved in the request must exhibit attributes a_1 and a_2. An attribute a_1 is turn into the security token t_1 by means of the TPF_1 function. Token t_1, in combination with attribute a_1 (from family *fam-a*(a)) is also turned into the security token t'_1, which is related to the aforementioned *readFilePermission* by means of an entry in the PA relation. As explained in Section III-A.1, a permission is granted if a requesting actor is able to *provision* a token that is associated with the PA relation. Following Definition 6, security tokens are *provisioned* by traversing a TP-Graph. Besides providing a stronger definition of the way security tokens are generated within our model, TP-Graphs allow for the development of security analysis techniques based on graph theory. Finally, the process of applying TP-Functions to attributes/security tokens (by traversing a given TP-Graph) is named as *token provisioning*. We also call the set of security tokens that can be potentially provisioned from a TP-Graph as the *token repository*.

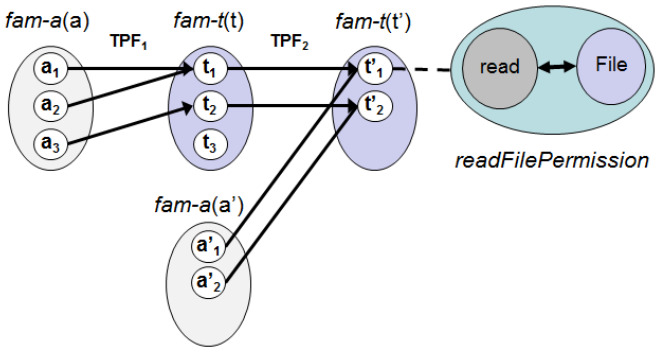


Fig. 15. An example of TP-Functions, token families and an associated permission.

B. Evaluation

1) Case Study: In order to illustrate the access control model proposed in Section III-A, we performed a case study with a *document management system* (DMS). Such an application is intended for end-users to store and share document files, e.g. text files, pictures, etc., while still enforcing a set of access control rules to protect the confidentiality and integrity of the documents managed by the system. Users are allowed to upload documents into the system, in such a way documents are *owned* by the user who uploaded them in the first place. Users are also allowed to *share* documents they own with other users by specifying the following access modes: *read* (R), *write* (W) and *execute* (X). As an example, a user u_1 may share a document *d* with another user u_2 by granting him/her the *read* access mode. Therefore, u_2 is allowed to inspect *d*, but not to change its contents. In addition, three different categories of end-users are defined, namely: *Employee*, *Departmental Manager* and *CEO*. Users with the *Employee* category are grouped in departments led by one or more users holding the *Department Manager* category, which in turn oversee users with the *CEO* category. The access control requirements for each category are as follows:

- Users with the *Employee* category share *full* access privileges (RWX) only with their corresponding *Department Manager* and the *CEO* by default.
- Users with the *Department Manager* category share *full* access privileges with the *CEO* as well, but not with *Employee* .
- Users with the *CEO* category share no access privileges with any other users by default.

Finally, the access control privileges defined for document *owners* remain intact despite the job category defined for a given user. This case study shows a couple of interesting features in terms of access control: first, it clearly depicts an approach involving access control constraints based on attributes. Second, derived from the aforementioned set of access control requirements based on organizational jobs, an access control model depicting the well-known *role-based access control* (RBAC) [38] immediately comes to mind. However, as depicted in [25], applying attributes (along with their corresponding access control constraints) to an RBAC setting is not a straightforward task. With this in mind, our case study presents an interesting alternative by leveraging an ABAC setting modeled with the approach we have described in previous sections.

Fig. 16 shows a representation of a model configuration depicting our case study. The set of access control entities includes the set of users of the DMS (actors), as well as all the documents that are uploaded into the system (targets). The operations include the ones described earlier and the set of permissions is specified by relating the operations with the set of documents uploaded to the DMS. Attributes can be grouped into the following families: *userID*, *jobCategory* and *deptID*, which are related to the set of users, and *docID*, *ownerID* and *sharingModes*, which are related to documents. Families of security tokens include *userID*, *docID*, *deptID*, *jobCategory*, *docID*, *docID*, *ownerID* and *ownerDeptID*, which are obtained from processing the corresponding attributes by means of the TP-Functions f_1, f_2, f_3, f_4, f_5 and f_7 respectively. In addition, the families of security tokens *shared-Owner*, *shared-Dept-Manager*, *shared-CEO* and *shared* represent the set of security states in the DMS where permissions can be safely granted, and are produced by functions f_6, f_8, f_9 and f_{10}, which derive the access control constraints for our case study. As an example, the TP-Function identified by f_9 compares the security tokens *userID* and *ownerID* then returns the *shared-Owner* if their *value* components match. Otherwise, an *empty* security token is returned. If the *shared-Owner* is provisioned by a requesting actor, the associated permission p_{ALL}, which allows complete access to a document within the DMS, is granted as a result. A similar convention is applied in the definition of all other TP-Functions within our case study.

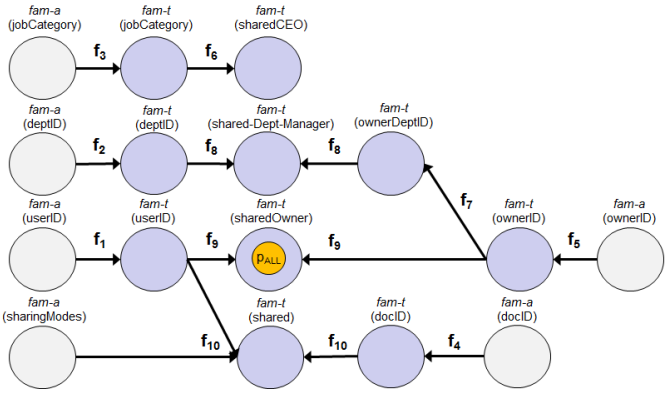


Fig. 16. A TP-Graph created from the TP-Functions depicted in our case study.

TABLE IV

A COMPARISON WITH EXISTING APPROACHES FOR ABAC.

Paper	Attributes	Constraints	Model Def.	Sec. State	Independence
[44]	√	√	√	×	×
[46]	×	×	×	×	×
[48]	√	×	√	√	√
[14]	√	×	×	×	×
[34]	×	×	√	×	×
[13]	×	√	√	×	×
[26]	√	√	×	×	×
[49]	√	×	×	×	×
[40]	√	√	×	×	×
[39]	×	×	×	√	×
[45]	×	×	√	×	×
[22]	√	×	√	×	√
Ours	√	√	√	√	√

2) Comparison with Existing Approaches: This section describes a brief comparison of our approach with existing approaches in the literature. We first articulate a comparison criteria based on the characteristics of ABAC: first, as we aim to provide a definition of the main ABAC component *attributes*, we check how this important component has been defined and considered in other existing approaches. Next, we explore the support offered by existing approaches in specifying access control constraints based on the concept of attributes, which we have described in Sections III-A.1 and III-A.4. Following with the same idea, we also compare our approach with existing ones with respect to the definition of the overall access control model. Next, we explore the support for the notion of *security state* (Sections III-A.1 and III-A.3) offered by the literature, in such a way a relationship between attributes and access rights are properly identified. Finally, as our approach is intended to be *independent* in terms of the supporting methodology, as described in Section III-A.1, we determine whether such a feature has been provided by other approaches. Using this comparison criteria, we examine existing approaches in the literature: one of the initial attempts in defining ABAC was proposed by Wang et al. [44], who explored the use of logic programming and computable set theory for modeling the main features of ABAC, taking into account a web-based context. Our work depicts a similar approach by proposing the evaluation of attributes through functions, but does not depend on any particular

supporting methodology for its implementation. Another approach was presented by Yuan et al. [46], who explored the use of ABAC for the specification of access control policies for web services. Their approach, while providing no formal definition of ABAC, depicts useful use cases that have influenced the ones described in Section III. Moreover, Priebe et al. [34] presented an approach leveraging the concepts of *ontologies* and the *semantic web* in order to formalize the notion of ABAC. Another approach leveraging the concept of the semantic web for ABAC include the one of Cirio et al. [13], who provide a model definition including attribute-based constraints. The work of Zhu and Smari [49] [40] has provided a definition of both attributes and attribute-based access control constraints tailored for supporting collaborative software systems. In the context of grid computing, the approach presented by Lang et al. [26] also provided an ABAC model mostly focused on the definition of attributes and the access control constraints. An approach close to ours was introduced by Covington and Sastry [14], who presented *contextual attribute access control* (CABAC) model which was realized in mobile applications. However, our approach takes a step further by describing the way such attributes are mapped to access rights (permissions) by means of TP-Functions and TP-Graphs. Recently, a noticeable approach was proposed by Jin et al. [22], whose approach is intended to formalize a series of ABAC model families. In addition, the relationship between ABAC and other well-known access control models was explored. We have been influenced by this work with respect to the definition of attributes as described in Section III-A.2. However, our approach introduces a notion of security token and TP-Functions to capture the mapping between attributes and corresponding access rights. Finally, another interesting approach was presented by Zhang et al.[48], who presented their *attribute-based access control matrix*, which extends classical theory in the field of access control to accommodate attributes as well as the notion of security state. However, it provides no definition for attribute-based constraints, which is considered in our approach by means of the proposed TP-Functions and TP-Graphs. Table IV summarizes our findings and comparison results.

IV. Concluding Remarks

In this report, we summarized our accomplishments. We first focused on access control and delegation models for ad-hoc collaboration including composite policies and schema integration. In addition, we attempted to detect unexpected and unauthorized information flows, and visualize our results in an effective manner. Also, we investigated how cryptographic approaches can enhance the integrity measurement in clouds. We also studied how integrity measurements of remote sites can be analyzed by policy analysis mechanisms and proposed a remote attestation framework. In addition, we focused on policy anomaly detection and resolution approaches for firewall policies and web-based policies. In addition, attempted to propose attribute-based access control in ad-hoc collaboration. Also, we investigated a formal way to describe identified characteristics and properties so that we can eventually formulate attribute-based access control model and relevant policy management schemes.

References

[1] Aws document, working with roles, `http://docs.amazonwebservices.com/IAM/latest/UserGuide/WorkingWithRoles.html`.

[2] Aws identity and access management, `http://docs.amazonwebservices.com/IAM/latest/UserGuide/Welcome.html`.

[3] 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.

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

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

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

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

[8] 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.

[9] D. Brewer and M. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214. IEEE, 1989.

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

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

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

[13] L. Cirio, I. F. Cruz, and R. Tamassia. A role and attribute based access control system using semantic web technologies. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems - Volume Part II*, OTM'07, pages 1256–1266, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] M. J. Covington and M. R. Sastry. A contextual attribute-based access control model. In *Proceedings of the 2006 international conference on On the Move to Meaningful Internet Systems: AWeSOMe, CAMS, COMINF, IS, KSinBIT, MIOS-CIAO, MONET - Volume Part II*, OTM'06, pages 1996–2006, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[16] 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.

[17] 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.

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

[19] 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.

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

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

[22] X. Jin, R. Krishnan, and R. Sandhu. A unified attribute-based access control model covering dac, mac and rbac. In *Proceedings of the 26th Annual IFIP WG 11.3 conference on Data and Applications Security and Privacy*, DBSec'12, pages 41–55, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] M. J. Kamfonas. Recursive hierarchies. *The Relational Journal*, 1992.

[24] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proceedings of the 16th international conference on World Wide Web*, page 686. ACM, 2007.

[25] D. R. Kuhn, E. J. Coyne, and T. R. Weil. Adding attributes to role-based access control. *Computer*, 43(6):79–81, June 2010.

[26] B. Lang, I. Foster, F. Siebenlist, R. Ananthakrishnan, and T. Freeman. A flexible attribute based access control method for grid computing. *Journal of Grid Computing*, 7(2):169–180, 2009.

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

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

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

[31] S. Marouf, M. Shehab, A. Squicciarini, and S. Sundareswaran. Statistics & Clustering Based Framework for Efficient XACML Policy Evaluation. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 118–125. IEEE, 2009.

[32] P. Mazzoleni, B. Crispo, S. Sivasubramanian, and E. Bertino. XACML Policy Integration Algorithms. *ACM Transactions on Information and System Security*, 11(1), 2008.
[33] T. Moses et al. Extensible access control markup language (XACML) version 2.0. *Oasis Standard*, 200502, 2005.
[34] T. Priebe, W. Dobmeier, and N. Kamprath. Supporting attribute-based access control with ontologies. In *Proceedings of the First International Conference on Availability, Reliability and Security*, ARES '06, pages 465–472, Washington, DC, USA, 2006. IEEE Computer Society.
[35] P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo. An algebra for fine-grained integration of xacml policies. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 63–72. ACM, 2009.
[36] R. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.
[37] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
[38] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb. 1996.
[39] H. Shen. A semantic-aware attribute-based access control model for web services. In *Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '09, pages 693–703, Berlin, Heidelberg, 2009. Springer-Verlag.
[40] W. W. Smari, J. Zhu, and P. Clemente. Trust and privacy in attribute based access control for collaboration environments. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '09, pages 49–55, New York, NY, USA, 2009. ACM.
[41] J. Staten and L. E. Nelson. Market Overview: Private Cloud Solutions, Q2 2011. Technical report, Forrester Research, Inc, 2011.
[42] G. Stoneburner, A. Y. Goguen, and A. Feringa. Sp 800-30. risk management guide for information technology systems. Technical report, Gaithersburg, MD, United States, 2002.
[43] SunXACML. Sun XACML Implementation, 2006. `http://sunxacml.sourceforge.net`.
[44] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, FMSE '04, pages 45–55, New York, NY, USA, 2004. ACM.
[45] Y. Wei, C. Shi, and W. Shao. An attribute and role based access control model for service-oriented environment. In *Control and Decision Conference (CCDC), 2010 Chinese*, pages 4451–4455, 2010.
[46] E. Yuan and J. Tong. Attributed based access control (abac) for web services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 561–569, Washington, DC, USA, 2005. IEEE Computer Society.
[47] L. Yuan, H. Chen, J. Mai, C. Chuah, Z. Su, P. Mohapatra, and C. Davis. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy*, pages 199–213, 2006.
[48] X. Zhang, Y. Li, and D. Nalla. An attribute-based access matrix model. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 359–363, New York, NY, USA, 2005. ACM.
[49] J. Zhu and W. Smari. Attribute based access control and security for collaboration environments. In *Aerospace and Electronics Conference, 2008. NAECON 2008. IEEE National*, pages 31–35, 2008.