Mining Likely Properties of Access Control Policies via Association Rule Mining

JeeHyun Hwang[1], Tao Xie[1], Vincent Hu[2], and Mine Altunay[3]

[1] Department of Computer Science, North Carolina State University, Raleigh,
jhwang4@ncsu.edu, xie@csc.ncsu.edu
[2] Computer Security Division, National Institute of Standards and Technology, Gaithersburg,
vincent.hu@nist.gov
[3] Computing Division, Fermi National Laboratory, Batavia,
maltunay@fnal.gov

Abstract. Access control mechanisms are used to control which principals (such as users or processes) have access to which resources based on access control policies. To ensure the correctness of access control policies, policy authors conduct policy verification to check whether certain properties are satisfied by a policy. However, these properties are often not written in practice. To facilitate property verification, we present an approach that automatically mines likely properties from a policy via the technique of association rule mining. In our approach, mined likely properties may not be true for all the policy behaviors but are true for most of the policy behaviors. The policy behaviors that do not satisfy likely properties could be faulty. Therefore, our approach then conducts likely-property verification to produce counterexamples, which are used to help policy authors identify faulty rules in the policy. To show the effectiveness of our approach, we conduct evaluation on four XACML policies. Our evaluation results show that our approach achieves more than 30% higher fault-detection capability than that of an existing approach. Our approach includes additional techniques such as basic and prioritization techniques that help reduce a significant percentage of counterexamples for inspection compared to the existing approach.

1 Introduction

Access control mechanisms are used to control which principals (such as users or processes) have access to which resources in a system. Database management systems often adopt access control mechanisms to offer fine-grained access control to sensitive resources based on access control policies (in short as policies). In such a situation, identifying discrepancies between polices and their intended function is crucial because correct policy behaviors are based on the premise that the policies are correctly specified. These discrepancies may result in unexpected policy behaviors such as allowing malicious users to access sensitive resources. To increase our confidence on the correctness of policy behaviors, policies must undergo rigorous verification.

There are property verification tools [1, 2] available for policies specified in specification languages such as XACML (eXtensible Access Control Markup Language) [3] and Ponder [4]. Given a policy and its properties, property verification is to verify whether the policy satisfies the properties. If a property is not satisfied, a property verification tool produces counterexamples that violate properties. An example property for a policy in a grading system used by Fisher et al. [1] is that a student cannot assign grades. Any violations (that allow a student to assign grades) against the property expose faults in the policy. In addition, the quality of the properties can be measured based on their fault-detection capability. Our previous work [5] showed that the confidence on policy correctness based on property verification is dependent on the quality of the specified properties. In other words, policy authors require properties of high quality (which have a high chance to detect faults in the policy) to increase the confidence on policy correctness sufficiently.

While property verification is useful to detect faults, in practice, most policies are not equipped with properties. In addition, manually writing properties is not a trivial task for two reasons. First, the policy authors must have sufficient domain knowledge of a given policy to identify properties for the policy. Second, as the size of a policy increases and the structure of a policy becomes complex, identifying properties is more challenging.

To address these issues, we present an approach that automatically mines likely properties (from a policy) via association rule mining [6]. Association rule mining is used to discover correlations among data in a large database. When a policy includes many rules in a sophisticated structure, manually inspecting each policy behavior for fault detection is not trivial and error-prone. In such a situation, mined patterns of policy behaviors can be used to detect a fault in a policy [7].

In the policy context, we apply association rule mining to mine patterns of interest, called *likely properties* characterizing correlations of policy behaviors with regards to attribute values. For example, in the policy for a grading system, based on similar policy behaviors of a lecturer and a faculty member, our approach mines a property: if a lecturer is permitted to conduct actions (e.g., assign/modify) on grades, a faculty member is likely to be permitted to conduct the same actions on grades. We call these properties as *likely properties* because our approach mines properties that may not be true for all the policy behaviors but are true for most of the policy behaviors. In such a situation, *likely properties* may lead to a small number of violations. As these violations are deviations from the policy's normal behavior, these violations are special cases for inspection to determine whether these violations expose faults.

This paper makes the following three main contributions:

- We develop an approach that analyzes a policy under verification and mines *likely properties* characterizing correlations of policy behaviors with regards to attribute values.
- We verify a policy under verification with likely properties to check whether a policy includes a fault. Our fault-detection approach includes two techniques:

```
 1 If role = Faculty
 2   and resource = (ExternalGrade or InternalGrade)
 3   and action = (View or Assign) then Permit
 4 If role = TA
 5   and resource = (InternalGrade)
 6   and action = (Assign or Receive) then Permit // Faulty Line
 7 If role = Student
 8   and resource = (ExternalGrade)
 9   and action = (Receive) then Permit
10 If role = Family
11   and resource = (ExternalGrade)
12   and action = (Receive) then Permit
13 If role = Lecturer
14   and resource = (ExternalGrade or InternalGrade))
15   and action = (Assign or View) then Permit
16 Deny
```

Fig. 1. An example policy including a fault (in Line 6); "Receive" (instead of "View", which is correct) is specified.

		Assign	View	Receive
External Grade	Faculty	Permit	Permit	Deny
	TA	Deny	Deny	Deny
	Student	Deny	Deny	Permit
	Family	Deny	Deny	Permit
	Lecturer	Permit	Permit	Deny
Internal Grade	Faculty	Permit	Permit	Deny
	TA	Permit	Deny	Permit
	Student	Deny	Deny	Deny
	Family	Deny	Deny	Deny
	Lecturer	Permit	Permit	Deny

Fig. 2. Decision table for the policy in Figure 1 based on action relations.

the basic technique is to inspect counterexamples in no particular order, and the prioritization technique is to inspect counterexamples by the order of their fault-detection likelihood.

– We compare our approach with a previous related approach [8] in terms of cost and effectiveness. Our approach achieves more than 30% higher fault-detection capability than that of the previous related approach. Our approach such as the basic and prioritization techniques helps reduce a significant percentage of counterexamples for inspection compared to the existing approach.

The rest of the paper is organized as follows. Section 2 presents an illustrative example. Section 3 presents definitions of our proposed likely properties. Section 4 presents our fault-detection approach. Section 5 presents evaluation of our approach. Sections 6 and 7 discuss related work and issues. Section 8 concludes the paper.

Relation	Frequency	Confidence (%)
({Receive}, Permit) → ({View}, Deny)	3	100
({View}, Permit) → ({Assign}, Permit)	4	100
({View}, Permit) → ({Receive}, Deny)	4	100

Fig. 3. Implication relations $R1$ with 100% confidence.

Relation	Frequency	Confidence (%)
({Receive}, Permit) → ({Assign}, Deny)	2	66
({Assign}, Permit) → ({View}, Permit)	4	80
({Assign}, Permit) → ({Receive}, Deny)	4	80

Fig. 4. Implication relations $R2$ with at least 65% but less than 100% confidence.

Counterexamples	Fault Detection
TA is Permitted to Assign InternalGrades	
TA is Denied to View InternalGrades	detected
TA is Permitted to Receive InternalGrades	detected

Fig. 5. Counterexamples and their fault-detection capability.

2 Example

Figure 1 illustrates an example access control policy for a grading system in a university as if-else statements in code. Lines 1-3 include rules that allow a faculty member to assign or view ExternalGrade or InternalGrade. Lines 4-6 include rules that allow a Teaching Assistant (TA) to assign or receive InternalGrade. Lines 7-9 include rules that allow a student to receive ExternalGrade. Lines 10-12 include rules that allow a family member to receive ExternalGrade. Lines 13-15 include rules that allow a lecturer to assign or view ExternalGrade or InternalGrade. Line 16 is a tautology rule to deny requests that are not applicable in the preceding rules.

Figure 1 is a faulty version of the policy used by Fisher et al. [1]. The faulty version includes a fault at Line 6, where action attribute "Receive" is used instead of "View". Due to this fault, we observe two incorrect policy behaviors. First, a TA is *Denied* to *View* InternalGrade while a correct behavior is that a TA is *Permitted* to *View* InternalGrade. Second, a TA is *Permitted* to *Receive* InternalGrade while a correct behavior is that a TA is *Denied* to *Receive* InternalGrade.

We observe that actions over different roles may have similar policy behaviors. Figure 1 is a Role-Based Access Control policy (RBAC) [9]. In RBAC policies, one role'permissions may inherit another role's permissions based on role inheritance such as that a faculty member inherits all permissions of a TA in a policy. In such a situation, one's (e.g., a faculty member's) permissions may be dependent on another's (e.g., TA's) permissions. Based on this implication, we discover correlation of subjects with regards to their corresponding decisions. We can also discover correlation of actions with regards to their corresponding decisions. For example, in Figure 1, regardless of any roles or resources, if one

role (e.g., Faculty) is *Permitted* to *Assign* a resource (e.g., InternalGrade), the role is likely to be *Permitted* to *View* the resource. In the paper, we denote such a correlation as implication relation (based on action attributes). Formally, an attribute item $Item$ (v, dec) represents any request that includes v is evaluated to dec. For example, $Item$ ({Assign}, Permit) represents that any request that includes an "Assign" action is evaluated to be *Permitted*. We represent the example correlation as $\{Item$ (Assign, Permit)$\} \Rightarrow \{Item$ (View, Permit)$\}$ described in Section 3.

We next describe how to mine such implication relations. To mine implication relations (based on action attributes), we describes all possible request-decision pairs in a table in Figure 2. Each request requires three attribute values (such as subject, resource, and action attributes). Columns 1 and 2 show all possible combinations of resource and subject (role) attributes, respectively. Columns 3-5 describe the decisions (e.g., Permit or Deny) of a combination of a subject and a resource (in Columns 1 and 2) associated with an action "Assign", "View", or "Receive", respectively. For example, in the second row, given a role (Faculty) and a resource (ExternalGrade), the table describes decisions associated with action attributes such as "Assign", "View", or "Receive". Three requests, r_1 (Faculty, Assign, ExternalGrade), r_2 (Faculty, View, ExternalGrade), and r_3 (Faculty, Receive, ExternalGrade) are evaluated to "Permit", "Permit", and "Deny" (which are described in the second row), respectively.

Then, we feed these request-decision pairs into an association mining tool to mine implication relations. We set the confidence threshold as 65%, which is derived based on our preliminary experience. The confidence (described in Section 4) reflects likelihood of an implication relation. Figures 3 and 4 show mined implication relations $R1$ with 100%, and relations $R2$ with at least 65% but less than 100% confidence. For example, in $R1$, the relation $\{Item$ (Receive, Permit)$\}$ $\Rightarrow \{Item$ (View, Deny)$\}$ in Figure 3 indicates that if a subject is *Permitted* to *Receive* a resource r, the subject is *Denied* to *View* r with 100% confidence. An example case is that if a Student is *Permitted* to *Receive* ExternalGrades, then, a Student is *Denied* to *View* ExternalGrade (as described at fourth row in Figure 2). Column "Frequency" denotes the number of occurrences of such cases.

As implication relations in $R2$ cannot achieve 100% confidence, we find counterexamples violating the implication relations. If a counterexample is evaluated to be an unexpected decision, we say that the counterexample exposes a fault. The relation $\{Item$ (Assign, Permit)$\} \Rightarrow \{Item$ (Receive, Deny)$\}$ in Figure 4 indicates that if a subject is *Permitted* to *Assign* a resource r, the subject is *Denied* to *Receive* r with 80% confidence. As this relation cannot achieve 100% confidence, we can find a counterexample satisfying $\{Item$ (Assign, Permit)$\} \Rightarrow$ $\{\neg\ Item$ (Receive, Deny)$\}$. A counterexample against the implication relation is that a TA is *Permitted* to *Receive* InternalGrade (while the TA is *Permitted* to *Assign* InternalGrade). Note that the correct policy behavior is that a TA is *Denied* to *Receive* InternalGrade. Therefore, we inspect the counterexample and determine that the counterexample exposes the fault in the example policy. Fig-

ure 5 describes counterexamples, which do not satisfy the implication relations in $R2$. In Figure 5, two counterexamples are determined to expose the fault.

3 Definitions

This section presents definitions for attribute item set and implication relations. Let $\mathcal{S}$, $\mathcal{O}$, and $\mathcal{A}$, respectively, denote the set of all the subjects (e.g., user's role or rank), resources (e.g., file) and actions (e.g., write or read) in an access control system.

3.1 Attribute Item Set

An attribute item set is used to represent a policy behavior with regards to a specific set of attribute values v (e.g., faculty and file) and a decision dec (e.g., Permit). An attribute item $Item$ (v, dec) represents that any request that includes v is evaluated to dec. For example, $Item$ ({Faculty}, Permit) represents that any request that includes a faculty role is evaluated to be *Permitted*. Note that v may include multiple attribute values.

3.2 Implication Relations

An implication relation $\{Item_1\ (v_1,\ dec_1)\} \Rightarrow \{Item_2\ (v_2,\ dec_2)\}$ represents that, if a request r_1 including v_1 and values V of other attributes is evaluated to dec_1, then a request r_2 including v_2 and the same values V of other attributes is likely to be evaluated to dec_2.

In this paper, we propose implication relations based on subjects, actions, and subject-action relations, as presented next. Based on selection of attributes, other types of relations can be mined from a policy. We discuss these other implication relations in Section 7.

Implication relation of subject attribute item sets. We denote this implication relation as $\{Item_1\ (s_1,\ dec_1)\} \Rightarrow \{Item_2\ (s_2,\ dec_2)\}$ where s_1 and s_2 are subjects (i.e., $s_1 \in \mathcal{S}$ and $s_2 \in \mathcal{S}$). This implication relation indicates that dec_1 of a request including s_1 and values V of other attributes implies dec_2 of a request including s_2 and the same values V of other attributes. In a Role-Based Access Control (RBAC) policy [9], one role'permissions may inherit another role's permissions according to role inheritance. In such a situation, one role's permissions may be associated with another role's permissions. For example, Faculty inherits permissions of TA in a grading policy. We represent this role inheritance as $\{Item_1$ ({TA}, Permit)$\} \Rightarrow \{Item_2$ ({Faculty}, Permit)$\}$.

Implication relation of action attribute item sets. We denote this implication relation as $\{Item_1\ (a_1,\ dec_1)\} \Rightarrow \{Item_2\ (a_2,\ dec_2)\}$ where a_1 and a_2 are actions (i.e., $a_1 \in \mathcal{A}$ and $a_2 \in \mathcal{A}$). This implication relation indicates that $dec_1$of a request including a_1 and values V of other attributes implies dec_2 of a request including a_2 and the same values V of other attributes. For example,

in a grading policy, if a user is *Permitted* to *Assign* grades, the user is likely to be *Permitted* to *View* grades. In such a case, the "Assign" action is likely to be correlated with "View". We represent this case as $\{Item_1$ ({Assign}, Permit)} $\Rightarrow \{Item_2$ ({View}, Permit)}.

Implication relation of subject-action attribute item sets. We denote this implication relation as $\{Item_1\ (\{s_1,\ a\},\ dec_1)\} \Rightarrow \{Item_2\ (\{s_2,\ a\},\ dec_2)\}$ where s_1 and s_2 are subjects, and a is an action (i.e., $s_1 \in \mathcal{S}$, $s_2 \in \mathcal{S}$, and $a \in \mathcal{A}$). This implication relation indicates that dec_1 of a request including s_1, a, and values V of other attributes implies dec_2 of a request including s_2, a, and the same values V of other attributes. For example, in a grading policy, if a TA is *Permitted* to *Assign* grades, *Faculty* is likely to be *Permitted* to *Assign* grades. We represent this role inheritance with specific action *assign* as $\{Item_1$ ({TA, *Assign*}, Permit)} $\Rightarrow \{Item_2$ ({Faculty, *Assign*}, Permit)}. This implication relation considers both subjects and actions together.

Based on the preceding definitions, we mine relations of various attribute item sets. Each of implication relations focuses on mining relations of specific attribute items.

4 Approach

This section presents our approach for detecting faults in a policy using our likely-property verification techniques. Our approach includes three components: relation-table generation, association rule mining, and likely-property verification. The relation-table generation component takes a policy p as an input and generates tables based on attribute items in the policy p. The association rule mining component takes attribute items (from the table produced by the previous component) and mines our proposed implication relations r of attribute item sets. The likely-property verification component takes p and r as inputs and verifies p against r. The component produces verification reports based on whether the given likely properties p are satisfied; when a property is violated, counterexamples are generated accordingly. The policy authors inspect counterexamples to determine whether they expose faults. To detect faults effectively, we propose a prioritization technique to recommend the policy authors to inspect counterexamples by the order of their fault-detection likelihood.

4.1 Relation-Table Generation

Our approach first analyzes a policy p and generates a policy behavior report charactering all possible request-response pairs in the policy p. Our approach next analyzes the policy behavior report, and then generates relation tables (including all request-response pairs) that can be used as input for an association rule mining tool. For example, to mine implication relations of action attribute items (as shown in Figure 2), we generate a relation table that organizes all possible request-decision pairs. Based on this table, we generate our proposed attribute item sets used to mine implication relations.

4.2 Implication Relations of Attribute Items

Given attribute items, we use association rule mining [6] to mine relations of attribute items. We focus on mining implication relations, which are of the form $\{Item\ (v_1,\ dec_1)\} \Rightarrow \{Item\ (v_2,\ dec_2)\}$ described in Section 3. We use an association rule mining tool, called Apriori [10], that takes attribute items in a relation table as an input and generates implication relations of attribute item sets.

In association rule mining, thresholds such as *support* and $confidence$ are used to constrain generating association relations. Let t denote the total number of transactions that corresponds to the number of rows in a relation table. For example, Figure 2 includes 10 transactions. Let d denote the number of transactions including an attribute item X. The support supp(X) of X is $\frac{d}{t}$. We measure $confidence$, which is likelihood of an implication relation: $\text{confidence}(X \Rightarrow Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X)}$. These implication relations are *likely properties*, which are true for most of the policy behaviors and may lead to a small number of violations. Our rationale is that violations produced by likely-property verification deviate from the policy's normal behaviors and are special cases for inspection to determine whether these violations expose faults.

As mined implication relations can be many, our approach filters out mined implication relations with two mechanisms. First, we report only implication relations with confidence values over a pre-defined confidence threshold. As a confidence value measures likelihood of likely properties, likely properties with high confidence values are true for most of the policy behaviors. We set a confidence threshold based on our preliminary experience. Second, we report only implication relations each of which has fewer than n counterexamples where n is a pre-defined number. Consider a policy that is mostly correct and faults in the policy are not many. If the number of counterexamples (produced by verification of a likely property) is small, this property may deviate from normal policy behaviors. Therefore, we constrain the number of counterexamples produced for likely properties as less than n, i.e., mined likely properties with more than n counterexamples are filtered out and not reported. Based on these filtering mechanisms, we can reduce a large number of implication relations and report only reduced implication relations as likely properties.

4.3 Likely-Property Verification

Our approach next verifies the policy with the likely properties to check whether the policy includes a fault. Our rationale is that, as likely properties are true for most of the policy behaviors, counterexamples (which do not satisfy the likely properties) deviate from the policy's normal behaviors and are special cases for inspection.

Basic and Prioritization Techniques. A basic technique is to inspect counterexamples without any inspection order among the counterexamples. Since

Algorithm 1: Counterexample classification

```
Input: c_1, c_2, ..., c_n where each c_i is a counterexample, m, which is the largest
       number of counterexamples generated for a likely properties.
Output: CS_du, CS_1, ..., CS_m where each CS_j is a set of counterexamples.
1  CS_du := ∅; CS_1 := ∅; ...; CS_m := ∅; for i := 1 to n do
2      if c_i ∉ CS_du then
3          Flag := false ;
4          for j := 1 to m do
5              if c_i ∈ CS_j then
6                  CS_j = CS_j − {c_i} ;
7                  CS_du = CS_du ∪ {c_i} ;
8                  Flag := true ;
9          if Flag = false then
10             Prop := the property for which counterexample c_i is generated;
11             w := the number of counterexamples generated for Prop;
12             CS_w = CS_w ∪ {c_i} ;
13 return CS_du, CS_1, ..., CS_m;
```

the number of generated counterexamples can be large, manual inspection of the counterexamples can be tedious. To address the preceding issue, we propose a prioritization technique that classifies counterexamples into various counterexample sets based on their fault-detection likelihood. The technique evaluates counterexamples in each of the counterexample sets by the order of their fault-detection likelihood until a fault is detected. The prioritization technique maintains the same level of fault-detection capability of the basic technique when the policy contains a single fault.

We next describe how we classify counterexamples into counterexample sets CS_{du}, CS_1, ..., CS_n, based on their fault-detection likelihood. First, we give the highest priority to duplicate counterexamples, which are classified to CS_{du}. Duplicate counterexamples produced from different likely properties can be more suspicious to expose fault. Second, we investigate the number of counterexamples produced by likely properties to set priorities among counterexamples. As a likely property may lead to less number of counterexamples, the policy authors are required to verify less number of counterexamples to ensure the correctness of likely properties to be true for all policy behaviors. Given a property that has w counterexamples, we classify these counterexamples to CS_w ($1 \leq w \leq m$ where m is the largest number of counterexamples generated for a likely property). The pseudocode of the classification algorithm is in Algorithm 1. The policy authors first inspect counterexamples in CS_{du}. The policy authors then inspect counterexamples in CS_i by the order of CS_1, ..., CS_m ($1 \leq i \leq m$) until a fault is detected.

5 Evaluation

We next describe the evaluation results to show the effectiveness of our approach with four real-world access control policies as subjects.

5.1 Research Questions and Metrics

In our evaluation, we try to address the following research questions:

- RQ1: How higher percentage of faults are detected by our approach compared to an existing related approach [8]? This question helps to show that our approach can perform better than the existing approach in terms of fault-detection capability.
- RQ2: How lower percentage of distinct counterexamples are generated by our approach compared to the existing approach [8]? This question helps to show that our approach can perform better than the existing approach in terms of cost (i.e., the number of distinct counterexamples for inspection) for detecting faults.
- RQ3: For cases where a fault in a faulty policy is detected by our approach, how high percentage of distinct counterexamples (for inspection) are reduced by our prioritization technique (in terms of detecting the first-detected fault) over our basic technique? This question helps to show that our prioritization technique can perform better than the basic technique in terms of cost (i.e., the number of distinct counterexamples for inspection) for detecting the first fault.

To measure fault-detection capability in our evaluation, we synthesize faulty policies, $f_1, f_2, ..., f_n$ by seeding faults into a subject policy f_o, with only one fault in each faulty policy for ease of evaluation. Then, the chosen approach generates counterexamples for each faulty policy to detect the seeded fault. Note that we seed a single fault for f_i. For n faulty policies, n faults exist. Let $CP(f_i)$ be distinct counterexamples generated by the chosen approach for f_i. Let $Count(f_i)$ be the number of distinct counterexamples in $CP(f_i)$ for f_i. Let $DE(f_i)$ be the reduced number of distinct counterexamples by the prioritization technique to detect the fault in f_i for cases where the fault in f_i is detected by our approach.

- **Fault-detection ratio (FR)**. Let p be the number of faults detected by counterexamples (generated by the chosen approach) for $f_1, f_2, ..., f_n$. The FR is $\frac{p}{n}$. The FR is measured to address RQ1.
- **Counterexample count (CC)**. The counterexample count is the average number of distinct counterexamples generated by the chosen approach for each faulty policy. The counterexample count is $\frac{\sum_{i=1}^{n} Count(f_i)}{n}$. Note that a counterexample is synonymous to a request. The CC is measured to address RQ2. The CC is used to define the CRB metric below.
- **Counterexample-reduction ratio (CRB) for our approach over the existing approach**. Let CC_1 and CC_2 be counterexample counts (CCs) by our approach and the existing approach, respectively. The CRB is $(\frac{CC_2 - CC_1}{CC_2})$. The CRB is measured to address RQ2.

– **Counterexample-reduction ratio (CRP) for the prioritization technique over the basic technique**. Let $f'_1, f'_2, ..., f'_m$ be faulty policies that are detected by our generated counterexamples. The CRB is a percentage that measures the reduction ratio in terms of the number of the counterexamples for inspection to detect the first fault by the prioritization technique over the basic technique. The CRP is $(\frac{\sum_{i=1}^{m} Count(f'_i) - \sum_{i=1}^{m} DE(f'_i)}{\sum_{i=1}^{m} Count(f'_i)})$. The CRP is measured to address RQ3.

5.2 Evaluation Setup

We use fault types defined in a policy fault model [11] to automatically seed a policy with faults for synthesizing faulty policies, with only one fault in each faulty policy for ease of evaluation. We use four fault types: Change-Rule Effect (CRE), Rule-Target True (RTT), Rule-Target False (RTF), and Removal Rule (RMR). A CRE fault inverts a decision (e.g., change `Permit` to `Deny`) in a rule. An RTT fault indicates changing a rule to be applicable for any request. An RTF fault indicates changing a rule to be applicable for no request. An RMR fault indicates that a rule is missing. We seed one fault to form each of faulty policies, i.e., each synthesized faulty policy includes only a single fault.

For the inspection for our approach, we use a tool, called Margrave [1], that is verification tool for XACML policies. Margrave also has a feature that statically analyzes an XACML policy and produces all possible request-decision pairs in a summarized format. Given a faulty policy, Margrave generates all possible request-decision pairs to be used for generating relation tables. We next mine implication relations from the relation tables using an association rule mining tool [10]. Our approach filters out implication relations each of which produces at most five counterexamples.

We compare the results of our approach with those of a previous related approach [8]. Let a decision tree (DT) denotes the related approach that uses a decision tree to infer properties. Given request-decision pairs, DT learns policy behaviors and generates request-classification rules. Therefore, incorrectly classified requests (i.e., counterexamples) deviate from normal policy behaviors, and are required to be inspected. We specify a confidence threshold as 0.4% based on our tuning of evaluation setup for DT to generate similar counterexamples as our approach for the small sample of faulty policies used in the tuning of evaluation setup. In our evaluation, inspection of counterexamples (to determine whether the counterexamples expose faults) is automatically conducted by comparing the two decisions evaluated by a faulty policy and its corresponding original policy (that is assumed to be correct). However, in general, this inspection is often a manual process conducted by the policy authors.

5.3 Evaluation Subjects

In our evaluation, we use four policies specified in XACML [3]. XACML is an access control policy specification language. Figure 6 summarizes the characteristics of each policy. Columns 1-5 show the evaluation subject name, the number of

Policy	# Rules	# roles	# actions	# resource
codeD2	12	5	3	2
continue-a	298	5	5	26
continue-b	306	5	5	26
univ.	27	7	7	8

Fig. 6. Subjects used in our evaluation

		DT Approach		Basic Technique			Prioritization Technique			
Policy	# Pol	% FR	# CC	% FR	# CC	% CRB	% FR	# CC	% CRB	% CRP
code2D	12	66.6	4.0	83.3	1.1	72.5	83.3	1.1	72.5	27.3
univ	27	0.0	26.0	51.8	7.1	72.7	51.8	7.1	72.7	46.5
continue-a	33	21.2	85.4	66.6	39.8	53.4	66.6	39.8	53.4	44.5
continue-b	38	15.8	81.1	47.3	39.5	51.3	47.3	39.5	51.3	31.4
AVERAGE	27.5	25.9	49.1	62.3	21.9	55.5	62.3	21.9	55.5	38.5

FR : fault-detection ratio CC : counterexample count

CRB : counterexample reduction ratio for our approach over the existing approach [8]

CRP : counterexample reduction ratio for the prioritization technique over the basic technique

Fig. 7. Fault-detection capability results of Change-Rule Effect (CRE) faulty policies for each policy and each technique

rules, and distinct attribute values in the subject, resource, and action attributes in the policy, respectively. A subject attribute corresponds a role attribute since the policies are based on the Role-Based Access Control (RBAC) model [9]. We denote the number of roles, actions, and resources as # roles, # actions, and # resource, respectively. Policies such as `continue-a` include attributes to describe constraints (e.g., checking whether a role has conflicts with another role). Our approach does not use these attributes for mining implication relations. The largest policy consists of 306 rules. The `codeD2` is a modified version of the `codeD`[4] by adding rules for a Lecturer role. For grading, a Lecturer role has the same privileges as a Faculty role. Two of the policies, namely `continue-a` and `continue-b`, are examples used by Fisler et al. [1] to specify access control policies for a conference review system. The `univ` policy is an RBAC policy used by Stoller et al. [12]. As its original policy is not written in XACML, we specified its policy behaviors in XACML.

5.4 Results

We conducted our evaluation on a laptop PC running Windows XP SP2 with 1G memory and dual 1.86GHz Intel Pentium processor. In our evaluation, for a faulty policy, we also measure the total processing time of request-response-pair generation, likely-property generation, counterexample generation, and automated inspection for correctness of given counterexamples. For each faulty

[4] `http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/college`

policy (with at most 306 rules), our results show that the total processing time is less than 10 seconds.

We first show our detailed evaluation results for only Change-Rule-Effect faulty policies due to space limit. We then show our summarized evaluation results in Figure 8 for Rule-Target-True, Rule-Target-False, and Removal-Rule faulty policies. Figure 7 summarizes the detailed results for Change-Rule-Effect (CRE) faulty policies of each policy. Columns 1-2 show the evaluation subject name and the number of CRE faulty policies. Columns 3-11 show fault-detection ratio (denoted as "% FR"), counterexample count (denoted as "# CC"), CRB (for only the basic and prioritization techniques), and CRP (for only the prioritization technique) for each technique/approach, respectively. Let *Basic* and *Prioritization* denote our basic and prioritization techniques, respectively.

Results to address $RQ1$. In Figure 7, we observe that DT, *Basic* and *Prioritization* detect averagely 25.9%, 62.3%, and 62.3% (in Column "% FR") of CRE faulty policies, respectively. Our approach (including *Basic* and *Prioritization* techniques) outperform DT in terms of fault-detection capability. Our approach uses implication relations based on similar policy behaviors of different attributes values (e.g., Faculty and Lecturer). Therefore, if a faulty rule violates certain implication relations of attribute items, our techniques have better fault-detection capability than that of DT. However, DT constructs classification rules based on the number of the same decisions without taking into how different attribute values interact. Therefore, generated rules are rigid and often may easily miss certain correct policy behaviors. For example, in Figure 1, most requests that include a Faculty are evaluated to be *Permitted*. DT generates a classification rule that classifies requests including a Faculty to be *Permitted*. The rule is rigid since the rule's counterexamples reflects cases where a Faculty is *Denied* to take certain actions (e.g., a Faculty is *Denied* to *Receive* InternalGrades in Figure 1).

Results to address $RQ2$. Our goal is to detect a fault with as fewer counterexamples for inspection as possible. Intuitively, with more counterexamples to be inspected, fault-detection capability is likely to be improved. Our results show that our approach reduced the number of counterexamples by 55.5% (in Column "% CRB") over DT. As a result, we observe that our approach significantly reduced the number of counterexamples while our approach detected a higher percentage of faults (addressed in $RQ1$). In addition, our approach requires a small number of counterexamples for inspection compared with the number of all possible counterexamples. Given N_s subject, N_a action, and N_r resource values, the maximum number MAX_c of possible counterexamples is $N_s \times N_a \times N_r$. For example, for the `continue-b` policy, MAX_c is $5(N_s) \times 5(N_a) \times 26(N_r) = 650$ counterexamples. However, our approach generated only averagely 39.5 counterexamples (in Column "# CC") for inspection.

Results to address $RQ3$. *Prioritization* is a technique that enables to inspect counterexamples by the order of their fault-detection likelihood while keeping the same level of fault-detection capability of the *Basic* technique. Figure 7 shows

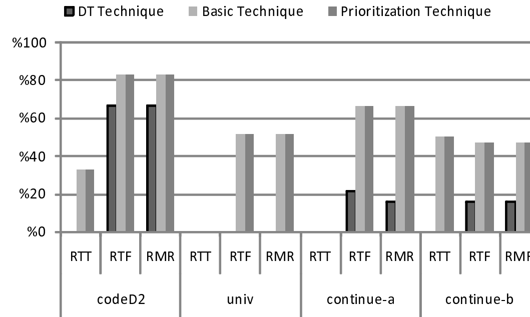

Fig. 8. Fault-detection ratios of faulty policies for each policy, each fault type, and each technique/approach

that *Prioritization* reduced averagely 38.5% of counterexamples (for inspection) (in Column "% CRP") over *Basic*.

Note that inspecting counterexamples could not always detect faults. The `continue-a` policy consists of 298 rules and is complex enough to handle corner cases for granting correct decisions to different roles (e.g., an Administrator and a Member for paper review). Consider that rel_3 $\{Item$ ({Write}, Permit)$\}$ $\Rightarrow$ $\{Item$ ({Read}, Permit)$\}$ represents an implication relation of "Write" and "Read" attribute items. For the `continue-a` policy (without any seeded fault), there exist 41 requests satisfying rel_3. There are 3 requests (counterexamples) violating rel_3. One counterexample is that Members are *Denied* to read their Password resources, while they are *Permitted* to write Password resources. Considering a Password resource as a critical resource and are *Denied* to be read, this counterexample does not reveal a fault in the policy. In our evaluation, assuming that an original policy is correct, such counterexamples could not detect faults. However, we suspect that inspecting these special cases of policy behaviors would still provide value in gaining high confidence on the policy correctness, reflected by the preceding password example.

In addition, Figure 8 illustrates the average fault-detection ratios for each policy, each other fault type, and each technique/approach. For other fault types, our results show that *Prioritization* and *Basic* achieve the highest faulty-detection capability.

6 Related Work

Prior work that is closest to ours is Bauer et al.'s approach [7]. They proposed an approach to mine association rules, which are used to detect miscofiguration in a policy. Our proposed approach is different from their approach in three aspects. First, given subject, action, and resource attributes, our approach mines various implication relations such as relations of subject, action, and subject-action attribute item sets. In contrast, their mined implication relations are limited

since their approach does not consider action attributes separately. Second, our approach includes a technique to prioritize which counterexamples should be inspected first based on their fault-detection likelihood while their approach does not include such a prioritization technique. Third, our approach exploits of characteristics of RBAC policies to mine implication relations whereas their approach uses historical access data to mine implication relations.

Our previous work [5] developed an approach for measuring the quality of policy properties in policy verification. Given user-specified properties, our previous approach measures the quality of the properties based on fault-detection capability. Our previous work [8] developed an approach to use machine learning algorithms (e.g., a classification algorithm) to mine policy properties automatically. Given request-decision pairs, this previous approach mines request-classification rules that classify requests to certain decisions. The rules there are based on a statistical policy-behavior model, which is statistically true. Therefore, faults can be likely to be detected when the policy violates this model. While this previous approach relies on classification rules, in this paper, we propose a new approach to mine likely properties based on implication relations (via association rule mining) and our evaluation shows that our new approach performs better than this previous approach.

7 Discussion

Our approach could be practical and effective to detect real faults in policies. Real faults may consist of one or several simple faults as described in our evaluation, and may cause a policy's behaviors to deviate from the policy's normal behaviors. Detecting real faults often depend on detecting such simple faults, which are shown to be effectively detected by our proposed approach. Our approach relies on attribute items (generated from a policy) for mining likely properties and thus could be applied to other types of access control policy beyond XACML policies.

In this paper, we do not consider implication relations based on resource, subject-resource, or action-resource attribute item sets. These implication relations can be used to derive valuable information indicating how resource attributes (with subject or action attributes) are correlated. Therefore, these relations may be useful for a policy with resource hierarchy (e.g., classified, unclassified, and shared resources) in a system. We plan to mine these implication relations to empirically investigate their effectiveness in terms of fault-detection capability.

8 Conclusions

We have developed an approach that analyzes a policy under verification and mines likely properties based on implication relations of subject, action, and subject-action attributes via association rule mining. Our approach also conducts likely-property verification to produce counterexamples, which are used to

help policy authors detect faults in a policy. We compared our two techniques in our approach with a previous related approach [8] in terms of fault-detection capabilities in four different XACML policies. Our results showed that our approach has more than 30% higher fault-detection capability than that of the previous related approach, which mines properties based on a classification algorithm. Our results showed that our basic and prioritization techniques reduce a significant percentage of counterexamples for inspection compared to the related technique. Moreover, the prioritization technique further reduced a number of counterexamples (for inspection) to detect a first fault over the basic technique.

Acknowledgment

This work is supported in part by NSF grant CNS-0716579 and a NIST contract.

References

1. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: Proc. 27th International Conference on Software Engineering. (2005) 196–205
2. Hughes, G., Bultan, T.: Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara (2004)
3. : OASIS eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml/ (2009)
4. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. In: Proc. International Workshop on Policies for Distributed Systems and Networks. (2001) 18–38
5. Martin, E., Hwang, J., Xie, T., Hu, V.: Assessing quality of policy properties in verification of access control policies. In: Proc. Annual Computer Security Applications Conference. (2008) 163–172
6. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. 20th International Conference on Very Large Data Bases. (1994) 487–499
7. Bauer, L., Garriss, S., Reiter, M.K.: Detecting and resolving policy misconfigurations in access-control systems. In: Proc. 13th ACM Symposium on Access control Models and Technologies. (2008) 185–194
8. Martin, E., Xie, T.: Inferring access-control policy properties via machine learning. In: Proc. 7th IEEE Workshop on Policies for Distributed Systems and Networks. (2006) 235–238
9. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. **4**(3) (2001) 224–274
10. Borgelt, C.: Apriori - Association Rule Induction/Frequent Item Set Mining. http://www.borgelt.net/apriori.html/ (2009)
11. Martin, E., Xie, T.: A fault model and mutation testing of access control policies. In: Proc. 16th International Conference on World Wide Web. (2007) 667–676
12. Stoller, S.D., Yang, P., Ramakrishnan, C., Gofman, M.I.: Efficient policy analysis for administrative role based access control. In: Proc. 14th ACM Conference on Computer and Communications Security. (2007) 445–455