Research Article

Elimination of the Redundancy Related to Combining Algorithms to Improve the PDP Evaluation Performance

Fan Deng,¹ Li-Yong Zhang,² Bo-Yu Zhou,³ Jia-Wei Zhang,² and Hong-Yang Cao²

¹School of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an 710054, China

²School of Software, Xidian University, Xi'an 710071, China

³School of Computer Science and Technology, Xidian University, Xi'an 710071, China

Correspondence should be addressed to Fan Deng; deng_enya@126.com

Received 17 December 2015; Revised 6 April 2016; Accepted 20 April 2016

Academic Editor: Veljko Milutinovic

Copyright © 2016 Fan Deng et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

If there are lots of redundancies in the policies loaded on the policy decision point (PDP) in the authorization access control model, the system will occupy more resources in operation and consumes plenty of evaluation time and storage space. In order to detect and eliminate policy redundancies and then improve evaluation performance of the PDP, a *redundancy related to combining algorithms* detecting and eliminating engine is proposed in this paper. This engine cannot only detect and eliminate the *redundancy related to combining algorithms*, but also evaluate access requests. A *Resource Brick Wall* is constructed by the engine according to the resource attribute of a policy's target attributes. By the *Resource Brick Wall* and the policy/rule combining algorithms, three theorems for detecting *redundancies related to combining algorithms* are proposed. A comparison of the evaluation performance of the *redundancy related to combining algorithms* detecting and eliminating engine with that of Sun PDP is made. Experimental results show that the evaluation performance of the PDP can be prominently improved by eliminating the *redundancy related to combining algorithms*.

1. Introduction

Currently, access control is an important protection mechanism in the network and information security [1]. And using the security policy to describe security requirements in information systems has been the main method for authorization access control [2]. A policy can be defined as a group of descriptive principles implemented on the basis of practical requirements. These principles are the constraints for authorizing the system to make a decision. The XACML (eXtensible Access Control Markup Language) [3] is widely used in a distributed application system [4] in the SOA (Service-Oriented Architecture) [5] environment so that the security policy can be better expressed. The XACML is an open-standard language based on the XML and is used to standardize access control implemented in the XML (eXtensible Markup Language). It provides a uniform policy description language and improves the efficiency of collaboration

among different organizations in the SOA environment. The XACML's characteristics are as follows:

- (i) *Great adaptability*: the XACML can be normally applied in multiple application environment.
- (ii) *Preferable compatibility*: the XACML supports multiple data types and policy/rule combining algorithms.
- (iii) *Outstanding expression ability*: the XACML can accurately express access control requirements with different complexities and fine grains.

The XACML is a policy describing language with strong adaptability, universality, and extensibility and it supports distributed applications [6]. However, it does not have a standard agreement on factors such as granularity size, resource level, and condition interval [7]. Some rules in a policy may lose their effect because of policy/rule combining algorithms

[8]. And a policy set may be formulated and edited by multiple administrators, each of whom has a different understanding of the policy and a different opinion on policy setting, which could lead to redundancies in a policy set's rules [9]. Thus, detecting and eliminating the *redundancy related to combining algorithms* is the core and key point of our research.

On the other hand, the XACML assumes that all the rules can be trusted, so a request might be responded to repeatedly [10]. In the authorization access control model, if the policy loaded on the PDP contains lots of redundancies related to policy/rule combining algorithms, it not only consumes and wastes lots of system resources, but also greatly increases the time the PDP spends in evaluating access requests. And then the evaluation performance of the PDP is affected. In policy management [11] based on the XACML, as many redundancies related to policy/rule combining algorithms must be eliminated as possible to accelerate the speed of policy matching so that the PDP in the authorization center can effectively make authorization decisions and feed authorization results back [12].

To sum up, the important and critical problem to be solved is how to find an effective way to detect and eliminate the *redundancy related to combining algorithms* accurately and then improve the evaluation performance of the PDP.

This paper makes the following contributions.

- (1) A *redundancy related to combining algorithms* detecting and eliminating engine is presented, which not only can detect and eliminate the *redundancy related to combining algorithms* in a policy and between policies, but also has the same ability as the PDP. This engine can evaluate the access control by loading policies whose *redundancies related to combining algorithms* have been eliminated.
- (2) A *Resource Brick Wall* is constructed by the *redundancy related to combining algorithms* detecting and eliminating engine according to the *resource* attribute in a policy's target attributes. The *Resource Brick Wall* can effectively express the dependent relationship between resource attributes and detecting and eliminating the *redundancy related to combining algorithms* by the additional subject attribute, condition attribute, and effect information.
- (3) By the *Resource Brick Wall* and policy/rule combining algorithms, the corresponding detecting and eliminating algorithm is proposed aiming at different types of *redundancies related to combining algorithms*. This algorithm only compares a rule with those which are likely to generate redundancies and this could avoid a large number of unnecessary comparisons with irrelevant rules. And it optimizes the traditional traversal redundancy detection and greatly improves the efficiency of detecting and eliminating redundancies.

The remainder of this paper is organized as follows. Section 2 reviews the related work on policy analysis, management, and high performance evaluation. Section 3 introduces definitions used in the research on the *redundancy related to combining algorithms* detection and elimination. In Section 4,

a *redundancy related to combining algorithms* detecting and eliminating engine is introduced which can detect and eliminate the *redundancy related to combining algorithms* in a policy and between policies and evaluate access requests. In Section 5, a *Resource Brick Wall* is proposed when we discuss the *redundancy related to combining algorithms* detecting and eliminating algorithm. Section 6 focuses on the theorems for detecting *redundancies related to combining algorithms*. A relevant detection and elimination algorithm for the *redundancy related to combining algorithms* is given in Section 7. Section 8 makes a comparison in evaluation performance between the *redundancy related to combining algorithms* detecting and eliminating engine and Sun PDP. Finally, Section 9 presents some conclusions and directions for our future work.

2. Related Work

There are numerous prior works on access control policies that mainly focus on policy evaluation, analysis, and testing to detect and eliminate redundancy. Masi et al. [13] formalized the XACML 2.0 semantics and proposed an alternative syntax supporting policy composition. They implemented a tool to compile policies into Java classes following the proposed semantic rules, where these classes are executed to compute policy decisions. Liu et al. [14] proposed XEngine, which can convert a textual XACML policy to a numerical policy and can convert a numerical policy with complex structures to a numerical policy with a normalized structure. Also, XEngine can convert the normalized numerical policy to tree data structures for efficient processing of requests. Marouf et al. [15] applied a clustering technique to policy sets based on the *K*-means algorithm. The proposed clustering technique categorizes policies and rules within a policy set and policy, respectively, in respect to target subjects. When a request is received, it is redirected to applicable policies and rules that correspond to its subjects, hence, avoiding unnecessary evaluations from occurring. Mourad and Jebbaoui [16] proposed (1) a mathematical intermediate representation of policies based on set theory that maintains the same XACML structure and accounts for all its elements and their subelements including rule conditions, obligations, policy request, and policy response and (2) a formal semantics that takes advantage of the mathematical operations to provide efficient evaluation of policy elements and constructs through deductive logic and inference rules. Jebbaoui et al. [17] resolve the complexity of policies by elaborating an intermediate set-based representation to which the elements of XACML are automatically converted. Moreover, it allows detecting flaws, conflicts, and redundancies between rules by offering new mechanisms to analyze the meaning of policy rules through semantics verification by inference rule structure and deductive logic.

Recently, people focus more and more on studying the methods for detecting and eliminating policy redundancies. And researches on the policy redundancies can be categorized into two groups.

- (1) The first group detect and eliminate redundancies by optimizing the XACML or improving the policy itself.

Chen and Xu [18] analyzed the reason why policy redundancies exist first. Then they proposed formal methods based on the descriptive logic so that the machine can understand a policy's semantics and context. On the basis of this, policy expansion and reasoning were carried on, which made the XACML gain better reasoning ability and semantics expressing ability. The method based on the descriptive logic can detect policy redundancies and eliminate these redundancies according to policy/rule combining algorithms. But every attribute in a policy's target elements may cause redundancies and this problem was not fully considered in this paper. Lei et al. [19] indicated that although the XACML provides a convenient way to describe the security policy, this method still has its weakness. This paper analyzed the fact that using the XACML to express the inheritance relationship of rules will lead to policy redundancies and introduced atomic privilege to eliminate policy redundancies. Martin [20] indicated that if a rule is unreachable because of a group of constraints that this rule cannot meet, then it becomes a redundancy rule. A policy redundancy detection method based on *Chang-Impact* analysis was presented in this paper; however, no corresponding detection model was proposed and the support and application of this detection technique were absent. Besides the above researches, policy redundancies can be detected by classifying the subject based on the security level and role [21, 22], and then all policy redundancies can be eliminated by revising policies.

- (2) The second group put forward policy redundancy detecting methods by establishing research models and relevant mechanisms. Aiming at this problem, Hu et al. [23] indicated that the Internet provides users with more convenient services, and at the same time the information system suffers from security attacks caused by unauthorized operations in business service. Besides, the design and management of the Web access control policy are always prone to errors for lack of efficient analysis mechanisms and tools. A policy analyzing method based on the Web access control was proposed in this paper. Also, a splitting technique based on the XACML policy was introduced to identify policy redundancies accurately and then policy redundancies could be eliminated. Dinu et al. [24] indicated that the XACML itself does not support policy redundancy analysis, though it has the great expression ability. The thought of state coverage was utilized to analyze the cause of policy redundancy, and a judging theorem that can be applied for redundancies within a policy or between policies under multiple policy/rule combining algorithms was put forward. The target element of a policy was considered when detecting redundancies in this paper while the condition factor's effect on policy redundancies was ignored, which lead to the imperfection of redundancy detection. Zhang et al. [25] focused on

security policy integration and redundancy elimination among different medical institutions. They emphasized that eliminating policy redundancies before healthcare institutions collaborate with each other is vitally important and pointed out that the present research does not consider how to eliminate redundancies in the dynamically loaded policy set according to the type of redundancy in the healthcare collaborative procedure. On the premise of taking both the constraint condition and metadata information, they also proposed a method based on the role-based access control policy model to eliminate redundancies in the XACML policy. Finally, they proposed a method of filtering and collecting the policies the user needs among different organizations and institutions. Guarnieri et al. [26] proposed three methods of eliminating redundancies in the access control policy and these methods reduced the authorization operation frequency in a policy step by step. They also indicated that problems relevant to redundancy belong to an NP-hard problem. Aiming at the Minimum Policy Problem and Minimum Irreducible Policy Problem in these problems, they proposed solutions for eliminating redundancies in the security policy.

Mainly on the basis of optimizing the XACML or improving the policy itself, the above researches pointed out the great flexibility of the XACML policy structure and the difficulty in detecting policy redundancies. Detection was only applied to part of a policy while how to detect and eliminate the redundancy related to policy/rule combining algorithms was not considered. And their research did not go in depth and rarely considered the complex situation. Then, although the above researches can detect redundancies in a policy and between policies, they traverse rules in a policy many times in the process of detecting policy redundancies, which unavoidably increases the detection time. The *Resource Brick Wall* proposed in our paper carries out the reasonable partition and hierarchical process on resources. These processes can effectively avoid unnecessary comparisons among resources in different levels. Thus, they can optimize comparisons among rules and reduce the frequency of traversing each rule in a policy set when detecting redundancies and then improve the efficiency of detecting policy redundancies and lay a foundation for an efficient evaluation of the PDP.

3. Related Definitions

Since redundancy exists in rules within a policy or between different policies, the formal description of the relative definition of rules will be given first and the rules needed in detecting redundancies are also presented.

Definition 1 (rule). sub_{i1} , res_{i2} , act_{i3} , env_{i4} , con_{i5} , and $Effect_i$ represent the elements of attributes like subject, resource, action, environment, condition, and effect, respectively, and these six attributes belong to the rule $Rule_i$. $\bigcup_{i1=1}^m sub_{i1}$, $\bigcup_{i2=1}^n res_{i2}$, $\bigcup_{i3=1}^o act_{i3}$, $\bigcup_{i4=1}^p env_{i4}$, and $\bigcup_{i5=1}^q con_{i5}$ represent the union of elements of each attribute, respectively. And

m, n, o, p and q are the number of each attribute's elements. The definition of $Rule_i$ can be represented by Formula (1) as follows:

$$Rule_i = \left(\bigcup_{i1=1}^m sub_{i1}, \bigcup_{i2=1}^n res_{i2}, \bigcup_{i3=1}^o act_{i3}, \bigcup_{i4=1}^p env_{i4}, \bigcup_{i5=1}^q con_{i5}, Effect_i \right). \quad (1)$$

There are three aspects needed to be made clear.

- (1) Each resource discussed in this paper has a unique resource path. Resources on different resource paths are identified as different resources. In other words, we think that two identical resources can only appear on an identical resource path and cannot appear on different resource paths.
- (2) We just focus on the clock time when studying the condition attribute.
- (3) The environment attribute will be ignored in the following analysis and discussion, because it does not cause the *redundancy related to combining algorithms*, which is studied in this paper.

Definition 2 (state satisfaction). If a subject attribute sub_{i1} in $Rule_i$ can access the resource attribute res_{i2} with action act_{i3} in the condition of con_{i4} and its effect is $Effect_i$ (*Deny* or *Permit*), then one has the definition that $State_i$ satisfies $(sub_{i1}, res_{i2}, act_{i3}, con_{i4}, Effect_i)$, as shown in Formula (2) as follows:

$$State_i \triangleright (sub_{i1}, res_{i2}, act_{i3}, con_{i4}, Effect_i). \quad (2)$$

Definition 3 (atomic attribute). If an attribute in a rule has no more than one element which may be absent, then one calls this attribute the *atomic attribute*.

Definition 4 (composite attribute). If an attribute in a rule has more than one element, then one calls it the *composite attribute*.

Since rules containing the atomic attribute are easier to store and process than those containing the composite attribute, so we need to apply attribute atomization to rules containing composite rules.

Rule 1 (rule attribute atomization). If there are *composite attributes* in a rule, one can divide a *composite attribute* into several *atomic attributes* and then composite it with other attributes' elements. Finally, one can get several rules which only contain *atomic attribute*.

By Rule 1, we can carry out attribute atomization on all rules in a policy and transfer them into rules containing only the atomic attribute. Thus, we lay the foundation for building and theoretical analysis of the *redundancy related to combining algorithms* detecting and eliminating engine.

Definition 5 (dependent relationship of resources). For resource attributes res_1 and res_2 , if res_1 is in the upper level

compared to res_2 , then one says that resource attributes res_1 and res_2 satisfy the *dependent relationship of resources*. res_2 's dependency on res_1 is denoted as $res_1 \mapsto res_2$.

Definition 6 (overlapping relationship of conditions). For condition attributes con_1 and con_2 , if $con_1 \cap con_2 \neq \emptyset$, one says that condition attributes con_1 and con_2 satisfy the *overlapping relationship of conditions*. The intersection between con_1 and con_2 is denoted as $con_1 \odot con_2$.

Definition 7 (redundancy). For a rule in a policy, if its function cannot work, namely, this rule will not affect the result of evaluating access requests, then one calls this rule a *redundancy rule*:

$$\begin{aligned} R_1 &= (sub_1, res_1, act_1, con_1, Effect_1), \\ R_2 &= (sub_2, res_2, act_2, con_2, Effect_2). \end{aligned} \quad (3)$$

Definition 8 (common resource redundancy). For two rules R_1 and R_2 as shown in Formula (3), they belong to the same policy or two different policies. If R_1 and R_2 have the same subject attribute, resource attribute, action attribute, and environment attribute (the environment attribute will be ignored), and without loss of generality, their condition attributes have an overlapping relationship ($con_1 \cap con_2 \neq \emptyset$) and their effects are the same (i.e., to say these two rules' $Effect_i$ should be *Deny* or *Permit* at the same time); then the *common resource redundancy* happens between R_1 and R_2 . These conditions could be expressed by Formula (4) as follows:

$$\begin{aligned} &(sub_1 = sub_2) \wedge (res_1 = res_2) \wedge (act_1 = act_2) \\ &\wedge (con_1 \cap con_2 \neq \emptyset) \wedge (Effect_1 = Effect_2). \end{aligned} \quad (4)$$

Definition 9 (dependent resource redundancy). For two rules R_1 and R_2 as shown in Formula (3), they belong to the same policy or two different policies. If R_1 and R_2 have the same subject attribute, action attribute, and environment attribute (the environment attribute will be ignored), and without loss of generality, there exists the *dependent relationship of resources* between R_1 and R_2 ($res_1 \mapsto res_2$), their condition attributes have an overlapping relationship ($con_1 \cap con_2 \neq \emptyset$), and their effects are both *Deny*; then the *dependent resource redundancy* happens between R_1 and R_2 . These conditions could be expressed by Formula (5) as follows:

$$\begin{aligned} &(sub_1 = sub_2) \wedge (res_1 \mapsto res_2) \wedge (act_1 = act_2) \\ &\wedge (con_1 \cap con_2 \neq \emptyset) \\ &\wedge (Effect_1 = Effect_2 = Deny). \end{aligned} \quad (5)$$

Definition 10 (redundancy related to combining algorithms). For a redundancy identified as the *redundancy related to combining algorithms*, the prerequisite is that the detection is on the basis of the policy/rule combining algorithms.

This paper aims at statically detecting and eliminating the *redundancy related to combining algorithms* in the policies loaded on the PDP, so that the evaluation performance of the

FIGURE 1: Redundancy related to combining algorithms detecting and eliminating engine XDPRE.

FIGURE 2: Resource Brick Wall.

PDP can be improved. The research in this paper focuses on the *redundancy related to combining algorithms* and its detecting and eliminating algorithm is proposed to solve this kind of problem.

4. Redundancy Related to Combining Algorithms Detecting and Eliminating Engine XDPRE

Our proposed *redundancy related to combining algorithms* detecting and eliminating engine termed the XDPRE (XiDian Policy Redundancy Engine) is shown in Figure 1. The XDPRE is able to detect and eliminate the *redundancy related to combining algorithms* within a policy or between policies. The XDPRE receives all the policies $Policy_1, Policy_2, \dots, Policy_n$ in the policy set $PolicySet$ as the input and its responsibilities are shown in the following three aspects.

- (1) Detect and eliminate the *redundancy related to combining algorithms* within a policy.
- (2) Detect and eliminate the *redundancy related to combining algorithms* between policies.
- (3) The XDPRE itself has the function of the PDP. By loading policies whose *redundancies related to combining algorithms* have been eliminated, the XDPRE

can evaluate access requests and return the authorization result to context processors and the PDP.

The output of the XDPRE is a new policy set termed $PolicySet'$. And there is no *redundancy related to combining algorithms* in the policy $Policy'_1, Policy'_2, \dots, Policy'_n$ of the policy set $PolicySet'$.

We adopt the XDPRE to detect and eliminate the *redundancy related to combining algorithms* within a policy and between policies successively.

5. Resource Brick Wall

In order to detect and eliminate the *redundancy related to combining algorithms* contained in the policies loaded on the PDP, so as to achieve the purpose of efficient evaluation of the PDP, in this section, we construct the *Resource Brick Wall* (RBW) in the XDPRE, an engine for detecting and eliminating the *redundancy related to combining algorithms*, based on the resources existing in the target attribute of policies, as shown in Figure 2.

Definition 11 (Resource Brick Wall). For each resource in the rules, one builds a *resource brick*, and then according to the dependent relationship between resources, one can build a *Resource Brick Wall* of all the bricks representing resources

from top to bottom, as shown in Figure 2(a). The axis *Con* represents the condition attribute in the rules corresponding to each resource and the height of a resource brick is equal to the size of time distinct of its condition attribute. The axis *Lev* represents the level corresponding to each resource. The axis *Res* represents resources; the length of a resource brick in the i th ($i \geq 1$) layer equals the sum of all the resources which have the dependency with it in the $(i + 1)$ th layer. Each resource brick has correlation with a *Subject-Condition-Effect List* (SCEL). The SCEL stores all the subject attributes' information corresponding to the resources the resource brick represents, as well as the condition attribute information and effect information of these subjects, as shown in Figure 2(b). As for the resource brick *res*, its SCEL is denoted as the $SCEL_{\llbracket res \rrbracket}$ in which any subject attribute *sub* is denoted as $sub_{\llbracket res \rrbracket}$. In Figure 2(a), res_0 is the first layer resource, and $res_1, res_2(res'_2)$, and $res_3(res'_3)$ are all the lower resources of res_0 , that is, the second layer resource. res_2 and res'_2 are the same resource attribute of two different rules, whose condition attributes have an overlapping relationship. res_3 and res'_3 are also the same resource attribute of two different rules whose condition attributes do not have an overlapping relationship.

$$\begin{array}{c} State_i \triangleright (sub_0, res_1, act, con_i, Deny) \\ res_1 \mapsto res_2 \end{array} \implies State_i \triangleright (sub_0, res_2, act, con_i, Deny). \quad (6)$$

② If a subject sub_0 of the subject set $SubjectSet_{\llbracket res_1 \rrbracket}^{Permit}$ of the resource attribute res_1 permits accessing resource attribute res_1 , then we cannot deduce what operation of access

$$\begin{array}{c} State_i \triangleright (sub_0, res_1, act, con_i, Permit) \\ res_1 \mapsto res_2 \end{array} \not\Rightarrow State_i \triangleright (sub_0, res_2, act, con_i, Effect_i). \quad (7)$$

The transmission rules of access authority can be summarized as the form of Figure 3. In Figure 3, between res_1 and res_2 exists the dependent relationship $res_1 \mapsto res_2$. If a subject sub_0 denies accessing the upper layer resource attribute, we can deduce that it also denies accessing the lower layer resource attribute res_2 . The solid line marked Y in the figure indicates that *Deny* access authority can be transmitted from the upper resource attribute to the lower resource attribute. If a subject sub_0 permits accessing the upper resource attribute, we cannot deduce what operation of access authority it has to the lower resource attribute res_2 , and the dotted line marked N in the figure indicates that *Permit* access authority cannot be transmitted from the upper resource attribute to the lower resource attribute. For example, if the subject *Employee* denies the writing operation to the resource $E:\backslash File$, then we can deduce that it denies the writing operation to the resource $E:\backslash File\backslash ProductFile$. However, if the subject *Employee* permits the writing operation to the $E:\backslash File$, then we cannot deduce what operation of access authority it has to the resource $E:\backslash File\backslash ProductFile$.

It is worth noting that if two resource bricks are the same, we should view them as one and merge their corresponding SCELs to ensure that each of the resource bricks in the RBW is the only one. The RBW can efficiently represent the *dependent relationship of resources*.

There is a hierarchical relationship between resource attributes. For example, the hierarchical relationship of the system resource directory is represented by the dependent relationship between resource attributes. According to the definition specification of the target attribute in the XACML (Version 3.0) OASIS [27], together with the definition of state satisfaction, we give the transmission rules of access authority, as shown in Rule 2.

Rule 2 (transmission rules of access authority). As for the two resource attributes res_1 and res_2 in the two rules R_1 and R_2 , when res_1 and res_2 satisfy $res_1 \mapsto res_2$, consider the following:

① If a subject sub_0 of the subject set $SubjectSet_{\llbracket res_1 \rrbracket}^{Deny}$ of the resource attribute res_1 denies accessing resource attribute res_1 , then it also denies accessing its dependent resource res_2 , as shown in Formula (6) as follows:

authority it has to the dependent resource attribute res_2 ($Effect_i$ is *Deny* or *Permit*), as shown in Formula (7) as follows:

6. Detection and Elimination of Redundancy Related to Combining Algorithms

The eventual evaluation result of a rule in a policy depends on the evaluation of the condition attribute. If the return value of the condition is *Indeterminate*, the rule will return *Indeterminate*. If the return value of the condition is *False*, the rule will return *NotApplicable*. If the return value of the condition is *True*, the rule will return *Effect*, which can be *Permit* or *Deny*.

For a redundancy identified as the *redundancy related to combining algorithms*, the prerequisite is that the detection is on the basis of the policy/rule combining algorithms. The definition and application of the policy/rule combining algorithms are related to the return value of evaluation result, which is *Indeterminate*, *NotApplicable*, *Permit*, or *Deny*. Because the return value *Indeterminate* occurs when faults appear in evaluation and *NotApplicable* occurs when the matched rule is not found, the two return values are meaningless for discussing and analyzing whether there exists redundancy in rules. Therefore, in this section we only discuss and

FIGURE 3: Transmission relationship of access authority.

analyze redundancy when the return value is *Permit* or *Deny* and ignore *Indeterminate* or *NotApplicable*.

This section proposes three theorems for detecting *redundancies related to combining algorithms* according to the “*Deny-Overrides*”, “*Permit-Overrides*”, and “*First-Applicable*” policy/rule combining algorithms supported by the XACML standards and gives the methods for eliminating *redundancies related to combining algorithms*.

6.1. Redundancy Related to the Deny-Overrides Combining Algorithm. The policy combining algorithms and rule combining algorithms supported by the XACML standards all contain “*Deny-Overrides*”; that is, if the effect of either of the policies (or rules) is *Deny*, the final authorized result is *Deny*. Next, we give the theorem for detecting the *redundancy related to “Deny-Overrides” combining algorithm* with the “*Deny-Overrides*” policy/rule combining algorithm:

$$\begin{aligned} R_1 &= (sub_1, res_1, act_1, con_1, Effect_1), \\ R_2 &= (sub_2, res_2, act_2, con_2, Effect_2). \end{aligned} \quad (8)$$

Theorem 12. *In the condition of the “Deny-Overrides” policy/rule combining algorithm, for any two rules R_1 and R_2 in a policy or between policies, as represented by Formula (8), if R_1 and R_2 have the same subject attribute “sub” and action attribute “act”, and there is an overlapping relationship between the condition attributes of R_1 and R_2 ($con_1 \cap con_2 \neq \emptyset$), and if, without losing the generality, between the resource attributes of R_1 and R_2 exists the equation relationship ($res_1 = res_2$) or dependent relationship ($res_1 \mapsto res_2$) and the effect of R_1 is “Deny”, that is, Formula (9) is satisfied, then R_2 is the redundancy rule:*

$$\begin{aligned} &(sub_1 = sub_2) \wedge (act_1 = act_2) \wedge (con_1 \cap con_2 \neq \emptyset) \\ &\wedge ((res_1 = res_2) \wedge (res_1 \mapsto res_2)) \\ &\wedge (Effect_1 = Deny). \end{aligned} \quad (9)$$

Proof. State satisfaction of rules R_1 and R_2 is as shown in Formula (10) as follows:

$$\begin{aligned} State_1 &\triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Deny), \\ State_2 &\triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2). \end{aligned} \quad (10)$$

According to the relationship of resource attributes of rules R_1 and R_2 and the different effect values of R_2 , we give the proof in the following four situations:

① When $res_1 = res_2$ and $Effect_2 = Deny$, then $Effect_1 = Effect_2$. We have the formal derivation process as shown in Formula (11) as follows:

$$\begin{aligned} State_1 &\triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Effect_1) \\ State_2 &\triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2) \\ sub^{\llbracket res_1 \rrbracket} &= sub^{\llbracket res_2 \rrbracket} \\ res_1 &= res_2 \\ con_1 \cap con_2 &\neq \emptyset \\ Effect_1 &= Effect_2 = Deny \\ \Downarrow \\ Redundancy_{=} &(State_1, State_2, con_1 \odot con_2) \\ &= Redundancy_{=} (R_1, R_2, con_1 \odot con_2). \end{aligned} \quad (11)$$

According to the known condition and Definition 8, we deduct that rules R_1 and R_2 have the *common resource redundancy* in the overlapping condition $con_1 \odot con_2$. What is more, since we use the “*Deny-Overrides*” policy/rule combining algorithm, R_2 is the redundancy rule.

② When $res_1 = res_2$ and $Effect_2 = Permit$, then $Effect_1 \neq Effect_2$. Because conditions con_1 and con_2 have an overlapping relationship ($con_1 \cap con_2 \neq \emptyset$), according to the definition of “*Deny-Overrides*” policy/rule combining algorithm, under the condition that there appear no faults, we finally should use the effect $Effect_1 = Deny$ of R_1 as the authorization result. However, the rule R_2 does not have any effect on the process of evaluation, so the function of the rule cannot be realized. Therefore, according to Definition 7, we conclude that R_2 is the redundancy rule.

③ When $res_1 \mapsto res_2$ and $Effect_2 = Deny$, then $Effect_1 = Effect_2$. We have the formal derivation process as shown in Formula (12) as follows:

$$\begin{aligned} State_1 &\triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Effect_1) \\ State_2 &\triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2) \\ sub^{\llbracket res_1 \rrbracket} &= sub^{\llbracket res_2 \rrbracket} \\ res_1 &\mapsto res_2 \\ con_1 \cap con_2 &\neq \emptyset \\ Effect_1 &= Effect_2 = Deny \\ \Downarrow \\ Redundancy_{\mapsto} &(State_1, State_2, con_1 \odot con_2) \\ &= Redundancy_{\mapsto} (R_1, R_2, con_1 \odot con_2). \end{aligned} \quad (12)$$

According to the assumptions and Definition 9, we deduct that rules R_1 and R_2 have the *dependent resource redundancy*

in the overlapping condition $con_1 \odot con_2$. What is more, since we use the “Deny-Overrides” policy/rule combining algorithm, R_2 is the redundancy rule.

$$\begin{aligned} State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Deny) \\ sub^{\llbracket res_1 \rrbracket} = sub^{\llbracket res_2 \rrbracket} \\ res_1 \mapsto res_2 \end{aligned} \implies State_1 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_1, Deny). \quad (13)$$

From Formula (13), we can conclude that $State_1$ satisfies the fact that the subject $sub^{\llbracket res_1 \rrbracket}$ of R_1 denies accessing to the resource res_1 in the condition con_1 , and from assumptions and Formula (10) we can conclude that $State_2$ satisfies the fact that the same subject $sub^{\llbracket res_2 \rrbracket}$ of R_2 permits accessing to res_2 which is the same as res_1 in the condition con_2 . Because conditions con_1 and con_2 have an overlapping relationship ($con_1 \cap con_2 \neq \emptyset$), according to the conclusion of proof ②, we can conclude that R_2 is the redundancy rule. $\square$

According to Theorem 12, we can obtain the form derivation of detecting the *redundancy related to the “Deny-Overrides” combining algorithm*, as shown in Formula (14), where

- (i) $Redundancy_{DO}(State_1, State_2, con_1 \odot con_2)$ indicates that $State_1$ satisfying the rule R_1 and $State_2$ satisfying the rule R_2 have the *redundancy related to the “Deny-Overrides” combining algorithm* in the overlapping condition $con_1 \odot con_2$;
- (ii) $Redundancy_{DO}(R_1, R_2, con_1 \odot con_2)$ indicates that the rule R_1 corresponding to the $State_1$ and the rule R_2 corresponding to the $State_2$ have the *redundancy related to the “Deny-Overrides” combining algorithm* in the overlapping condition $con_1 \odot con_2$:

CombiningAlg = Deny – Overrides

$$\begin{aligned} State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Deny) \\ State_2 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2) \\ sub^{\llbracket res_1 \rrbracket} = sub^{\llbracket res_2 \rrbracket} \\ res_1 = res_2 \cup res_1 \mapsto res_2 \\ con_1 \cap con_2 \neq \emptyset \\ \Downarrow \\ Redundancy_{DO}(State_1, State_2, con_1 \odot con_2) \\ = Redundancy_{DO}(R_1, R_2, con_1 \odot con_2). \end{aligned} \quad (14)$$

When detecting the *redundancy related to the “Deny-Overrides” combining algorithm* between the two rules R_1 and R_2 (as shown in Formula (8)) in a policy or between policies, the method for eliminating the redundancy is to reserve the rule R_1 firstly. If $con_1 \odot con_2 = con_2$, we will delete the rule R_2 directly. If $con_1 \odot con_2 \neq con_2$, we will update the condition

④ When $res_1 \mapsto res_2$ and $Effect_2 = Permit$, then $Effect_1 \neq Effect_2$. According to Rule 2 (the transmission rules of access authority), we can get Formula (13) as follows:

attribute of the rule R_2 to $con'_2 = con_2 - con_1 \odot con_2$, that is, update the rule R_2 to a new rule as shown in Formula (15) as follows:

$$R'_2 = (sub_2, res_2, act_2, con_2 - con_1 \odot con_2, Effect_2). \quad (15)$$

6.2. Redundancy Related to the Permit-Overrides Combining Algorithm. The policy combining algorithms and rule combining algorithms supported by the XACML standards all contain “Permit-Overrides”; that is, if the return effect of either of the policies (or rules) is *Permit*, the final authorized result is *Permit*. Next, we give the theorem for detecting the *redundancy related to “Permit-Overrides” combining algorithm* with the “Permit-Overrides” policy/rule combining algorithm:

$$\begin{aligned} R_1 &= (sub_1, res_1, act_1, con_1, Effect_1), \\ R_2 &= (sub_2, res_2, act_2, con_2, Effect_2). \end{aligned} \quad (16)$$

Theorem 13. In the condition of the “Permit-Overrides” policy/rule combining algorithm, for any two rules R_1 and R_2 in a policy or between policies, as shown in Formula (16), if R_1 and R_2 have the same subject attribute “sub”, resource attribute “res”, and action attributes “act”, and there is an overlapping relationship between the condition attributes of R_1 and R_2 ($con_1 \cap con_2 \neq \emptyset$), without losing the generality, one supposes that the effect of R_1 is *Permit*; that is, Formula (17) is satisfied, and then R_2 is the redundancy rule:

$$\begin{aligned} (sub_1 = sub_2) \wedge (res_1 = res_2) \wedge (act_1 = act_2) \\ \wedge (con_1 \cap con_2 \neq \emptyset) \wedge (Effect_1 = Permit). \end{aligned} \quad (17)$$

Proof. State satisfaction of rules R_1 and R_2 is as shown in Formula (18) as follows:

$$\begin{aligned} State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Permit), \\ State_2 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2). \end{aligned} \quad (18)$$

In Formula (18), $res_1 = res_2$. According to the different effect values of R_2 , we give the proof in the following two situations:

① When $Effect_2 = Deny$, then $Effect_1 \neq Effect_2$. Because conditions con_1 and con_2 have an overlapping relationship ($con_1 \cap con_2 \neq \emptyset$), according to the definition of “Permit-Overrides” policy/rule combining algorithm, under the condition that there appear no faults, we should finally use the effect $Effect_1 = Permit$ of R_1 as the authorization result.

However, the rule R_2 does not have any effect on the process of evaluation, so the function of the rule cannot be realized. Therefore, according to Definition 7, we conclude that R_2 is the redundancy rule in the “Permit-Overrides” policy/rule combining algorithm.

② When $Effect_2 = Permit$, then $Effect_1 = Effect_2$. We have the form deduction process as shown in Formula (19) as follows:

$$\begin{aligned}
& State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Effect_1) \\
& State_2 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2) \\
& sub^{\llbracket res_1 \rrbracket} = sub^{\llbracket res_2 \rrbracket} \\
& res_1 = res_2 \\
& con_1 \cap con_2 \neq \emptyset \\
& Effect_1 = Effect_2 = Permit \\
& \Downarrow \\
& Redundancy_{\text{PO}}(State_1, State_2, con_1 \odot con_2) \\
& = Redundancy_{\text{PO}}(R_1, R_2, con_1 \odot con_2).
\end{aligned} \tag{19}$$

According to the known condition and Definition 8, we deduct that rules R_1 and R_2 have the *common resource redundancy* in the overlapping condition $con_1 \odot con_2$. What is more, since we use the “Permit-Overrides” policy/rule combining algorithm, R_2 is the redundancy rule. $\square$

According to Theorem 13, we can conclude that the form derivation of detecting the *redundancy related to the “Permit-Overrides” combining algorithm* is as shown in Formula (20), where

- (i) $Redundancy_{\text{PO}}(State_1, State_2, con_1 \odot con_2)$ indicates that $State_1$ satisfying the rule R_1 and $State_2$ satisfying the rule R_2 have the *redundancy related to the “Permit-Overrides” combining algorithm* in the overlapping condition $con_1 \odot con_2$;
- (ii) $Redundancy_{\text{PO}}(R_1, R_2, con_1 \odot con_2)$ indicates that the rule R_1 corresponding to the $State_1$ and the rule R_2 corresponding to the $State_2$ have the *redundancy related to the “Permit-Overrides” combining algorithm* in the overlapping condition $con_1 \odot con_2$:

$$CombiningAlg = Permit - Overrides$$

$$\begin{aligned}
& State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Permit) \\
& State_2 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2) \\
& sub^{\llbracket res_1 \rrbracket} = sub^{\llbracket res_2 \rrbracket} \\
& res_1 = res_2
\end{aligned}$$

$$con_1 \cap con_2 \neq \emptyset$$

$\Downarrow$

$$\begin{aligned}
& Redundancy_{\text{PO}}(State_1, State_2, con_1 \odot con_2) \\
& = Redundancy_{\text{PO}}(R_1, R_2, con_1 \odot con_2).
\end{aligned} \tag{20}$$

When detecting the *redundancy related to the “Permit-Overrides” combining algorithm* between the two rules R_1 and R_2 (as shown in Formula (16)) in a policy or between policies, the method for eliminating the redundancy is to reserve the rule R_1 firstly. If $con_1 \odot con_2 = con_2$, we will delete the rule R_2 directly. If $con_1 \odot con_2 \neq con_2$, we will update the condition attribute of the rule R_2 to $con_2 = con_2 - con_1 \odot con_2$, that is, update the rule R_2 to a new rule as shown in Formula (21) as follows:

$$R'_2 = (sub_2, res_2, act_2, con_2 - con_1 \odot con_2, Effect_2). \tag{21}$$

6.3. Redundancy Related to the First-Applicable Combining Algorithm. The policy combining algorithms and rule combining algorithms supported by the XACML standards all contain “First-Applicable”; that is, the final authorized result of policy (or rule) is the judgment result of the first matched policy (or rule). Next, we give the theorem for detecting the *redundancy related to “First-Applicable” combining algorithm* with the “First-Applicable” policy/rule combining algorithm:

$$\begin{aligned}
R_1 &= (sub_1, res_1, act_1, con_1, Effect_1), \\
R_2 &= (sub_2, res_2, act_2, con_2, Effect_2).
\end{aligned} \tag{22}$$

Theorem 14. In the condition of the “First-Applicable” policy/rule combining algorithm, for any two rules R_1 and R_2 in a policy or between policies, as shown in Formula (22), if R_1 and R_2 have the same subject attribute “sub”, resource attribute “res”, and action attribute “act”, and there is an overlapping relationship between the condition attributes of R_1 and R_2 ($con_1 \odot con_2 \neq \emptyset$), that is, Formula (23) is satisfied, without losing the generality, one supposes that R_1 is the first applicable rule, and then R_2 is the redundancy rule:

$$\begin{aligned}
& (sub_1 = sub_2) \wedge (res_1 = res_2) \wedge (act_1 = act_2) \\
& \wedge (con_1 \cap con_2 \neq \emptyset).
\end{aligned} \tag{23}$$

Proof. State satisfaction of rules R_1 and R_2 is as shown in Formula (24) as follows:

$$\begin{aligned}
& State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Effect_1), \\
& State_2 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2).
\end{aligned} \tag{24}$$

In Formula (24), $res_1 = res_2$. According to the different effect values of R_2 , we give the proof in the following three situations:

① When $Effect_1 = Deny$ and $Effect_2 = Deny$, then $Effect_1 = Effect_2$. We have the formal deduction process as shown in Formula (25) as follows:

$$\begin{aligned}
& State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Effect_1) \\
& State_2 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2) \\
& sub^{\llbracket res_1 \rrbracket} = sub^{\llbracket res_2 \rrbracket} \\
& res_1 = res_2 \\
& con_1 \cap con_2 \neq \emptyset \\
& Effect_1 = Effect_2 = Deny \\
& \Downarrow \\
& Redundancy = (State_1, State_2, con_1 \odot con_2) \\
& = Redundancy = (R_1, R_2, con_1 \odot con_2).
\end{aligned} \tag{25}$$

According to the known condition and Definition 8, we deduct that rules R_1 and R_2 have the *common resource redundancy* in the overlapping condition $con_1 \odot con_2$. What is more, since we use the “First-Applicable” policy/rule combining algorithm and R_1 is the first applicable rule, R_2 is the redundancy rule.

② When $Effect_1 = Permit$ and $Effect_2 = Permit$, then $Effect_1 = Effect_2$. We have the formal derivation process as shown in Formula (26) as follows:

$$\begin{aligned}
& State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Effect_1) \\
& State_2 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2) \\
& sub^{\llbracket res_1 \rrbracket} = sub^{\llbracket res_2 \rrbracket} \\
& res_1 = res_2 \\
& con_1 \cap con_2 \neq \emptyset \\
& Effect_1 = Effect_2 = Permit \\
& \Downarrow \\
& Redundancy = (State_1, State_2, con_1 \odot con_2) \\
& = Redundancy = (R_1, R_2, con_1 \odot con_2).
\end{aligned} \tag{26}$$

According to the known condition and Definition 8, we deduct that rules R_1 and R_2 have the *common resource redundancy* in the overlapping condition $con_1 \odot con_2$. What is more, since we use the “First-Applicable” policy/rule combining algorithm and R_1 is the first applicable rule, R_2 is the redundancy rule.

③ $Effect_1 \neq Effect_2$ (i.e., $Effect_1 = Deny$ and $Effect_2 = Permit$, or $Effect_1 = Permit$ and $Effect_2 = Deny$) and R_1 is the first applicable rule. Because conditions con_1 and con_2 have an overlapping relationship ($con_1 \odot con_2 \neq \emptyset$), according to the definition of “First-Applicable” policy/rule combining algorithm, under the condition that there appear no faults, we finally should use the effect $Effect_1$ of R_1 as the

authorization result. However, the rule R_2 does not have any effect on the process of evaluation; the function of the rule cannot be realized. Therefore, according to Definition 7, we conclude that R_2 is the redundancy rule in the “First-Applicable” policy/rule combining algorithm. $\square$

According to Theorem 14, we can conclude the form derivation of detecting the *redundancy related to the “First-Applicable” combining algorithm*, as shown in Formula (27), where

- (i) R_1 is the first applicable rule;
- (ii) $Redundancy_{FA}(State_1, State_2, con_1 \odot con_2)$ indicates that $State_1$ satisfying the rule R_1 and $State_2$ satisfying the rule R_2 have the *redundancy related to the “First-Applicable” combining algorithm* in the overlapping condition $con_1 \odot con_2$;
- (iii) $Redundancy_{FA}(R_1, R_2, con_1 \odot con_2)$ indicates that the rule R_1 corresponding to the $State_1$ and the rule R_2 corresponding to the $State_2$ have the *redundancy related to the “First-Applicable” combining algorithm* in the overlapping condition $con_1 \odot con_2$;

CombiningAlg = First – Overrides

$$\begin{aligned}
& State_1 \triangleright (sub^{\llbracket res_1 \rrbracket}, res_1, act, con_1, Effect_1) \\
& State_2 \triangleright (sub^{\llbracket res_2 \rrbracket}, res_2, act, con_2, Effect_2) \\
& sub^{\llbracket res_1 \rrbracket} = sub^{\llbracket res_2 \rrbracket} \\
& res_1 = res_2 \\
& con_1 \cap con_2 \neq \emptyset \\
& \Downarrow \\
& Redundancy_{FA}(State_1, State_2, con_1 \odot con_2) \\
& = Redundancy_{FA}(R_1, R_2, con_1 \odot con_2).
\end{aligned} \tag{27}$$

When detecting the *redundancy related to the “First-Applicable” combining algorithm* by Theorem 14 between the two rules R_1 and R_2 (as Formula (22) shows) in a policy or between policies, without losing the generality, we suppose that R_1 is the first applicable rule; the method for eliminating the redundancy is to reserve the rule R_1 firstly. If $con_1 \odot con_2 = con_2$, we will delete the rule R_2 directly. If $con_1 \odot con_2 \neq con_2$, we will update the condition attribute of the rule R_2 to $con'_2 = con_2 - con_1 \odot con_2$, that is, update the rule R_2 to a new rule as shown in Formula (28) as follows:

$$R'_2 = (sub_2, res_2, act_2, con_2 - con_1 \odot con_2, Effect_2). \tag{28}$$

7. Detection and Elimination Algorithms for Redundancy Related to Combining Algorithms

By the relevant definitions introduced in the paper and RBW, as well as detection and elimination methods for the

Input: (1) *PolicySet*: {*Policy*₁, *Policy*₂, ..., *Policy*_{*n*}}, where *Policy*_{*i*} is a policy
 (2) policy/rule combination algorithm *CA*
Output: a policy set where the *redundancy related to combining algorithms* have been eliminated
 (1) *ResBrickWallInitialization*(*PolicySet*)
 (2) *AddRuleToResBrickWall*(*P*, *ResBrickWall*, *CA*)
 (3) *BackTrackingDetectRedundancy*(*ResBrickWall*, *PresentResBrickId*, *CA*)
 (4) *DetectRedundancyBetweenPolicies*(*ResBrickWall*₁, *ResBrickWall*₂, *CA*)

ALGORITHM 1: Detection and elimination algorithm for redundancy related to combining algorithms. *Redundancy_DetectionAndElimination*(*PolicySet*, *CA*).

Input: *PolicySet*: {*Policy*₁, *Policy*₂, ..., *Policy*_{*n*}}, where *Policy*_{*i*} is a policy
Output: *RBW ResBrickWall*
 (1) foreach *Policy*_{*i*} ∈ *PolicySet*
 (2) foreach *r* ∈ *Policy*_{*i*}
 (3) *ResPath* = *GetResPath*(*r*)
 (4) *AddResPathToDirectory*(*ResPath*, *ResDirectory*)
 (5) *ResBrickWall* = *ConvertResDirectoryToResBrickWall*(*ResDirectory*)

ALGORITHM 2: Initialization algorithm for resource brick wall. *ResBrickWallInitialization*(*PolicySet*).

redundancy related to combining algorithms, in this section we present the detection and elimination algorithms for the *redundancy related to combining algorithms*, as shown in Algorithm 1.

In Algorithm 1, rows (1)~(3) complete the detection and elimination of the *redundancy related to combining algorithms* in a policy, and row (4) completes the detection and elimination of the *redundancy related to combining algorithms* between policies. Algorithm 1 calls four subalgorithms in turn to finish the work. Here, row (1) is the initialization algorithm for the *RBW*, responsible for initializing the *RBW*. Row (2) is the algorithm for adding rules to the *RBW*, responsible for adding every rule to the *RBW*. Row (3) is the detection and elimination backtracking algorithm, used for detecting and eliminating *redundancy related to combining algorithms* by backtracking methods in the constructed *RBW*.

Next, we will introduce algorithms used in each step of Algorithm 1. We will give the time complexity of Algorithm 1 after discussing the four subalgorithms.

7.1. Initialization Algorithm for the Resource Brick Wall. Before detecting and eliminating *redundancy related to combining algorithms*, we first need to iterate through each rule of each policy of all the policy sets and then get the resource directory structure of each rule. Finally, we initialize the *RBW* by the resource directory structure. The initialization algorithm for the *Resource Brick Wall* is as shown in Algorithm 2.

The functions contained in Algorithm 2 are as follows:

① *GetResPath*(*r*): the parameter of the function is the rule *r*, whose function is to parse the resource attribute information in the rule *r*, eventually return the resource path related to the rule *r*, and give the return value to *ResPath* variables.

② *AddResPathToResDirectory*(*r*): the parameter of the function is the resource path *ResPath* and resource directory

ResDirectory, whose function is to store the resource path *ResPath* in the proper place of the resource directory *ResDirectory*.

③ *CovertResDirectoryToResBrickWall*(*r*): the parameter of the function is the resource directory *ResDirectory*, whose function is to construct the corresponding *RBW* through the analysis of the resource directory *ResDirectory* and return the constructed *RBW* to the *ResBrickWall* variable.

Supposing that the policy set *PolicySet* has *m* policies with each policy *Policy*_{*i*} having *n* rules, then the time complexity of Algorithm 2 is $O(m * n)$.

7.2. Detection Algorithm for Redundancy Related to Combining Algorithms. The detection algorithm for the *redundancy related to combining algorithms* is as shown in Algorithm 3. Algorithm 3 contains three situations as follows:

- (1) In the condition of the “*Deny-Overrides*” policy/rule combining algorithm, for any two rules *r*₁ and *r*₂ in a policy or between policies, if *r*₁ and *r*₂ have the same subject attribute and action attribute, there is an overlapping relationship between the condition attributes of *r*₁ and *r*₂, *r*₁ and *r*₂ have the same resource attribute, the effect of either of the two rules is *Deny*, or the resource attributes of *r*₁ and *r*₂ have a dependent relationship and the effect of *r*₁ is *Deny*, then *r*₁ and *r*₂ have the *redundancy related to the “Deny-Overrides” combining algorithm*.
- (2) In the condition of the “*Permit-Overrides*” policy/rule combining algorithm, for any two rules *r*₁ and *r*₂ in a policy or between policies, if *r*₁ and *r*₂ have the same subject attribute and action attribute, there is an overlapping relationship between the condition attributes of *r*₁ and *r*₂, *r*₁ and *r*₂ have the same resource attribute, and the effect of either of the two rules is *Permit*, then

Input: (1) two rules r_1 and r_2
 (2) policy/rule combination algorithm CA

Output: (1) if between rules r_1 and r_2 exists the *redundancy related to combining algorithms*, output rules r_1 and r_2 where the *redundancy related to combining algorithms* has been eliminated
 (2) if between rules r_1 and r_2 exists the *redundancy related to combining algorithms*, output SCEL

```

(1)  if  $r_1.sub = r_2.sub \ \& \ r_1.act = r_2.act \ \& \ r_1.con \cap r_2.con \neq \emptyset$ 
(2)     $\& (r_1.res = r_2.res \mid r_1.res \mapsto r_2.res)$  then
(3)    /* detection and elimination of the redundancy related to the “Deny-Overrides” combining algorithm */
(4)    if CA = Deny_Overrides then
(5)      if  $r_1.res = r_2.res$  then
(6)        if  $r_1.Effect = Deny$  then
(7)          EliminateRedundancy( $r_1, r_2$ )
(8)        else if  $r_2.Effect = Deny$  then
(9)          EliminateRedundancy( $r_2, r_1$ )
(10)     else if  $r_1.res \mapsto r_2.res$  then
(11)       if  $r_1.Effect = Deny$  then
(12)         EliminateRedundancy( $r_1, r_2$ )
(13)    /* detection and elimination of the redundancy related to the “Permit-Overrides” combining algorithm */
(14)    if CA = Permit_Overrides  $\& \ !(r_1.res \mapsto r_2.res)$  then
(15)      if  $r_1.Effect = Permit$  then
(16)        EliminateRedundancy( $r_1, r_2$ )
(17)      else if  $r_2.Effect = Permit$  then
(18)        EliminateRedundancy( $r_2, r_1$ )
(19)    /* detection and elimination of the redundancy related to the “First-Applicable” combining algorithm */
(20)    if CA = First_Overrides  $\& \ !(r_1.res \mapsto r_2.res)$  then
(21)      if  $r_1.No < r_2.No$  then
(22)        EliminateRedundancy( $r_1, r_2$ )
(23)      else if  $r_2.No < r_1.No$  then
(24)        EliminateRedundancy( $r_2, r_1$ )
  
```

ALGORITHM 3: Detection algorithm for redundancy related to combining algorithms. $CA_DetectRedundancy(r_1, r_2, CA)$.

r_1 and r_2 have the *redundancy related to the “Permit-Overrides” combining algorithm*.

- (3) In the condition of the “First-Applicable” policy/rule combining algorithm, for any two rules r_1 and r_2 in a policy or between policies, if r_1 and r_2 have the same subject attribute and action attribute, there is an overlapping relationship between the condition attributes of r_1 and r_2 , r_1 and r_2 have the same resource attribute, and the sequence of application is certain, then r_1 and r_2 have the *redundancy related to the “First-Applicable” combining algorithm*.

7.3. Elimination Algorithm for Redundancy Related to Combining Algorithms. The elimination algorithm for the *redundancy related to combining algorithms* is as shown in Algorithm 4. Algorithm 4 adds the corresponding attribute information about rules r_1 and r_2 to the SCEL, and reserve the rule r_1 . If $con_1 \cap con_2 = con_2$, we will delete the rule r_2 directly. If $con_1 \cap con_2 \neq con_2$, then we update the condition attribute of r_2 to $con'_2 = con_2 - con_1 \cap con_2$, so as to achieve the purpose of eliminating redundancies. This algorithm will update the SCEL, which will make administrators instantly track and search for the attribute information about the corresponding rules in the process of detecting and eliminating redundancies conveniently. The time complexity of Algorithm 4 is $O(1)$.

Input: two rules r_1 and r_2 having the *redundancy related to combining algorithms*
Output: (1) rules r_1 and r_2 where the *redundancy related to combining algorithms* has been eliminated
 (2) SCEL

```

(1)   $r_0.con \leftarrow r_1.con \odot r_2.con$ 
(2)  AddRuleToSCEL( $r_1, r_2$ )
(3)  if  $r_2.con = r_0.con$  then
(4)    delete  $r_2$ 
(5)  else
(6)     $r'_2.con \leftarrow r_2.con - r_0.con$ 
  
```

ALGORITHM 4: Elimination algorithm for redundancy related to combining algorithms. $EliminateRedundancy(r_1, r_2)$.

7.4. Algorithm for Adding Rules to the Resource Brick Wall. The algorithm for adding rules to the Resource Brick Wall is as shown in Algorithm 5. In Algorithm 5, rows (2)~(3) add the rules in the policy P one by one to the RBW. By calling the detection algorithm for the *redundancy related to combining algorithms*, row (4) accomplishes the detection and elimination of the *redundancy related to combining algorithms* between the new added rules and rules already in the RBW.

Input: (1) policy $P : \{r_1, r_2, \dots, r_n\}$, where r_i is a rule
 (2) initialized *RBW ResBrickWall*
 (3) policy/rule combination algorithm *CA*
Output: *RBW ResBrickWall* that is added rules
 (1) foreach $r \in P$ do
 (2) $ResBrickId \leftarrow ResBrickWall.Find(r)$
 (3) $ResBrickWall[ResBrickId].List.Append(r)$
 (4) $CA.DetectRedundancy(r_1, r_2, CA)$

ALGORITHM 5: Algorithm for adding rules to resource brick wall. *AddRuleToResBrickWall*(P , *ResBrickWall*, *CA*).

The time complexity under the worst condition of Algorithm 5 is $O(n)$.

7.5. Algorithm for Backtracking Detection and Elimination of Redundancies. The algorithm for backtracking detection and elimination of redundancies is as shown in Algorithm 6. In Algorithm 6, rows (1)~(2) accomplish backtracking resource bricks from the lower layer to the upper layer through the postorder traversal. By calling the detection algorithm for the *redundancy related to combining algorithms*, rows (3)~(6) accomplish the detection and elimination of the *redundancy related to combining algorithms* between rule r_1 of the upper layer resource bricks and rule r_2 of the lower layer resource bricks. The time complexity under the worst condition of Algorithm 6 is $O(n^2)$.

7.6. Detection and Elimination Algorithm for Redundancy Related to Combining Algorithms between Policies. The detection and elimination algorithm for the *redundancy related to combining algorithms* between policies is as shown in Algorithm 7. Firstly, we add each rule in *ResBrickWall*₂ to the resource brick structure of *ResBrickWall*₁, that is, combine the two input *RBWs* to one *RBW*. Then we detect and eliminate the *redundancy related to combining algorithms* in the combined *RBW*. Finally, by the algorithm for backtracking detection and elimination of redundancies, we detect and eliminate the *redundancy related to combining algorithms*.

In Algorithm 7, the time complexity of rows (1)~(2) is $O(n)$, the time complexity of rows (3)~(5) under the worst condition is $O(n)$, and the time complexity of row (6) under the worst condition is $O(n^2)$. Thus, the time complexity of Algorithm 7 under the worst condition is $O(n * n + n^2)$, that is, $O(n^2)$.

To sum up, the time complexity of Algorithm 1 is $O(m * n + n^2 + n^2 + n^2)$, that is, $O(m * n + n^2)$.

The problem of detecting and eliminating the *redundancy related to combining algorithms* between several policies and between policy sets all can be turned into the problem of detecting and eliminating the *redundancy related to combining algorithms* between two policies, and in this paper we do not go into details about it.

8. Experimental Results and Analysis

In order to assess the evaluation performance improvement of the PDP after detecting and eliminating the *redundancy related to combining algorithms*, the test policies are introduced first. We do the following experiments:

- (1) A comparison of the evaluation performance of the *XDPRE* before and after eliminating the *redundancy related to combining algorithms* is made.
- (2) A comparison in evaluation performance between Sun PDP and the *XDPRE* after eliminating the *redundancy related to combining algorithms* is made.

8.1. Test Policies. In order to simulate practical application scenarios, we select the following three XACML access control policies from practical systems [28–30]:

- (i) *Library Management System (LMS)*. The *LMS* provides access control policies by which a public library can use the web to manage books.
- (ii) *Virtual Meeting System (VMS)*. The *VMS* provides access control policies by which the web conference services can be managed.
- (iii) *Auction Sale Management System (ASMS)*. The *ASMS* provides access control policies by which items can be bought or sold online.

The policy of the *LMS* contains 720 rules, that of the *VMS* 945 rules, and that of the *ASMS* 1760 rules. In the light of actual requirement, the policies of the *LMS*, *VMS*, and *ASMS* need to be expanded to contain more rules. What we do is that according to the Cartesian product of different subjects, actions, resources, and conditions in all rules of a policy, we construct new rules and add them to the original policy. The number of rules in the policies of the *LMS*, *VMS*, and *ASMS* is expanded separately to 3000, 6000, and 9000.

At present, Sun PDP is a widely used policy decision point [31] and can process redundancies in a policy selectively according to the internal rule matching mechanism [32]. Sun PDP is adopted to evaluate requests as a decision engine in our following *redundancy related to combining algorithms* detecting and eliminating experiments. We choose Sun PDP in our experiments for two main reasons. First, Sun PDP is the first and the most widely deployed implementation of XACML evaluation engines. It has become the industrial standard. Second, Sun PDP is open source. To eliminate the performance factor of implementation languages, we implemented *XDPRE* in Java because Sun PDP is written in Java. The results of Sun PDP are compared with those of the *XDPRE* to verify that the *XDPRE* can effectively improve the evaluation performance of the policy decision point.

8.2. Generation of Test Requests. Dan et al. [33] put forward that policies are analyzed by *Change-Impact* in order to automatically generate access requests that conform to *Change-Impact*. The purpose is to improve the coverage of the test. The main idea is that conflicting policies or rules can be obtained by conflict detection tools according to the fact that

Input: (1) *RBW ResBrickWall*
 (2) current resource bricks *PresentResBrickId*
 (3) policy/rule combination algorithm *CA*

Output: *RBW ResBrickWall* where *redundancies related to combining algorithms* have been eliminated in a policy

- (1) foreach *LowerResBrickId* $\in$ *PresentResBrickId*
- (2) *BackTrackingDetectRedundancy*(*ResBrickWall*, *LowerResBrickId*, *CA*)
- (3) *UpperResBrickId* = *ResBrickWall.getUpperResBrickId*(*PresentResBrickId*)
- (4) foreach $r_1 \in$ *ResBrickWall*[*UpperResBrickId*].*List*
- (5) foreach $r_2 \in$ *ResBrickWall*[*PresentResBrickId*].*List*
- (6) *CA.DetectRedundancy*(r_1, r_2, CA)

ALGORITHM 6: Algorithm for backtracking detection and elimination of redundancies. *BackTrackingDetectRedundancy*(*ResBrickWall*, *PresentResBrickId*, *CA*).

Input: (1) two *RBWs*, *ResBrickWall₁* and *ResBrickWall₂*
 (2) policy/rule combination algorithm *CA*

Output: combined *RBW ResBrickWall₁*

- (1) for each *ResBrickId* $\in$ *ResBrickWall₁.IDList*
- (2) for each $r_2 \in$ *ResBrickWall₂*[*ResBrickId*].*List*
- (3) *ResBrickWall₁*[*ResBrickId*].*List.Append*(r_2)
- (4) for each $r_1 \in$ *ResBrickWall₁*[*ResBrickId*].*List*
- (5) *CA.DetectRedundancy*(r_1, r_2, CA)
- (6) *BackTrackingDetectRedundancy*(*ResBrickWall₁*, *root*, *CA*)

ALGORITHM 7: Detection and elimination algorithm for redundancy related to combining algorithms between policies. *DetectRedundancy-BetweenPolicies*(*ResBrickWall₁*, *ResBrickWall₂*, *CA*).

different policies or different rules in the same policy could make inconsistent results of evaluation for the same request, and that correlative access requests can be constructed for testing according to the conflicting policies or rules.

She et al. [34] proposed that access requests can be automatically generated to test the correctness of the PDP as well as the configured policies. They pointed out that the *Context Shema* which is defined by the XML Schema of the XACML describes all the structures of the access requests that might be accepted by the PDP, or all the valid input requests. This paper indicates that their developed X-CREATE can generate possible structures of access requests according to the Context Shema of the XACML. The policy analyzer obtains possible input values of every attribute from a policy. The policy manager adopts the method for random allocation to distribute the obtained input values into structures of access requests. Another test scheme is *Simple Combinatorial*, which can generate access requests according to all the possible combinations of attribute values of subject, action, and resource in the XACML policies. This paper also analyzes the advantages of these two schemes in debugging.

According to the actual requirement of the performance test, the method for combining *Change-Imapct*, *Context Shema*, and *Simple Combinatorial* is adopted to simulate the practical access requests.

8.3. Performance Tests and Comparisons. The XDPRE itself can be viewed as a policy decision point, so the evaluation

performance improvement of the XDPRE can verify the evaluation performance improvement of the PDP.

8.3.1. Comparison of Evaluation Performance of the XDPRE before and after Eliminating the Redundancy Related to Combining Algorithms. In order to assess the evaluation performance improvement of the XDPRE after eliminating the *redundancy related to combining algorithms* in policies, a comparison of evaluation time of the XDPRE before and after eliminating the *redundancy related to combining algorithms* is made. We generate 500, 1000, ..., 6000 access requests randomly to measure the evaluation time of the PDP. For the policies of the LMS, VMS, and ASMS, the variation of evaluation time of the XDPRE with the number of access requests is shown in Figure 4.

In Figure 4, we observe the following:

- (i) The evaluation time of the XDPRE increases when the number of access requests grows whether the *redundancy related to combining algorithms* in a policy is eliminated or not.
- (ii) The growth rate of the evaluation time of the XDPRE after eliminating the *redundancy related to combining algorithms* in a policy is less than that of the XDPRE without the *redundancy related to combining algorithms* elimination.
- (iii) For the policies of the LMS, VMS, and ASMS, when the number of access requests reaches 6000, the

FIGURE 4: Variation of evaluation time with access requests before and after eliminating redundancy related to combining algorithms.

evaluation time of the *XDPRE* after eliminating the redundancy related to combining algorithms in a policy is shortened by 9.09%, 11.78%, and 14.39%, respectively. Thus, the *XDPRE* can effectively enhance the PDP's evaluation performance by eliminating the redundancy related to combining algorithms in a policy.

8.3.2. Comparison in Evaluation Performance between Sun PDP and the *XDPRE* after Eliminating the Redundancy Related to Combining Algorithms. In order to further assess the evaluation performance improvement of the *XDPRE*, we compare the evaluation performance of Sun PDP with that of the *XDPRE* after eliminating the redundancy related to combining algorithms. We generate 500, 1000, ..., 6000 access requests randomly to measure the evaluation time of PDPs. For the policies of the *LMS*, *VMS*, and *ASMS*, the variations with the number of access requests of the evaluation time of Sun PDP and the *XDPRE* after eliminating the redundancy related to combining algorithms are shown in Figure 5.

In Figure 5, we observe the following:

- (i) The evaluation time of Sun PDP and the *XDPRE* after eliminating the redundancy related to combining algorithms increases when the number of access requests grows.
- (ii) The growth rate of the evaluation time of the *XDPRE* after eliminating the redundancy related to combining algorithms is less than that of Sun PDP.
- (iii) For the policies of the *LMS*, *VMS*, and *ASMS*, when the number of access requests reaches 6000, the evaluation performance of the *XDPRE* after eliminating the redundancy related to combining algorithms has been improved by 10.01%, 10.95%, and 12.04%, respectively, compared to that of Sun PDP. Therefore, the evaluation performance of the *XDPRE* after eliminating the redundancy related to combining algorithms has been improved by 11% on average as compared with that of Sun PDP. So the *XDPRE* can effectively improve the evaluation performance of PDPs.

FIGURE 5: Comparison in evaluation performance between Sun PDP and XDPRE after eliminating redundancy related to combining algorithms.

9. Conclusions

The *redundancy related to combining algorithms* detecting and eliminating engine termed the XDPRE is presented in this paper, which not only can detect and eliminate the *redundancy related to combining algorithms* within a policy, but also has the same ability as the PDP. This engine can evaluate access requests by loading the policies in which the *redundancy related to combining algorithms* has been eliminated and return the authorization result to the context processors and the PEP. A *Resource Brick Wall* is constructed by the XDPRE on the basis of the resource attribute of a policy's target attributes. By the *Resource Brick Wall* and the policy/rule combining algorithms, three theorems for detecting *redundancies related to combining algorithms* are presented. The *Resource Brick Wall* can effectively avoid unnecessary comparisons of resources at different levels, so it could optimize the comparison of rules and reduce traversing frequency of each rule in the policy set when detecting redundancies. As a result, it greatly improves the efficiency of detecting and

eliminating redundancies and the evaluation performance of the PDP. Finally, a detection and elimination algorithm for the *redundancy related to combining algorithms* is proposed.

There still remain some shortcomings in this paper. We simplified the discussion of the condition attribute in the process of detecting and eliminating the *redundancy related to combining algorithms*. And we focused on condition constraints of the clock time, but factors in practical life such as year and region were not considered. This part of work needs to be further solved and improved in our future research.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This work is supported by the Scientific Research Cultivation Fund of Xi'an University of Science and Technology in China.

References

- [1] M. F. Faridus, M. H. A. Wahid, N. Sabani et al., "SOA characterization for AND logic operation on SOA based NOLM," in *Proceedings of the IEEE Regional Symposium on Micro and Nanoelectronics (RSM '11)*, pp. 372–376, Kota Kinabalu, Malaysia, 2011.
- [2] Y. Sun, X. Guan, T. Liu, and Y. Qu, "An identity authentication mechanism based on timing covert channel," in *Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '12)*, pp. 832–836, Liverpool, UK, June 2012.
- [3] A. Sara, M. Azzam, O. Hadi et al., "New XACML-AspectBPEL approach for composite web services security," *International Journal of Web and Grid Services*, vol. 9, no. 2, pp. 127–145, 2013.
- [4] L. Griffin, B. Butler, E. de Leastar, B. Jennings, and D. Botvich, "On the performance of access control policy evaluation," in *Proceedings of the IEEE 13th International Symposium on Policies for Distributed Systems and Networks (POLICY '12)*, pp. 25–32, Chapel Hill, NC, USA, July 2012.
- [5] Y. Zheng, S. Li, and H. Qiu, "Networked coordination-based distributed model predictive control for large-scale system," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 3, pp. 991–998, 2013.
- [6] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, "First experiences using XACML for access control in distributed systems," in *Proceedings of the ACM Workshop on XML Security*, pp. 25–37, ACM, Fairfax, Va, USA, October 2003.
- [7] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti, "XAC-MUT: XACML 2.0 mutants generator," in *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW '13)*, pp. 28–33, IEEE, March 2013.
- [8] A. Bertolino, F. Lonetti, and E. Marchetti, "Systematic XACML request generation for testing purposes," in *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA '10)*, pp. 3–11, Lille, France, September 2010.
- [9] M. L. Pardal, M. Harrison, S. E. Sarma, and J. A. Marques, "Performance assessment of XACML authorizations for supply chain traceability web services," in *Proceedings of the 4th International Conference on Computational Aspects of Social Networks (CASoN '12)*, pp. 378–383, São Carlos, Brazil, November 2012.
- [10] K. Fatema and D. Chadwick, "Resolving policy conflicts—integrating policies from multiple authors," in *Advanced Information Systems Engineering Workshops: CAiSE 2014 International Workshops, Thessaloniki, Greece, June 16–20, 2014. Proceedings*, vol. 178 of *Lecture Notes Business Information Processing*, pp. 310–321, Springer, Berlin, Germany, 2014.
- [11] J. Fong and H. Shiu, "An interpreter approach for exporting relational data into XML documents with structured export markup language," *Journal of Database Management*, vol. 23, no. 1, pp. 49–77, 2012.
- [12] I.-H. Lim, T. S. Sidhu, M.-S. Choi et al., "Design and implementation of multiagent-based distributed restoration system in Das," *IEEE Transactions on Power Delivery*, vol. 28, no. 2, pp. 585–593, 2013.
- [13] M. Masi, R. Pugliese, and F. Tiezzi, "Formalisation and implementation of the XACML access control mechanism," in *Engineering Secure Software and Systems*, vol. 7159 of *Lecture Notes in Computer Science*, pp. 60–74, Springer, Berlin, Germany, 2012.
- [14] A. X. Liu, F. Chen, J. Hwang, and T. Xie, "Xengine: a fast and scalable XACML policy evaluation engine," in *Proceedings of the SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '08)*, pp. 265–276, 2008.
- [15] S. Marouf, M. Shehab, A. Squicciarini, and S. Sundareswaran, "Adaptive reordering and clustering based framework for efficient XACML policy evaluation," *IEEE Transactions on Services Computing*, vol. 4, no. 4, pp. 300–313, 2012.
- [16] A. Mourad and H. Jebbaoui, "SBA-XACML: set-based approach providing efficient policy decision process for accessing Web services," *Expert Systems with Applications*, vol. 42, no. 1, pp. 165–178, 2015.
- [17] H. Jebbaoui, A. Mourad, H. Otrok, and R. Haraty, "Semantics-based approach for detecting flaws, conflicts and redundancies in XACML policies," *Computers & Electrical Engineering*, vol. 44, pp. 91–103, 2015.
- [18] X. Chen and W. Xu, "Study of XACML policy based on description logic," *Computer Engineering*, vol. 39, no. 4, pp. 71–74, 2013.
- [19] X. Lei, J. Liu, J. Xiao, and J. Li, "An improved description method for role access control based on the XACML," *Computer Science*, vol. 35, no. 4, pp. 94–104, 2008.
- [20] E. Martin, "Automated test generation for access control policies," in *Proceedings of the Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pp. 752–753, ACM, Portland, Ore, USA, 2006.
- [21] T. Scheffler, S. Schindler, and B. Schnor, "Using AOP-based enforcement of prioritised XACML policies for location privacy," *International Journal of Internet Technology and Secured Transactions*, vol. 5, no. 1, pp. 84–104, 2013.
- [22] Q. Ni, E. Bertino, J. Lobo et al., "Privacy-aware role-based access control," *ACM Transactions on Information and System Security*, vol. 13, no. 3, article 24, 2010.
- [23] H. Hu, G.-J. Ahn, and K. Kulkarni, "Discovery and resolution of anomalies in web access control policies," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 6, pp. 341–354, 2013.
- [24] C.-M. Dinu, F. Pop, and V. Cristea, "Pattern detection model for monitoring distributed systems," in *Proceedings of the 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '11)*, pp. 268–275, Timisoara, Romania, September 2011.
- [25] A. J. Zhang, C. Ji, and J. Wang, "Security policy conflict detection for distributed system," *Advanced Materials Research*, vol. 282–283, pp. 173–176, 2011.
- [26] M. Guarnieri, M. A. Neri, E. Magri et al., "On the notion of redundancy in access control policies," in *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies (SACMAT '13)*, pp. 161–172, Amsterdam, The Netherlands, 2013.
- [27] W. H. Chen and N. N. Wang, "Research on XACML policy evaluation optimization technology," *Application Research of Computers*, vol. 30, no. 3, p. 70, 2013.
- [28] A. Pretschner and B. Baudry, "Test-driven assessment of access control in legacy applications," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pp. 238–247, Lillehammer, Norway, 2008.
- [29] T. Mouelhi, F. Fleurey, B. Baudry et al., "A model-based framework for security policy specification, deployment and testing,"

in *Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28–October 3, 2008. Proceedings*, vol. 5301 of *Lecture Notes in Computer Science*, pp. 537–552, Springer, Berlin, Germany, 2008.

- [30] T. Mouelhi, Y. L. Traon, and B. Baudry, “Transforming and selecting functional test cases for security policy testing,” in *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST '09)*, pp. 171–180, Denver, Colo, USA, April 2009.
- [31] D. El Kateb, T. Mouelhi, Y. Le Traon, J. Hwang, and T. Xie, “Refactoring access control policies for performance improvement,” in *Proceedings of the 3rd Joint WOSP/SIPEW International Conference on Performance Engineering (ICPE '12)*, pp. 323–334, New York, NY, USA, April 2012.
- [32] C. D. P. K. Ramli, H. R. Nielson, and F. Nielson, “The logic of XACML,” *Science of Computer Programming*, vol. 83, pp. 80–105, 2014.
- [33] N. Dan, S. Huaji, C. Yuan, and G. Jia-Hu, “Attribute based access control (ABAC)-based cross-domain access control in service-oriented architecture (SOA),” in *Proceedings of the International Conference on Computer Science and Service System (CSSS '12)*, pp. 1405–1408, Nanjing, China, August 2012.
- [34] W. She, I.-L. Yen, F. Bastani, B. Tran, and B. Thuraisingham, “Role-based integrated access control and data provenance for SOA based net-centric systems,” in *Proceedings of the 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE '11)*, pp. 225–234, Irvine, Calif, USA, December 2011.