Policy Modeling and Anomaly Detection in ABAC Policies

Maryam Davari and Mohammad Zulkernine

School of Computing, Queen's University, Kingston, Canada
{maryam.davari, mz}@queensu.ca

Abstract. Sensitive data is available online through web and distributed protocols that highlight the need for access control mechanisms. System designers write access control policies to represent conditions on accessing data. Access control policies can contain anomalies (redundancy, inconsistency, irrelevancy, and incompleteness) that can lead to security vulnerabilities. Detecting anomalies in large and complex policies is challenging due to the lack of effective analysis mechanisms and tools. In this paper, we introduce a formal tree-based policy modeling technique to represent, update, and analyze access control policies. Based on the proposed formal policy modeling, we propose an anomaly detection technique. Our approach focuses on Attribute Based Access Control (ABAC) policies as they are widely adopted. Also, they can provide high flexibility and enhance security and information sharing. The effectiveness of our policy modeling and anomaly detection technique has been demonstrated through experimental evaluation.

Keywords: Access Control Policies · Security · Attribute Based Access Control · Policy Modeling · Policy Anomalies · Anomaly Detection and Resolution.

1 Introduction

With the massive growth of online data ranging from web interfaces to web services, access control has become a critical part of systems. Access control policies are used to control which users (e.g., human, process, application) have access to which protected resources (e.g., databases, files) for performing which actions (e.g., write, read) under which environment conditions (e.g., time, location). A vital requirement to assure correct enforcement of access control policies is that policies have high qualities. Weak policies lead to conflicts at the policy enforcement phase that can result in availability issues (i.e., rejecting a legitimate user to access a resource) and security problems (i.e., allowing an illegitimate user to access a resource) [20]. Different methods (including matrix, event calculus, mathematics, and tree) have been used as policy modeling techniques to represent, update, and analyze access control policies [16]. However, these methods were developed in the network, EXtensible Access Control Markup Language (XACML) [3], and Role Based Access Control (RBAC) [25] domains rather than the Attribute Based Access Control(ABAC) [15] domain.

Access control policies can contain different types of anomalies. One of the critical problems of ABAC policies is redundancy detection and removal. Redundancy can be considered as an anomaly when the response time of access requests relies on the number of policies to be parsed. It affects the performance of policy evaluation. Redundancy detection and elimination is an optimized solution to improve the performance of policies as they are growing fast in size and complexity. In a policy set, multiple policies might overlap which means one access request matches more than one policy. In addition, policies conflict with each other which means that the policies not only overlap but also yield different decisions. This issue is referred to as inconsistency.

Irrelevancy and incompleteness are other problems with ABAC policies. Irrelevancy refers to a situation that a given policy is not suitable for any users' request. Incompleteness refers to a scenario when the current policies cannot cover an access request. In an access control system, it is important to assure that the access control policies do not result in permission leakage to unauthorized principals. For instance, an incomplete policy might lead to granting access to an intruder unintentionally. Detecting and removing irrelevant and incomplete policies can considerably enhance the security, performance, and usability of the system.

Anomaly detection in large and complicated policies is not easy as there are not enough effective analysis mechanisms and tools. Policy anomaly detection has attracted the attention of many researchers [1], [35], [13], [6]. Different policy analysis tools such as FIREMAN [1], Firewall Policy Advisor [35], FAME [13], and Capretta et al. [6] have been developed that focused on detecting firewall policy anomalies. They may not be directly applied to ABAC policies as some policy anomaly analysis mechanisms still need improvement [2]. Policy fields should be considered as a whole piece; while most of the existing approaches detect pairwise policy anomalies. Furthermore, ABAC policies can be multi-valued while the firewall policies are specified by fixed fields.

In this paper, we propose an access control policy modeling technique that includes a formalization of policy anomalies (redundancy, inconsistency, irrelevancy, and incompleteness). The technique adopts a tree data structure to represent ABAC policies. The policy tree maintains different information about users, resources, actions, environments, effects, and policy ids. Based on this modeling technique, policy anomalies are detected. We attempt to develop an approach not only for accurate anomaly detection but also for efficient anomaly resolution that checks if policies permit legitimate users to reach their goals and whether policies prevent intruders from reaching malicious goals.

The major contributions of this paper can be summarized as follows:

- The formalization of policy anomalies (redundancy, inconsistency, irrelevancy, and incompleteness) for the ABAC model.
- The design and implementation of the tree-based policy modeling for representing, updating, and analyzing access control policies.
- The design and implementation of anomaly detection and resolution techniques according to the anomaly formalization and policy modeling.

We deal with the lack of a large real dataset of ABAC policies to evaluate the proposed model. For this purpose, we generate different datasets of ABAC policies for experimental purposes. The experimental results show that our approach is efficient to detect anomalies in large-size policy sets. In addition, our approach simplifies policy insertion, modification, and deletion.

The rest of the paper is organized as follows: Section 2 overviews the ABAC model. Section 3 describes the policy anomaly formalization, access control policies modeling, and policy updating. In Section 4, we present anomaly mitigation techniques (detection and resolution). In Section 5, we discuss the experiments and analyze the results. Section 6 presents the related work. Finally, Section 7 outlines conclusions and future work.

2 Overview of ABAC

In what follows, we provide background information about the Attribute Based Access Control (ABAC) model and define notations for ABAC attributes. The ABAC model has been used for over two decades and different ABAC-based models have been developed. The flexibility features of ABAC make it a powerful access control model to promote security.

ABAC attributes. Attributes are the basic unit of ABAC policies:

User attributes: Attributes that describe the characteristics of a user. Let U be a finite set of users and Att_u is a finite set of user attributes. The value of attribute $a \in Att_u$ for user $u \in U$ is represented by the function $d_u(a, u)$. Some user attributes have a single value and some contain multiple values. Single value attributes ($Att_{u,1}$) have a unique value for each user (e.g., user id), and multiple value attributes ($Att_{u,m}$) are a set of single values (e.g., university courses).

Resource attributes: Attributes that describe characteristics of resources. Let R be a finite set of resources and Att_r is a finite set of resource attributes. The value of attribute $a \in Att_r$ for resource $r \in R$ is represented by the function $d_r(a, r)$.

Environment attributes: Attributes that represent the current states of system environments (Att_{env}) such as time, date, and date-time. The environment attributes help in achieving dynamic access decisions. In addition to providing more fine-grained access control, the value of attribute $a \in Att_{env}$ for environment $env \in Env$ is represented by the function $d_{env}(a, env)$.

There are users, user attributes, resources, resource attributes, actions (i.e., operations on resources), environments, environment attributes, conditions, and effects (decisions). The simplest form of a policy instance in ABAC is a tuple: $< U, Att_u, d_u, R, Att_r, d_r,\ action, Att_{env}, d_{env}, condition, effect >$. The following grammar can specify a policy:

$$
\begin{aligned}
&policy ::=< expression\ [; expression],\ effect > expression ::= u.Att_u = val\ |\\
&\qquad r.Att_r = val\ |\\
&\qquad u.Att_u = r.att_r\ |\\
&\qquad action = val\ |\\
&\qquad env.Att_{env} = val
\end{aligned}
$$

$effect := Permit \mid Deny$

Attribute hierarchies. We consider hierarchies in user, resource, and environment attributes of ABAC [5], [26], [19] that can enhance access control flexibility and attribute management. This hierarchy can be a tree or forest without cycles. Hierarchy can be written as $\succeq_{att}$ that implies if attribute values of a child node are assigned to a user (resource or environment), all the attribute values of its parent nodes are acquired by the user (resource or environment). Moreover, we assign a code to each node indicating the hierarchical position of each node that can facilitate the searching of the hierarchy tree.

3 Policy Modeling

In this section, policy anomalies are illustrated and formalized for ABAC policies. Then, the structure of the policy modeling that is necessary for policy analysis is described along with the complexity of the model in terms of time. Our policy modeling makes policy updating (insertion, modification, and deletion) more efficient as described in this section.

Table 1. ABAC Sample Policies.

Policy	Description
	<r.type=budget; u.projectLed=r.project; action=write; Permit>
P_1	A project leader can write the project budget.
	<r.type=schedule; u.projectLed=r.project; action=read; Permit>
P_2	A project leader can read the project schedule.
	<u.project=r.project; r.type=schedule; action=read; Permit>
P_3	A user working on a project can read the project schedule.
	<u.adminRole=auditor; r.type=budget; u.project=r.project; action=read; Permit>
P_4	An auditor assigned to a project can read the budget.
	<r.type=budget; u.projectLed=r.project; u.department=dep1; action=request; Permit>
P_5	The project leader of the department of "dep1" can request to know the budget of a project assigned to her.
	<u.adminRole=auditor; r.type=budget; u.project=r.project; u.department= dep_security; action=read; Deny>
P_6	An auditor assigned to a project of the department "dep_security" cannot read the budget.

3.1 Formalization of Policy Anomalies

We formalize policy anomalies (redundancy, inconsistency, irrelevancy, and incompleteness) that are utilized for policy analysis. For illustration, we use some sample policies (from [34]) that are shown in Table 1 (with the description for each policy). These policies focus on project management that controls access by the project leaders, department managers, employees, contractors, auditors, accountants, and planners to manage tasks, schedules, and budgets.

1) Redundancy anomaly (RED). Redundancy indicates similarities among policies. The policies may not exactly match but every field in one policy is a subset of or equal to the corresponding field in another policy. Detecting and removing redundancies can help in decreasing the policy set size and enhancing policy evaluation performance. An access control policy P_j is redundant if and only if

$\exists P_i \in ACP.$
$P_i.Att_u \succeq_u P_j.Att_u \wedge P_i.Att_r \succeq_r P_j.Att_r \wedge P_i.action = P_j.action \wedge P_j.Att_{env} \subseteq P_i.Att_{env} \wedge P_j.condition \subseteq P_i.condition \wedge P_i.effect = P_j.effect.$

For example, P_2 in Table 1 specifies that a project leader can read the project schedule and P_3 specifies that any user working on the project can read the project schedule. Therefore, P_2 is redundant in the case of the project leader.

2) Inconsistency anomaly (INCON). Inconsistency refers to a situation when there are at least two policies that conflict with each other. Reducing the number of inconsistencies can mitigate the need for conflict resolution activities. Consider access control policies P_i and P_j. These two policies are inconsistent if and only if

$P_i.Att_u \succeq_u P_j.Att_u \wedge P_i.Att_r \succeq_r P_j.Att_r \wedge P_i.action = P_j.action \wedge P_j.Att_{env} \subseteq P_i.Att_{env} \wedge P_j.condition \subseteq P_i.condition \wedge P_i.effect \neq P_j.effect.$

As an example, P_4 specifies that an auditor assigned to a project can read the budget, while P_6 specifies that an auditor assigned to a project of the department "dep_security" cannot read the budget. P_4 and P_6 are inconsistent.

3) Irrelevancy anomaly (IRR). Irrelevancy refers to a scenario where a policy is never triggered or required for any kind of access request. An access control policy is irrelevant if and only if

$\nexists req_i \in Req \,|\, req_i.Att_u \subseteq P.Att_u \wedge req_i.Att_r \subseteq P.Att_r \wedge req_i.action = P.action \wedge req_i.Att_{env} \subseteq P_i.Att_{env} \wedge req_i.condition \subseteq P_i.condition.$

As an example, P_5 indicates that a project leader of the department "dep1" can request the budget of a project assigned to her. However, this policy is irrelevant as a team leader already has access and can write to the budget of a project assigned to her based on P_1.

4) Incompleteness anomaly (INCOM). A set of access control policies is incomplete when an access request cannot be covered by the current policies. A set of policies is incomplete if and only if

$\exists req_i \in Req \,|\, req_i = (att_{ui} \in Att_u, att_{ri} \in Att_r, a_i \in action, att_{envi} \in Att_{env}) \nexists P \in ACP \,|\, req_i.Att_u \subseteq P.Att_u \wedge req_i.Att_r \subseteq P.Att_r \wedge req_i.action = P.action \wedge req_i.Att_{env} \subseteq P_i.Att_{env} \wedge req_i.condition \subseteq P_i.condition.$

As an example, assume a contractor working on a project asks for information about the project budget, but there is no policy to address the request.

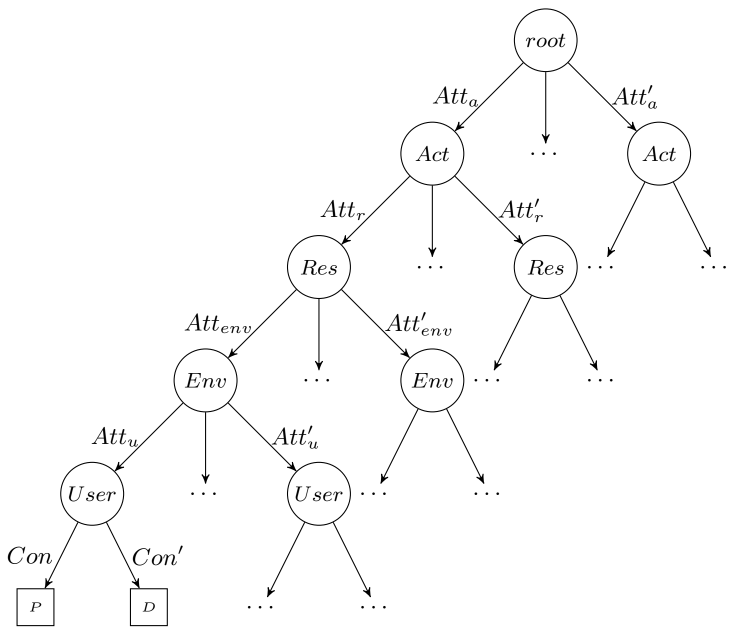

Fig. 1. The Policy Tree Structure for ABAC Policies.

3.2 Policy Modeling Structure

In this model, access control policies are represented by a single-rooted tree named policy tree. The policy tree prepares a simple representation of access control policies that makes the policy updating and anomaly detection technique easier (that will be discussed in the next section). This is a scalable representation for the analysis of access control policies. The tree is a type of decision diagram.

As shown in Figure 1, the first level of the tree is action nodes that represent all distinct actions of the system. Each action node points to the second level of the tree containing resource nodes. Then each resource node points to environment nodes in the third level of the tree. Each environment node points to the fourth level of the tree containing user nodes. Each user node can have one or two leaf nodes depending on effect and condition values. The leaf nodes contain a counter variable (that shows the frequency of policy definitions) and policy ids. The policy ids indicate a policy that is represented by that path and other policies that might be in anomaly with it.

To compute the time cost of constructing a policy tree, we consider the number of policies (n) and the number of node fields (p), where $p = max(num_u, num_r, num_a, num_{env})$. Num_u, num_r, num_a, and num_{env} are the number of user nodes, the number of resource nodes, the number of action nodes, and the number of environment nodes, respectively. The time cost of building a policy tree is $O(n\log_p n)$ and the time complexity of inserting a policy in the policy tree is $O(\log_p n)$.

Algorithm 1 Policy Tree Construction and Policy Insertion.

```
  Input: policy_tree, new_policy, updating_type
  node ← policy_tree.root
  type ← nil, current_node ← nil
 Output: policy_tree
 1: for each component $\in$ new_policy.components do
 2:   type ← nil
 3:   for each fnode $\in$ node.children do
 4:     if updating_type = Insert then
 5:       if component = fnode then
 6:         current_node ← fnode
 7:         type ← equal
 8:         break
 9:       end if
10:     end if
11:   end for
12:   if type = nil then
13:     for each fnode $\in$ node.children do
14:       if updating_type = Insert then
15:         if component $\supset$ fnode then
16:           fnode_left ← Add_node (fnode, left)
17:           Insert (fnode_left, component)
18:           current_node ← fnode_left
19:           break
20:         else if component $\subset$ fnode then
21:           fnode_right ← Add_node (fnode, right)
22:           Insert (fnode_right, component)
23:           current_node ← fnode_right
24:           break
25:         end if
26:       end if
27:     end for
28:   end if
29:   if current_node = nil then
30:     node ← Insert (node, component)
31:   end if
32:   node ← current_node
33:   if component.type = effect OR nodel.level=5 then
34:     Increase counter value of the path ()
35:     Add the policy id to the leaf node (component.policy_id)
36:   end if
37: end for
```

3.3 Policy Updating

Policies are written by system administrators and updated (inserted, modified, and deleted) occasionally. To insert a new policy into the policy tree, the tree nodes are searched and compared with the new policy components to find an appropriate position in the tree for the new policy as shown in Algorithm 1. If a node is exactly matched with the policy components (reaches the effect node), the counter value increases, and the policy id is added to the leaf node (lines 38-41). Otherwise, a node is added and the component is inserted (lines 12-33). The ordering of policies is crucial when policy components are a subset or superset of corresponding fields of the policy tree. Otherwise, the ordering is insignificant. In the case of disordered policies, some policies might not be used at all. When a policy component is a superset of a node (lines 12-22), the policy component is inserted in a node that is added right before the subset node. If a policy component is a subset of a node value (lines 23-33), the policy component is inserted in a node that is added right after the superset node. To modify or delete a policy in a policy tree, a similar process is applied to the policy tree. In case of deletion, the corresponding node is updated by decreasing the counter value and removing the policy id. If the counter value reaches zero, the path is deleted from the tree.

Algorithm 2 Anomaly Detection in the Policy Tree.

```
  Input: policy_tree
  RED ← {} , $INCON$ ← {} , $IRR$ ← {}
  Output: RED, INCON, IRR
  node ← policy_tree.root
 1: for each fnode ∈ node.children do
 2:   if fnode = “Permit” OR fnode = “Deny” then
 3:     if fnode.counter = 0 then
 4:       IRR ← fnode
 5:     else if fnode.counter > 1 OR Number of (fnode.policy_id) > 1 then
 6:       RED ← fnode
 7:     end if
 8:   else if fnode.level = 4 then
 9:     $P'$ ← fnode.children
10:     if there exists $p_i \in P'$ AND $p_j \in P'$ such that $p_i.effect \neq p_j.effect$ then
11:       INCON ← fnode
12:     end if
13:   end if
14: end for
15: Return RED, INCON, IRR
```

4 Anomaly Mitigation

Policy anomaly formalization and policy modeling for the ABAC model is developed to achieve the goal of effective anomaly mitigation (detection and resolution) as described in this section.

4.1 Anomaly Detection

The policy tree construction and update (described in Section 3.3) may avoid initiating anomalies in the system. They check for anomalies in the insertion time and store information (counter value, policy id) in leaf nodes that can help to identify anomalies. However, anomalies that may still exist in the policy tree can be detected by Algorithm 2.

The algorithm detects anomalies of access control policies by traversing the policy tree from the root to the leaf nodes. One simple approach to detect redundancy, inconsistency, and irrelevancy anomalies in the policy tree is to keep track of the counter variable (described in Section 3.2). If the counter value of a path is zero, it means that the policy was not referenced at all, and the policy is irrelevant (lines 3-4). If the counter value of a path is more than one, it means that the policy was defined several times and the policy is considered as a redundant policy (lines 5-7). If a user node has more than one child node, the policy is considered inconsistent (lines 8-12). Information about the corresponding policy ids is stored in the leaf node of the tree that can help to find anomaly policies in the policy set. Incompleteness anomalies cannot be detected by the policy tree as they can be detected by analyzing transactions (i.e., actions executed) in the system.

Algorithm 3 Policy Inconsistency Detection and Removal.

```
Input: $P_i$, $P_j$
1: if $P_i.effect \neq P_j.effect$ then
2:   if $P_i.Att_u \subseteq P_j.Att_u \wedge P_i.Att_r \subseteq P_j.Att_r \wedge P_i.action \subseteq P_j.action \wedge P_i.Att_{env} \subseteq P_j.Att_{env}$ then
3:     $u_k = (\forall u_f \in P_j.Att_u) - (\forall u_g \in P_i.Att_u)$
4:     Update($P_j, u_k$)
5:     $r_k = (\forall r_f \in P_j.Att_r) - (\forall r_g \in P_i.Att_r)$
6:     Update($P_j, r_k$)
7:     $a_k = P_j.action - P_i.action$
8:     Update($P_j, a_k$)
9:     $env_k = (\forall env_f \in P_j.Att_{env}) - (\forall env_g \in P_i.Att_{env})$
10:    Update($P_j, env_k$)
11:  else if $P_j.Att_u \subseteq P_i.Att_u \wedge P_j.Att_r \subseteq P_i.Att_r \wedge P_j.action \subseteq P_i.action \wedge P_j.Att_{env} \subseteq P_i.Att_{env}$ then
12:    Repeat lines 3-10 with swapping indexes $i$ and $j$.
13:  end if
14: end if
```

4.2 Anomaly Resolution

To have an anomaly-free policy tree, the detected anomalies (described in Section 4.1) should be resolved. Redundant policies can be safely removed that boost the runtime performance of the policy evaluator. Our policy tree can identify redundant policies at the insertion time simultaneously (described in Section 3.3). To detect a list of redundant policies in the policy tree, all the tree paths are traversed to reach the leaf nodes. When the leaf node contains more than one policy id, the policy is identified as redundant.

Eliminating policy conflicts by modifying policies is an intuitive solution by policy developers. However, it is remarkably difficult because of the following reasons [14]. The chance of conflicts in ABAC is high as it contains hundreds or thousands of policies. Conflicts are complicated; one policy may conflict with more than one policy. Moreover, in the distributed application, policies might aggregate. More than one administrator might maintain policies. Changing a policy might not resolve conflicts correctly, while it might modify the policy's semantics. The inconsistency issue can be addressed by eliminating the corresponding policies or limiting their applicability.

A policy might have an unintentional wide scope and cover policies that should be defined separately. The algorithm for inconsistency detection and removal is presented in Algorithm 3. Given two policies, the similarities between policy components (user attributes, resource attributes, actions, and environment attributes) are identified. In the case of similarity, redundant attributes are removed from the superset policy. In the case of irrelevancy, the service recommends deleting irrelevant policies. Incompleteness can be satisfied by adding incomplete policies or extending existing policies to cover incomplete policies.

5 Experimental Evaluation

The primary goal of this experiment is to evaluate the effectiveness and efficiency of our policy modeling and anomaly detection. As data sets may contain sensitive information, organizations are reluctant to share real-world policy sets. We perform our experiment on the project management policies adapted from [34] (described in Section 3.1) and extend it to support our work.

We create five synthetic datasets of policies (referenced as $DS1$, $DS2$, $DS3$, $DS4$, $DS5$) for the above-mentioned project management with different numbers of attributes and policies. Each policy can contain multiple users (maximum m), resources (maximum n), actions (maximum l), and environments (maximum k) and each one can have several attributes. Each user is uniformly and randomly assigned multiple attributes (maximum r). Similar to the user, each resource is uniformly and randomly assigned multiple attributes (maximum s). Likewise, each environment can have (maximum t) random attributes. The effect of each policy is uniformly and randomly selected as either Permit or Deny. Hierarchical relations between user attributes, resource attributes, and environment attributes are created. To generate user attribute hierarchies, some (about a quarter) of the user attributes are selected as parent nodes and the rest of the user attributes are considered child nodes. The child nodes are appended to the parent nodes. Similarly, resource and environment attribute hierarchies are generated. The anomaly detection algorithm is implemented in Java 11. The experiments and analyses were performed on an Intel Core i7 1.99 GHz processor with 16 GB of RAM and they do not require any specific architecture support.

Based on the above-mentioned parameters, we generate all combinations (referenced as P) of the users, user attributes, resources, resource attributes, actions, and environment attributes for each dataset. Other techniques (human experts) are used to identify true anomaly types (RED, INCON, and IRR) of each policy. The anomaly type is used in the evaluation part (described in the following paragraphs). Then, a subset of P refers to Q is uniformly and randomly selected (referenced as $Q \subset P$). This subset of policies is used for evaluating the anomaly detection technique. The policies might not be based on the specific real-world case study. However, they are intended to be analogous with policies in the application domain. Our modest size dataset can present the effectiveness of our algorithm as they contain anomalies that we discussed. In addition, they are complicated since each rule has lots of structures.

Table 2. Access Control Policy Datasets.

	$DS1$	$DS2$	$DS3$	$DS4$	$DS5$
$\mid U \mid$	50	100	150	200	250
$\mid Att_u \mid$	4	8	7	5	9
$\mid R \mid$	1000	2000	3000	4000	5000
$\mid Att_r \mid$	5	6	4	7	8
$\mid Env \mid$	10	20	30	40	50
$\mid Att_{env} \mid$	7	5	6	4	8
$\mid Q \mid$	50000	250000	500000	750000	1000000

Table 2 summarizes the size of sample datasets in terms of the number of users ($\mid U \mid$), the number of user attributes ($\mid Att_u \mid$), the number of resources ($\mid R \mid$), the number of resource attributes ($\mid Att_r \mid$), the number of environments ($\mid Env \mid$), the number of environment attributes ($\mid Att_{Env} \mid$), and the number of randomly selected policies ($\mid Q \mid$). To assess the efficiency of our anomaly detection technique, we report accuracy, precision, recall, and False Positive rate defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

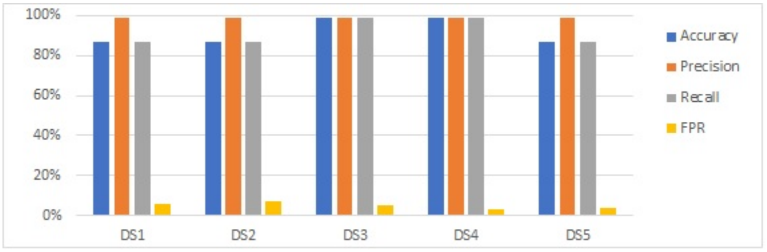

Fig. 2. Accuracy, Precision, Recall, and FPR Values Using the Datasets.

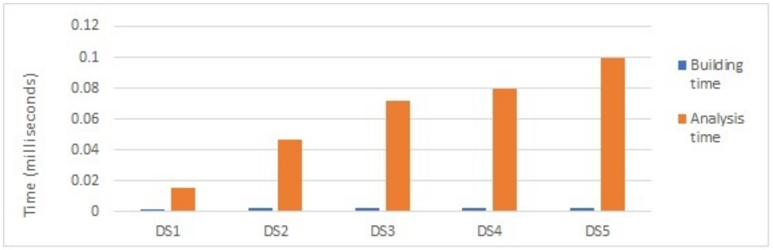

Fig. 3. Average Building Time and Analysis Time per Policy.

$$FPR = \frac{FP}{FP + TN} \tag{4}$$

Our techniques must achieve high accuracy, precision, and recall and a very low False Positive rate to be usable in practice. Policy trees are constructed for the generated datasets. Then, the anomaly detection technique is executed on each tree to detect anomalies. To calculate True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN), the detected anomalies are compared with the true anomaly types of policies. TP is the number of policies that are predicted as anomalies, and are truly anomalies. Furthermore, the type of detected anomalies and true anomalies should be identical. FP is the number of policies that are predicted as anomaly, while they are truly valid. FN is the number of policies that are predicted as valid, while they are truly anomalous. TN is the number of policies that are predicted as valid, and they are truly valid. As Figure 2 shows, the accuracy, recall, and precision of all five datasets are above 87%. Some datasets like DS3 achieved the maximum accuracy, recall, and precision of 99%, 99%, and 99%, respectively. Moreover, the observation shows that the FPRs are between 0-7%. It indicates that the anomaly detection technique can detect almost all anomalies and the overall performance is high. These promising results indicate the efficiency of our policy modeling and anomaly detection technique, especially in large datasets.

Furthermore, the building time of policy trees and analysis time of anomaly detection are shown in Figure 3. In general, the analysis time takes a longer time than the building time of the policy tree that is reasonable, especially for the large datasets. The negligible building time and reasonable analysis time make the tree-based policy modeling very efficient with respect to memory space and time.

6 Related Work

In the context of policy analysis, policies can be represented as graphs or trees to efficiently analyze and verify policies by using the tree or graph operations or features (e.g., graph traversal, graph union). Graph-based anomaly detection and resolution have been used for ensuring that policy specification fulfills system requirements and goals [24], [17].

Davy et al. [8] proposed a policy conflict algorithm by using an efficient policy selection process. The algorithm leveraged the information model to select an efficient set of deployed policies for analysis. They stored histories of previous policy comparisons in multiple tree data structures. Their goal was to improve the performance that relies on the relationships between deployed or newly added policies. However, their approach was not efficient and smart. It repeated over all deployed policies to make sure that they did not cause potential conflict. Mohan et al. [23] determined conflicts and inconsistencies in policies to protect biomedical data. They viewed biomedical ontology as a resource tree.

The formal method for policy analysis has been investigated [30], [31], [12]. Cuppens et al. [7] proposed a management mechanism for conflicts happening among permissions and prohibitions. In this approach, rules were grouped based on the organizations emitting them. This approach can effectively reduce the number of redundant policies. The model-checking technique that characterizes the system's model and specification into mathematical representation was also used to verify the correctness of system properties. First-order logic language Alloy [18] and formulae were used to detect security properties, conflicts, and inconsistencies [21], [4]. Another technique to support consistency checking of policies was Satisfiability Modulo Theories (SMT) solver. The SMT logic solvers separated the Boolean part of satisfiability checking from algorithms used property checking [32]. In addition, mutation testing [9] has been adopted for policy correctness [22]. Martin et al. [22] proposed a mutation verification to measure qualities specified for policy properties. After generating mutant policies having issues, it was verified if properties were held. When the properties did not hold, it meant the verification process detected faults in the faulty policies.

Some researchers [29], [28], [27], [10] applied data mining techniques (data classification and clustering) to detect policy conflicts. Shaikh et al. [29] used a modified C4.5 classification algorithm that has linear computational complexity to detect inconsistencies in access control policies. Moreover, Shaikh et al. [28] adopted different data classification techniques to detect incompleteness in access control policies. This approach consisted of three steps. First, attributes were ordered and Boolean expressions were normalized. Second, a decision tree (which was a modification of C4.5) was generated. Third, the proposed anomaly detection algorithm was executed on the decision tree. In clustering-based anomaly detection and resolution, policies were decomposed into clusters of rules, then the proposed approach was applied to each cluster [11].

Different techniques (graph-based, formal method, model-checking, mutation, and data mining) [16] have been explored to solve the problem of conflict and anomalies in access control policies. However, the computational complexity of some of these approaches had exponential growth and it could not deal with numerical and continuous values. In contrast, our policy anomaly detection and resolution techniques are not expensive in terms of time and space. In addition, most of the existing approaches were developed in the network management and RBAC domains rather than the ABAC domain. Although these works are useful in general, they are not suitable for anomaly detection and resolution in the ABAC domain. Furthermore, the previous works on policy analysis focused on some of the anomalies (inconsistency was the most investigated), while the primary goal of our approach is to detect all types of anomalies (redundancy, inconsistency, and irrelevancy).

7 Conclusion and Future Work

Enabling internet-based devices with a large number of access control policies might not provide appropriate security necessarily. One reason for this is the complexity of managing a large number of policies and potential vulnerabilities. Policies can contain different anomalies (redundancy, inconsistency, irrelevancy, and incompleteness). In this work, we have formalized policy anomalies and proposed a tree-based policy modeling that represents, updates, and analyzes ABAC policies. The policy insertion approach is considerably efficient as it can check for anomalies in the insertion time and facilitate policy updating. In addition, we have developed anomaly mitigation techniques based on anomaly formalization and policy tree modeling to detect and resolve policy anomalies. The experimental results show that the anomaly detection technique can achieve accuracy, recall, and precision of 87%. These promising results indicate the significant effect of the tree-based policy modeling. Moreover, short anomaly detection time illustrates the efficiency and effectiveness of our anomaly detection technique. As a part of future work, we will expand our policy modeling and anomaly detection technique to support other policy models [33].

References

1. Ehab S Al-Shaer and Hazem H Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE Infocom 2004*, volume 4, pages 2605–2616. IEEE, 2004.
2. Joaquin Garcia Alfaro, Nora Boulahia-Cuppens, and Frédéric Cuppens. Complete analysis of configuration rules to guarantee reliable network security policies. *International Journal of Information Security*, 7(2):103–122, 2008.
3. Anne Anderson, Anthony Nadalin, B Parducci, D Engovatov, H Lockhart, M Kudo, P Humenn, S Godik, S Anderson, S Crocker, et al. extensible access control markup language (XACML) version 1.0. *OASIS*, 2003.

4. Arosha Bandara, Seraphin Calo, Jorge Lobo, Emil Lupu, Alessandra Russo, and Morris Sloman. Toward a formal characterization of policy specification & analysis. In *Annual Conference of ITA (ACITA), University of Maryland, USA*. Citeseer, 2007.
5. Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. ABAC with group attributes and attribute hierarchies utilizing the policy machine. In *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control*, pages 17–28, 2017.
6. Venanzio Capretta, Bernard Stepien, Amy Felty, and Stan Matwin. Formal correctness of conflict detection for firewalls. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, pages 22–30, 2007.
7. Frédéric Cuppens, Nora Cuppens-Boulahia, and Meriam Ben Ghorbel. High level conflict management strategies in advanced access control models. *Electronic Notes in Theoretical Computer Science*, 186:3–26, 2007.
8. Steven Davy, Brendan Jennings, and John Strassner. Efficient policy conflict analysis for autonomic network management. In *Fifth IEEE Workshop on Engineering of Autonomic and Autonomous Systems (ease 2008)*, pages 16–24. IEEE, 2008.
9. Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
10. Maryem Ait El Hadj, Meryeme Ayache, Yahya Benkaouz, Ahmed Khoumsi, and Mohammed Erradi. Clustering-based approach for anomaly detection in XACML policies. In *SECRYPT*, pages 548–553, 2017.
11. Maryem Ait El Hadj, Ahmed Khoumsi, Yahya Benkaouz, and Mohammed Erradi. Formal approach to detect and resolve anomalies while clustering ABAC policies. *EAI Endorsed Transactions on Security and Safety*, 5(16), 2018.
12. Joaquin Garcia-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, and Stere Preda. MIRAGE: a management tool for the analysis and deployment of network security policies. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 203–215. Springer, 2010.
13. Hongxin Hu, Gail-Joon Ahn, and Ketan Kulkarni. Fame: a firewall anomaly management environment. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, pages 17–26, 2010.
14. Hongxin Hu, Gail-Joon Ahn, and Ketan Kulkarni. Discovery and resolution of anomalies in web access control policies. *IEEE Transactions on Dependable and Secure Computing*, 10(6):341–354, 2013.
15. Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (ABAC) definition and considerations (draft). *NIST Special Publication*, 800(162), 2013.
16. Amani Abu Jabal, Maryam Davari, Elisa Bertino, Christian Makaya, Seraphin Calo, Dinesh Verma, Alessandra Russo, and Christopher Williams. Methods and tools for policy analysis. *ACM Computing Surveys (CSUR)*, 51(6):1–35, 2019.
17. Amani Abu Jabal, Maryam Davari, Elisa Bertino, Christian Makaya, Seraphin Calo, Dinesh Verma, and Christopher Williams. Profact: A provenance-based analytics framework for access control policies. *IEEE Transactions on Services Computing*, 2019.
18. Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
19. Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *Proceedings of the 16th International Conference on World Wide Web*, pages 677–686, 2007.

20. Ninghui Li, Qihua Wang, Wahbeh Qardaji, Elisa Bertino, Prathima Rao, Jorge Lobo, and Dan Lin. Access control policy combining: theory meets practice. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 135–144, 2009.
21. Mahdi Mankai and Luigi Logrippo. Access control policies: Modeling and validation. In *5th NOTERE Conference (Nouvelles Technologies de la Répartition)*, pages 85–91, 2005.
22. Evan Martin, JeeHyun Hwang, Tao Xie, and Vincent Hu. Assessing quality of policy properties in verification of access control policies. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 163–172. IEEE, 2008.
23. Apurva Mohan, Douglas M Blough, Tahsin Kurc, Andrew Post, and Joel Saltz. Detection of conflicts and inconsistencies in taxonomy-based authorization policies. In *2011 IEEE International Conference on Bioinformatics and Biomedicine*, pages 590–594. IEEE, 2011.
24. Qun Ni, Dan Lin, Elisa Bertino, and Jorge Lobo. Conditional privacy-aware role based access control. In *European Symposium on Research in Computer Security*, pages 72–89. Springer, 2007.
25. Ravi S Sandhu. Role-based access control. In *Advances in Computers*, volume 46, pages 237–286. Elsevier, 1998.
26. Daniel Servos and Sylvia L Osborn. HGABAC: Towards a formal model of hierarchical attribute-based access control. In *International Symposium on Foundations and Practice of Security*, pages 187–204. Springer, 2014.
27. Riaz Ahmed Shaikh, Kamel Adi, and Luigi Logrippo. A data classification method for inconsistency and incompleteness detection in access control policy sets. *International Journal of Information Security*, 16(1):91–113, 2017.
28. Riaz Ahmed Shaikh, Kamel Adi, Luigi Logrippo, and Serge Mankovski. Detecting incompleteness in access control policies using data classification schemes. In *2010 Fifth International Conference on Digital Information Management (ICDIM)*, pages 417–422. IEEE, 2010.
29. Riaz Ahmed Shaikh, Kamel Adi, Luigi Logrippo, and Serge Mankovski. Inconsistency detection method for access control policies. In *2010 Sixth International Conference on Information Assurance and Security*, pages 204–209. IEEE, 2010.
30. Nikolaos I Spanoudakis, Antonis C Kakas, and Pavlos Moraitis. Gorgias-b: Argumentation in practice. In *COMMA*, pages 477–478, 2016.
31. Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. Analysis of XACML policies with SMT. In *International Conference on Principles of Security and Trust*, pages 115–134. Springer, 2015.
32. Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. Formal analysis of XACML policies using SMT. *Computers & Security*, 66:185–203, 2017.
33. D Verma, S Calo, Supriyo Chakraborty, Elisa Bertino, Christopher Williams, J Tucker, and Brian Rivera. Generative policy model for autonomic management. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–6. IEEE, 2017.
34. Zhongyuan Xu and Scott D Stoller. Mining attribute-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 12(5):533–545, 2014.
35. Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.