International Journal of Information Security, Springer, 2016
DOI: 10.1007/s10207-016-0317-1

A Data Classification Method for Inconsistency and Incompleteness Detection in Access Control Policy Sets

Riaz Ahmed Shaikh · Kamel Adi · Luigi Logrippo

Received: date / Accepted: date

Abstract Access control policies may contain anomalies such as incompleteness and inconsistency, which can result in security vulnerabilities. Detecting such anomalies in large sets of complex policies automatically is a difficult and challenging problem. In this paper, we propose a novel method for detecting inconsistency and incompleteness in access control policies with the help of data classification tools well known in data mining. Our proposed method consists of three phases: firstly, we perform parsing on the policy data set; this includes ordering of attributes and normalization of Boolean expressions. Secondly, we generate decision trees with the help of our proposed algorithm, which is a modification of the well-known C4.5 algorithm. Thirdly, we execute our proposed anomaly detection algorithm on the resulting decision trees. The results of the anomaly detection algorithm are presented to the policy administrator who will take remediation measures. In contrast to other known policy validation methods, the proposed method provides means for handling incompleteness, continuous values and complex Boolean expressions. In order to demonstrate the efficiency of our method in discovering inconsistencies, incompleteness and redundancies in access control policies, we also provide a proof-of-concept implementation.

Riaz Ahmed Shaikh
Computer Science Department, Faculty of Computing & Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia.
E-mail: rashaikh@kau.edu.sa

Kamel Adi · Luigi Logrippo
Computer Science & Engineering Department, Université du Québec en Outaouais, Gatineau, Québec, Canada.
E-mail: kamel.adi@uqo.ca, E-mail: luigi.logrippo@uqo.ca

Keywords Access Control, Data classification, Inconsistency, Incompleteness, Policy Validation

1 Introduction

In enterprise environments, permission to access data for various purposes is regulated by complex policies. These policies can be expressed in different forms, for example: natural language, XML-based formats (such as XACML [32]), or firewall code. Because of their complexity, distribution and size, these policies can contain anomalies or errors, which can result in security vulnerabilities. We will use the term *anomalies* to refer to potential errors in the policies, for which *inconsistency* and *incompleteness* are common examples. *Inconsistencies* are situations where for a specific situation, different incompatible policies can apply [9,35]. For example, a policy administrator may formulate the following policy: "*Only* doctors are allowed to read and write on patient files". Then later, as an afterthought, she may add: "Nurses can also read it". Probably the more recent policy is meant to be an exception to the first, however the policy administrator or auditor should be made aware that an exception has been created. *Incompleteness* is the existence of situations for which no policy applies [9,34]. In this case, not all possible combinations of attribute values have been considered in the policies. For example, if a policy considers the days of the week to decide an outcome, are all days in the week considered? If the time of the day is used, are all hours in the day considered? Some of these anomalies can be exploited by attackers, including the author of the policy, to obtain unintended access or to compromise integrity. The existence of anomalies may also be a symptom of poorly maintained policy sets. While anomalies exist frequently in sets of access control rules, and can be normal, as in the case of exceptions, the policy administrators and auditors should be aware of them. The detection of anomalies will be also called policy *validation*.

Before going further, we will introduce some terminology that will be used in the rest of the paper. The term Access control *policy* is used in two meaning. One (the one used so far) is to denote informally stated principles that can be used in organizations to regulate access control to data. Another meaning is to denote sets of related access control *rules*. *Rules* are precise statements that convey specific information on what decision should be taken if certain subjects request to perform certain operations on certain data. In many access control systems, rules can be either positive (to allow access) or negative (to deny it). It is in these systems that inconsistency and incompleteness can occur [9]. Notable examples of such systems are ABAC [20,32] and OrBAC [18], as well as some industrial tools. The terms policy sets and rule sets are synonyms in this paper.

Rephrasing the previous definitions in this terminology, a set of rules, or a policy set, is *inconsistent* if it is possible to find in it rules that lead to opposite decisions, such as both denial and allowance of access requests in certain cases; it is *incomplete* if for some access requests no applicable rule can be found. Both inconsistency and incompleteness have been known for some time [9], but most detection methods deal only with inconsistency.

Inconsistency and incompleteness checks can be performed by policy administrators or by auditors. Large organizations are known to require hundreds and thousands of rules, and so automated tools are necessary. Such tools have been developed for inconsistency only [4,37,11,5,28,7,1]. Several researchers have tried to solve the problem of detecting inconsistencies using methods based on formal logic. However, this approach is difficult to implement and inefficient for large policy sets. Also, existing methods are inefficient in handling continuous values (for example time), which are common in access control policies.

Incompleteness is addressed in many systems by implicit meta-rules such as: all behaviors that are not explicitly allowed are forbidden. Inconsistency is also usually addressed by meta-rules. A commonly used meta-rule is based on ordering rules by priority and applying only the rule of higher priority in case several rules become applicable. In XACML, conflict resolution meta-rules (override) can be provided by the user. Examples are deny overrides, permit overrides, first-applicable, etc. In firewalls the first-applicable meta-rule is used: rules are executed top-down and only the first-applicable rule is executed [13], later rules that may contradict it are ignored. However, the result of the application of these blanket meta-rules may not always reflect the intention of the security administrator or of the author of the policies. Therefore, an automated mechanism or tool is highly needed through which policy administrators can easily detect anomalies and validate policy sets.

In this work, we describe a novel method for detecting anomalies, both inconsistency and incompleteness, in policy sets. For this purpose we propose the use of data mining techniques, in particular an adaptation of the data classification algorithm C4.5. Our method offers:

– Simultaneous detection of inconsistency and incompleteness.
– Ability to detect redundant rules in the policy set.
– Ability to handle numeric / continuous values in an efficient manner.
– Ability to handle Boolean expressions.
– Ease of implementation.
– Visualization support that can help to identify relations and patterns in unstructured policy sets.

To the best of our knowledge, we are the first authors to propose the use of data classification algorithms to detect anomalies in access control policies. A characteristic of our method is to handle continuous values and Boolean expressions efficiently, and it is particularly useful that in our method inconsistency and incompleteness are detected simultaneously. The method we will describe is generic, i.e. independent of any underlying policy specification language.

Our proposed anomaly detection method consists of three phases:

1. *Normalizing and formatting input data*: Rules in policy sets are not usually in uniformly structured form. Some rules are simple but others may contain complex Boolean expressions of variable lengths. In order to use data classification algorithms, we need to order attributes and transform Boolean expressions into normalized forms.
2. *Constructing decision trees*: For decision trees we propose two algorithms. The first algorithm is the Discrete Decision Tree Generator (D-DTG) algorithm, which is used for sets of rules which only contain discrete attributes. The second algorithm is the Decision Tree Generator (DTG) algorithm, which is used for sets of rules which contain both discrete and numeric attributes.
3. *Executing the anomaly detection algorithm*: Once the decision tree is created we apply the proposed

anomaly detection algorithm to detect any inconsistency or incompleteness in access control policies.

Anomaly resolution is not the subject of this paper, we will briefly mention methods that can be used to address anomalies. Once detected, inconsistency can be manually addressed by removing rules or narrowing their range of applicability. For example, a rule may have an unintended wide scope that covers cases that should be handled by other rules. Incompleteness can be addressed by adding rules or enlarging the range of applicability of existing rules.

The rest of the paper is organized as follows. Section 2 introduces concepts and definitions. Section 3 provides brief overview of data classification algorithms. Section 4 contains a description of our proposed anomaly detection method. Section 5 presents the subject of the normalization of input data. Section 6 contains theoretical and implementation analysis and evaluation of the proposed method. Section 7 presents related work and comparison of the proposed method with existing methods. Finally, Section 8 concludes the paper.

2 Concepts and Definitions

2.1 Structure of rules

In our method, consistent with both data classification algorithms and access control systems such as ABAC, access control rules are described as ordered collections of attributes. These attributes are classified into two types: 1) *Non-category* attributes and 2) *Category* attributes. Non-category attributes are decision-making attributes, such as role, subject, location, time, etc. Each non-category attribute contains a qualitative or quantitative value. On the other hand, category attributes express the decision of the rule. Each rule contains only one category attribute that represents the class of rules to which the rule belongs. For these attributes we consider here only the values {*Allowed*, *Denied*}. In some access control policy languages [32], the category attributes can also take values such as '*Indeterminate*', '*Not Applicable*', etc. We will not consider these additional values for simplicity, but our technique can be extended to support them.

Further, we assume that all rules in a policy set contain the same ordered set of attributes. Rules that are missing some attributes can be completed by using default values, and this is one of the goals of the formatting phase.

Let policy set $\Re$ be a non-empty set of rules ($\Re = \{R_1, R_2, \dots, R_n\}$). Each rule $R \in \Re$ consists of a non-empty, finite number of elements A_i, each of the form $N = V$, where N is the name of the attribute and V is a set of values. One of these attributes, which we will always write at the end, is the category attribute Permission. This one will have exactly one value in each rule, either *Allowed* or *Denied*. The others are non-category attributes. Rules $R_i \in \Re$ will be written as follows:

$R_i : A_1 \wedge A_2 \wedge \dots \wedge A_n \rightarrow Permission.$

For example, consider the following rule:

R: Role={Doctor} $\wedge$ Resource={Medical_record} $\wedge$ Operation={Write} $\rightarrow$ Permission={Allowed}.

In this example, *role*, *resource* and *operation* are the non-category attributes of the rule and *Permission* is the category attribute of the rule. Doctor, Medical_record, etc. are the values of the attributes.

Note that only the AND ($\wedge$) operator has been used in the definition of rule structure. If a rule contains Boolean expressions which contain different operators (e.g. OR, NOT), then it is split in several rules, as we shall see in Section 5.2.

Hereafter, we formally define *inconsistencies* and *incompleteness* from the perspective of data classification.

2.2 Inconsistencies

We have a *direct inconsistency* when two rules present in a policy set lead to direct contradictory conclusions: suppose that one rule states that user x is allowed access to resource r and another rule states that user x is denied access to the same resource with the same attribute values. The formal definition of direct inconsistency is given below.

Definition 1: Let $\upsilon(R_i.A_j)$ denote the set of values assigned to an attribute A_j in rule R_i. Rules $R_i, R_j \in \Re$ are *mutually inconsistent* if and only if

1. $\forall A_k \in A, \quad \upsilon(R_i.A_k) \cap \upsilon(R_j.A_k) \neq \emptyset$ and
2. $\upsilon(R_i.Permission) \neq \upsilon(R_j.Permission)$.

Informally, condition one in the above definition states that each element of the set of non-category *attribute-values* of R_i has non-empty intersection with each corresponding element of the set of *attribute-values* of rule R_j and condition two states that the category *attribute-value* of rule R_i is different from the category *attribute-value* of rule R_j. The non-empty intersection condition will be deemed to be satisfied when default values are used on either rule.

Example 1. Let us assume that the policy set $\Re$ consists of the following two rules.

R_1: Subject={Alice,Bob} $\wedge$ Object={O_1} $\wedge$ Operation={Write} $\wedge$ Day={Tue,Wed,Thu,Fri} $\rightarrow$ Permission={Allowed}

R_2: Subject={Alice} $\wedge$ Object={O_1,O_2} $\wedge$ Operation={Read,Write} $\wedge$ Day={Fri,Sat,Sun,Mon} $\rightarrow$ Permission={Denied} .

The set of values assigned to the attributes in these two rules are overlapping and their permissions are different. Then, according to Definition 1, rule R_1 and R_2 conflict with each other.

There is an *indirect inconsistency* if an inconsistency is created by performing the union of the sets of rules in consideration. We can also have dynamic inconsistencies, which can occur when new rules are generated as the result of events such as delegation. Again, the newly created rules are inconsistent with the old rules if an inconsistency is created by performing the union of the old and the new rules.

Example 2. Assume that a system contains two sets of rules. The first set ($\Re$) contains static rules like,

R_1: Subject={Alice} $\wedge$ Operation={Create} $\wedge$ Object={Account} $\rightarrow$ Permission={Allowed}

R_2: Subject={Bob} $\wedge$ Operation={Create, Write} $\wedge$ Object={Account, Ledger} $\rightarrow$ Permission={Denied}.

The second set ($\Re'$) may contain rules which are created in the system at runtime. For example, if Alice delegates her rights to Bob then the following rule is generated into $\Re'$.

R_3: Subject={Bob} $\wedge$ Operation={Create} $\wedge$ Object={Account} $\rightarrow$ Permission={Allowed}

Then, according to Definition 1, R_2 and R_3 are mutually inconsistent.

2.3 Incompleteness

As mentioned, incompleteness is the existence of situations for which no rule is defined. The formal definition of incompleteness is presented below.

Definition 2: Let $\Upsilon(A_j)$ denote the set of all possible values that can be assigned to an attribute A_j. Let $\upsilon(R_i.A_j)$ denote the set of values assigned to an attribute A_j in rule R_i. The policy set $\Re$ is *incomplete* if and only if

$$\bigcup_{i=1..m} \prod_{j=1}^{n} \upsilon(R_i.A_j) \subset \prod_{j=1}^{n} \Upsilon(A_j).$$

Here $\subset$ denotes a proper subset, $\prod$ denotes Cartesian product, m denotes the number of rules ($m = |R|$) and n denotes the number of non-category attributes ($n = |A|$).

Example 3. Let us assume that a policy set $\Re$ contains three rules and each rule contains two non-category attributes: *Trusted* and *Weekend*. For simplicity, we assume that both are binary attributes. Assume that the policy set $\Re$ contains the following three rules.

R_1: Trusted={No} $\wedge$ Weekend={No} $\rightarrow$ Permission={Denied}

R_2: Trusted={No} $\wedge$ Weekend={Yes} $\rightarrow$ Permission={Denied}

R_3: Trusted={Yes} $\wedge$ Weekend={No} $\rightarrow$ Permission={Allowed}

According to Definition 3, this rule set $\Re$ is *incomplete* because there is no rule for case ([Trusted={Yes} $\wedge$ Weekend={Yes }]).

3 Overview of Data Classification Algorithms

The primary objective of data classification algorithms is to organize and categorize data in distinct classes. Many data classification algorithms e.g., ID3 / C4.5 use information gain to construct decision trees. It is an indicator for deciding the relevance of an attribute. In general, the attribute that provides the highest information gain will appear first in the decision tree.

The process of decision tree creation starts with a single node representing all data [41]. If all cases in a data set belong to the same category then the node becomes a leaf labeled with a category label. Otherwise, an algorithm will select an attribute according to the following criteria [21]:

1. For each attribute a, find the normalized information gain from splitting on a.
2. Let a_{best} be the attribute with the highest normalized information gain.
3. Create a decision node that splits on a_{best}.
4. Recur on the sublists obtained by splitting on a_{best}, and add the newly created nodes as children of the current node.

Let us assume that a data set S contains two classes P and N. Then, the information gain for an attribute A is calculated as follow [41]:

$$\text{gain}(A) = I(S_P, S_N) - E(A) \tag{1}$$

Here, $I(S_P, S_N)$ represents the amount of information needed to decide if an arbitrary example in S belongs

Table 1: Sample Policy Set

Subject	Action	Object	Location	Permission
Alice	Read	File 1	Building 1	Denied
Alice	Read	File 1	Building 2	Denied
Bob	Read	File 1	Building 1	Allowed
Carol	Write	File 1	Building 1	Allowed
Carol	Delete	File 2	Building 1	Allowed
Carol	Delete	File 2	Building 2	Denied
Bob	Delete	File 2	Building 2	Allowed
Alice	Write	File 1	Building 1	Denied
Alice	Delete	File 2	Building 1	Allowed
Carol	Write	File 2	Building 1	Allowed
Alice	Write	File 2	Building 2	Allowed
Bob	Write	File 1	Building 2	Allowed
Bob	Read	File 2	Building 1	Allowed
Carol	Write	File 1	Building 2	Denied

to P or N and $E(A)$ represents the information needed to classify objects in all subtrees. $I(S_P, S_N)$ is defined as [41]:

$$I(S_P, S_N) = -\frac{x}{x+y}\log_2\frac{x}{x+y} - \frac{y}{x+y}\log_2\frac{y}{x+y} \qquad (2)$$

where x is the number of elements in class P and y is the number of elements in class N. Let us assume that using attribute A as the root in the tree partitions S in sets $\{S_1, S_2, ...S_v\}$. If S_i contains x_i examples of P and y_i examples of N, then $E(a)$ is calculated as follows [41]:

$$E(a) = \sum_{i=1}^{v} \frac{x_i + y_i}{x+y} I(S_P, S_N) \qquad (3)$$

A sample data set is given in Table 1. This data set consist of four non-category attributes and class (category) attribute. Information gain for each attribute is given below.

- gain(Subject) = 0.246
- gain(Action) = 0.029
- gain(Object) = 0.151
- gain(Location) = 0.048

In this example, *Subject* attribute has the highest attribute. Therefore, it appears first in the decision tree as shown in the Figure 1. In this figure, the root node shows that out of 14 rules, 9 ($9/14 \approx 64\%$) belong to class *Allowed* and 5 ($5/14 \approx 35\%$) belong to class *Denied.*

4 Anomaly Detection Method

Intuitively, the decision trees that are generated by our method show the values of the category attributes for each combination of values of non-category attributes.

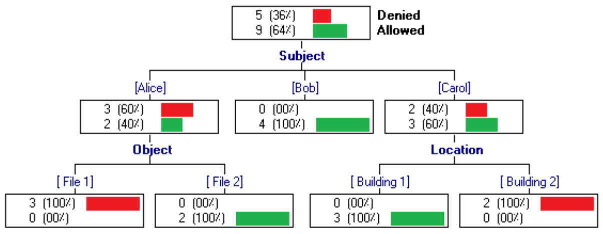


Fig. 1: Decision tree for the sample policy set

Each branch in the tree represents a set of values of a non-category attribute. Each path from root to leaf in a decision tree shows a possible combination of values for all attributes. Each node shows the probability of each category attribute being selected after the combination of attributes given in the path leading to it. Therefore, the root node, where no attribute values are known, will show Allowed and Denied each with 50% probability each. The leaf nodes, where all attribute values are known, should show only one of them with 100% probability, the other with 0% probability. However, if a leaf shows two category attributes with more than 0% probability, then the path leading to it identifies a combination of attribute values that leads to an inconsistency: two different decisions are possible in this case. If a leaf node shows that both category attributes have 0% probability, then the path leading to it identifies a case of incompleteness: no decision exists for this case. It can be easily seen that these definitions correspond to Definitions 1 and 2 above. The concept of probability is not needed for this application of the classification algorithm, although it could be useful in the case of access control methods with probabilistic outcomes.

Therefore, Definitions 1 and 2 can be implemented by generating decision trees as described above, and then checking the leaf nodes.

Our proposed anomaly detection method detects inconsistencies and incompleteness according to a given reference model and a policy set. The reference model is comprised of three things: the set of all possible non-category attributes that can be used to define rules, all possible values of each non-category attribute, and the set of category attributes. This information can be found by parsing the policy set or can be provided directly by the policy administrator. In the first case, we speak of a local reference model, and in the second case we speak of a global reference model. Clearly, for a given organization, any local reference model should be included in the relevant global reference model.

Table 2: Sample policy set

Subject	Resource	Action	Permission
Alice	File 1	Read	Allowed
Alice	File 1	Write	Denied
Alice	File 2	Read	Allowed
Alice	File 2	Write	Allowed
Alice	File 2	Write	Denied
Bob	File 2	Read	Denied
Bob	File 1	Read	Denied
Bob	File 1	Write	Allowed
Bob	File 2	Read	Denied

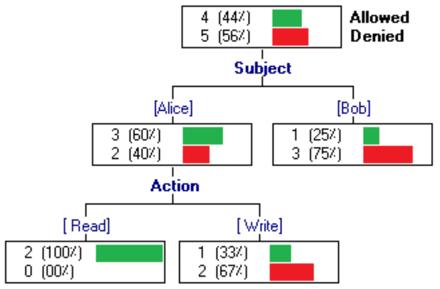


Fig. 2: Decision tree generated by the C4.5 algorithm

4.1 Constructing Decision Trees

As stated earlier, the attribute that provides the highest information gain will appear first in the decision tree. The attributes which provide low information gain may not necessarily appear in the decision tree. However for our analysis ignoring any attribute means loss of context and without complete context it is difficult to detect and diagnose inconsistencies and incompleteness. For example, if we apply the C4.5 algorithm on the policy data set given in the Table 2, we get the optimized decision tree[1] shown in Figure 2. Note that, in this figure, attribute “Resource” is not present. Due to this, it is not clear that Alice and Bob are both *Allowed* and *Denied* to perform *Read* and *Write* operations on which objects. In order to generate complete decision tree, a modification is needed in the decision tree construction process of the C4.5 algorithm

Algorithm 1 shows our proposed Discrete Decision Tree Generator (D-DTG) algorithm, that is used to generate complete decision trees when all non-category attributes are discrete / non-numeric. This algorithm is a simplified version of the standard ID3/C4.5 algorithm, in which we have changed the selection criteria of the attributes. Note that in this algorithm, we consider each attribute (Line 4) and all its possible values (Line 5) in the construction of a decision tree, whereas, in the ID3/C4.5 algorithm, attributes are selected based on the value of the information gain.

[1] For this purpose, we have used the Sipina data mining software package developed by Ricco Rakotomalala in the ERIC Research laboratory [30].

Algorithm 1 Discrete Decision Tree Generator (D-DTG) Algorithm

```
function D-DTG
Input: $A_d$: a set of non-category discrete attributes, $P$: the category attribute, $\Re$: a rule set
Output: a decision tree;
1: If $\Re$ is empty, return a single node with value Failure;
2: If $\Re$ consists of rules all with the same value for the category attribute $P$, return a single leaf node with that value;
3: If $A_d$ is empty, then
   Return a single node with the value of the most frequent of the values of the category attribute that are found in records of $\Re$;
4: Let $a$ be the first discrete attribute that is selected among attributes in $A_d$;
5: Let $\{a_j | j = 1, 2, .., m\}$ be the values of attribute $a$;
6: Let $\{R_j | j = 1, 2, .., m\}$ be the subsets of $\Re$ consisting respectively of records with value $a_j$ for $A_d$;
7: Return a tree with root labeled $a$ and arcs labeled $a_1, a_2, .., a_m$ going respectively to the trees (D-DTG($A_d/\{a\}, P, R_1$), D-DTG($A_d/\{a\}, P, R_2$), ... , D-DTG($A_d/\{a\}, P, R_m$));
8: Recursively apply D-DTG to subsets $R_j | j = 1, 2, .., m$ until they are empty;
```

The D-DTG algorithm takes three inputs: 1) a set of non-category discrete attributes A_d, 2) category attribute P, and 3) a policy set $\Re$. This algorithm first checks whether policy set $\Re$ contains some rules or not. If $\Re$ is empty, then a single node will be generated with error (Line 1). If it is not empty, then it will be checked whether all rules in $\Re$ belongs to the same category attribute-value (either Allowed or Denied). If all rules belong to the same category, then a leaf node will be created with that value (Line 2). After that the algorithm will check the non-category discrete attribute set A_d. If this is empty, then the algorithm will generate a single node with the value of the most frequent of the values of the category attribute that are found in records of $\Re$ (Line 3). If A_d is not empty, then the algorithm will select first the non-category attribute a (Lines 4) and all its possible values (Line 5) say $\{a_j | j = 1, 2, .., m\}$. After that all the rules in $\Re$, which contains records with value a_j are selected $\{R_j | j = 1, 2, .., m\} \subset \Re$ (Line 6). The D-DTG algorithm will return a tree with root labeled a and arcs labeled $a_1, a_2, .., a_m$ going respectively to the trees (D-DTG($A_D/\{a\}, P, R_1$), D-DTG($A_D/\{a\}, P, R_2$), . . . , D-DTG($A_d/\{a\}, P, R_m$)) (Line 7). This algorithm will be applied recursively to subsets $R_j | j = 1, 2, .., m$ until they are empty (Line 8).

Table 3: Sample password registration policy

User	Action	Contain alpha-numeric characters	Password length	Permission
*	Register	No	≤ 8	Reject
		No	≥ 9 and ≤ 12	Accept
		Yes	≤ 4	Reject
		Yes	≥ 5 and ≤ 8	Undefined
		Yes	≥ 9 and ≤ 12	Accept

When we apply our D-DTG algorithm on the policy set given in the Table 2, we get the complete tree shown in Figure 3. One can clearly see the difference between the trees generated by the standard C4.5 algorithm (Figure 2) and by the D-DTG algorithm (Figure 3). Note that all the attributes and their possible values, which are given as a reference model to the D-DTG algorithm, are present in the decision tree of Figure 3.

Algorithm 2 shows our proposed generic decision tree generator (DTG) algorithm, which is used to generate complete decision trees when there are both discrete and numeric non-category attributes. In order to handle numeric attributes, we have adopted a mechanism which is widely used in data classification algorithms. Essentially, the DTG algorithm reduces the continuous case to the discrete case by using threshold values and the fact that in a finite set of policies there will be only one finite set of such values. The numeric / continuous values are handled in the following manner.

Let us assume that the policy set $\Re$ contains a numeric attribute a. The possible values of the attribute a are first sorted into ascending order (Line 15), then the adjacent values that differ in their target classification are identified (Line 17). Commonly, in data classification algorithms, the midpoint $(a_j + a_{j+1})/2$ of each interval will act as a representative threshold value. We used the midpoint mechanism (Line 18). Sometimes, the smaller value a_i for each interval $\{(a_i, a_{i+1})\}$ could also be used as the threshold, rather than the mid point. Either of them can be used to compute the information gain to determine the optimal split [41,23] (Lines 18-26). The procedure for calculating the information gain is given in [41]. Note that data is partitioned into two sets based on the criteria $v(R_j.a) < \theta_k$ and $v(R_j.a) \geq \theta_k$ (Lines 21-25), for every numeric attribute a and every split point θ_k. So, the information gain is used here to generate the best split point, which will lead to the best optimal tree.

Let us assume that when a user registers her password, then a system will accept or reject that password

Algorithm 2 Decision Tree Generator (DTG) Algorithm

```
function DTG
Input: $A$: a set of non-category attributes, $P$: the category attribute, $\Re$: a rule set
Output: a decision tree;
1: If $\Re$ is empty, return a single node with value Failure;
2: If $\Re$ consists of rules all with the same value for the category attribute $P$, return a single leaf node with that value;
3: If $A$ is empty, then
   Return a single node with the value of the most frequent of the values of the category attribute that are found in records of $\Re$;
4: Let $A_d$ be the subset of $A$ consisting all discrete attributes.
5: Let $A_c$ be the subset of $A$ consisting all numeric/continuous attributes.
6: if $A_d$ is non-empty then
7:    Let $a$ be the first discrete attribute that is selected among attributes in $A_d$;
8:    Let $\{a_j | j = 1, 2, .., m\}$ be the values of attribute $a$;
9:    Let $\{R_j | j = 1, 2, .., m\}$ be the subsets of $\Re$ consisting respectively of records with value $a_j$ for $A_d$;
10:   Return a tree with root labeled $a$ and arcs labeled $a_1, a_2, .., a_m$ going respectively to the trees (DTG($A/\{a\}, P, R_1$), DTG($A/\{a\}, P, R_2$),..., DTG($A/\{a\}, P, R_m$));
11:   Recursively apply DTG to subsets $R_j | j = 1, 2, .., m$ until they are empty;
12: end if
13: if $A_c$ is non-empty then
14:   Let $a$ be the first continuous attribute that is selected among attributes in $A_c$;
15:   Let $\{a_j | j = 1, 2, .., m\}$ be the sorted values of attribute $a$;
16:   Let $\{R_j | j = 1, 2, .., m\}$ be the subset of $\Re$ consisting respectively of rules with value $a_j$ for $A_c$; The rules in $R_j$ are sorted on their values of attribute $a$.
17:   Let $I = \{(a_i, a_{i+1}), (a_j, a_{j+1}), ..., (a_m, a_{m+1})\}$ be the set of intervals, where each interval represents the adjacent values that differ in classification.
18:   Let $Z = \{\theta_j | j = 1, 2, .., m\}$ be the set of threshold values, where each threshold value represent the mid point of each interval present in $I$.
19:   Let $\theta_k$ is the threshold value in $Z$ with highest information gain.
20:   for $j = 1$ to $m$ do
21:      if $v(R_j.a) < \theta_k$ then
22:         Return a tree with root labeled $a$ and arc labeled $< \theta_k$ going to the tree DTG($A_c, P, R_j$);
23:      else
24:         Return a tree with root labeled $a$ and arc labeled $\geq \theta_k$ going to the tree DTG($A_c, P, R_j$);
25:      end if
26:   end for
27: end if
```

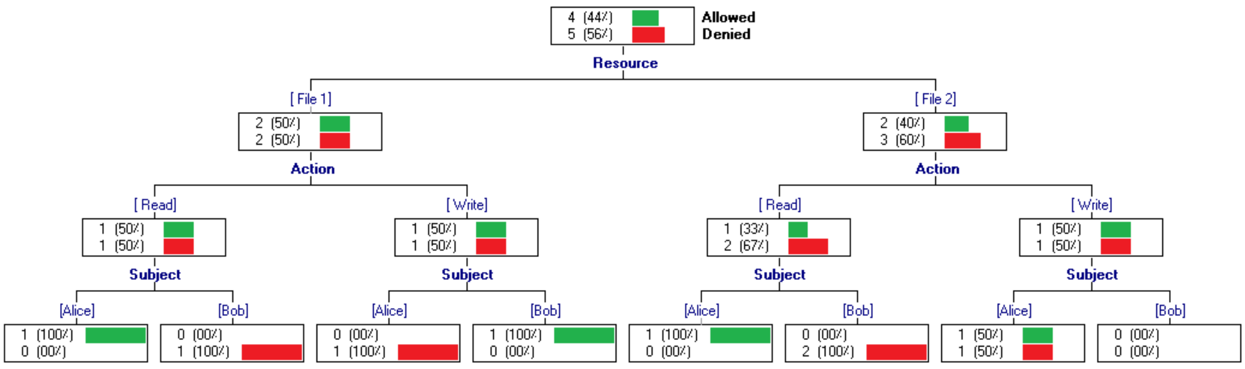


Fig. 3: Decision tree generated by the D-DTG algorithm

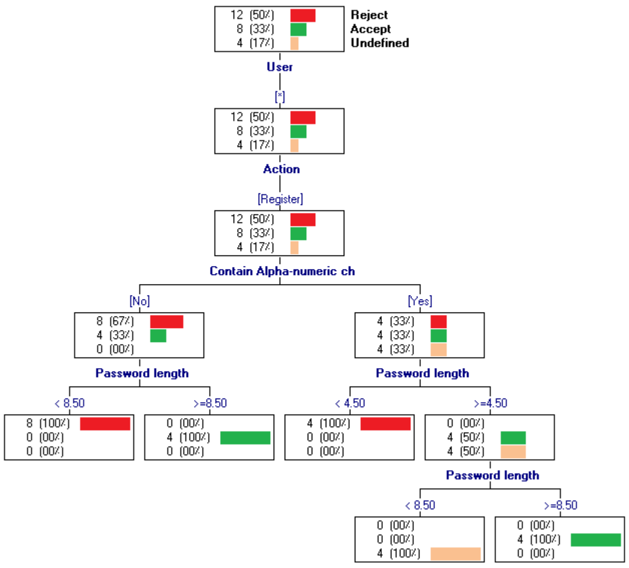


Fig. 4: Decision tree for sample password registration policy

according to the rules defined in Table 3. In this table, we have one numeric attribute, which represents the length of a password. When we apply our proposed DTG algorithm, we get the complete decision tree shown in Figure 4. In this figure, one can see that the mid points of all adjacent values, which differ in classification, are selected as threshold values.

Definition 3: A decision tree is said to be complete *iff*

$$T_{nodes} = \sum_{i=0}^{|A_d|} |\Upsilon(A_i)| + \sum_{i=0}^{|A_c|} |Z_i|$$

where $|A_d|$ represents the total number of non-category discrete attributes, $|A_c|$ represents the total number of non-category numeric attributes, T_{nodes} represents the total number of the leaf nodes, and $|Z_i|$ represents the total number of threshold values, where each threshold value represent the mid point of each interval as described in Line 18 of the Algorithm 2.

4.2 Executing the Anomaly Detection Algorithm

In a decision tree, each branch b_i (from the root to a leaf node) represents the complete context of one rule. In order to detect anomalies such as inconsistency and incompleteness, we apply Algorithm 3. First we check the leaf node of each branch. We have the following three main cases:

Case 1: Inconsistency. If any leaf node t_{node} contains more than one category (C) attribute value (Line: 5), this means that some rules in the policy set have the same non-category `attributes-values`, but different category `attribute-values`. Then according to **Definition 1**, this is a case of inconsistency. In order to determine which particular rules in the policy are mutually inconsistent, first we fetch all the attributes of the particular branch (Line: 6). The algorithm will then start searching the attribute-values in the actual policy set (Lines: 7-11). All the rules in the policy set that contain those `attribute-values` will be highlighted as inconsistent (Lines: 8-10).

Case 2: Incompleteness. If a leaf node t_{node} does not contain any category (C) attribute value (Line: 14), this means that no explicit rule is defined for the specific non-category attribute-values (Line:17). Then according to **Definition 2**, the given policy set is incomplete with respect to those attribute-values. The information about the complete context will be fetched from the complete branch (root to leaf node) (Line: 15).

Case 3: No anomaly. If each leaf node contains exactly one category attribute-value, then the policy set is complete and consistent (Lines: 21-26) according to the given reference model.

The algorithm can be generalized to the case where more than two category attributes exist in the reference model. In this case, we consider classes of equivalent category attributes, instead of a single category attribute.

Algorithm 3 Anomaly Detection Algorithm

Input: Decision tree
Output: Context of inconsistency and incompleteness

```
1: Let $A(b_i)$ be the set of all attributes present in one branch.
2: Bool $consistent = true$;
3: Bool $complete = true$;
4: for each branch $b_i$ in Decision tree do
5:     if more than 1 category attribute-value is assigned to leaf node $b_i.t_{node}$ then
6:         $A(b_i)$ = fetch_all_attributes_of_branch($b_i$);
7:         for each actual rule $R_a$ in the policy set do
8:             if $v(A(R_a)) = v(A(b_i))$ then
9:                 Highlight: $Ra : A_1 \wedge \ldots \wedge A_n \rightarrow C$;
10:            end if
11:        end for
12:        $consistent = false$;
13:    end if
14:    if no category attribute is assigned to leaf node $b_i.t_{node}$ then
15:        $A(b_i)$ = fetch_all_attributes_of_branch($b_i$);
16:        Policy set is incomplete w.r.to $label(b_i.t_{node})$;
17:        Complete context: $A(b_i)$;
18:        $complete = false$;
19:    end if
20: end for
21: if $consistent = true$ then
22:     No inconsistency found;
23: end if
24: if $complete = true$ then
25:     No incompleteness found;
26: end if
```

Let us take the decision tree shown in Figure 3. Note that the right-most leaf node contains no category attribute value. So, according to the Algorithm 3 and Definition 2, this is a case of incompleteness. By parsing the whole branch, we get the complete context of incompleteness, which gives the information that Bob is neither *Allowed* nor *Denied* to perform *Write* operation on *File 2*. Also, two category attribute-values (*Allowed* and *Denied*) exist at the 2^{nd} leaf node from the right. So, according to Algorithm 3 and Definition 1, this is a case of inconsistency, which states that Alice is simultaneously *Allowed* and *Denied* to perform *Write* operation on *File 2*.

Decision trees are also useful in detecting redundancy in access control policy sets. Note that each path from the root of a decision tree to one of its leaves can be transformed into a rule simply by the conjunction of the attributes that exists in that path [40,33]. So, in order to detect redundancy we only need to check whether a leaf node contains more than one rule with the same category value-attribute. If yes, this means that there exist redundant rules in the policy set. For example, if we examine the leaf nodes of the decision tree of Figure 3, we can find that the third leaf node from the right contains two rules with the same category attribute value (*denied*). This shows that the decision tree created from the policy set of Table 2 contains two rules one of which can be removed as redundant.

4.3 Anomaly Resolution

Our method does not provide assistance for anomaly resolution. For this, we refer to proposals published in the literature [4,16,6,27,10]. However, after the rule set is updated for resolution, our method can be applied again if anomalies were introduced.

Once detected, inconsistency can be manually addressed by removing rules or narrowing their range of applicability. For example, a rule may have an unintended wide scope that covers cases that should be handled by other rules. Incompleteness can be addressed by adding rules or enlarging the range of applicability of existing rules.

Automatic or even semi-automatic resolution of incompleteness is not easy. Our algorithm 3 will highlight the incompleteness instances. Those will be presented to the policy administrator. She will manually add rules for the highlighted instances based on the overall organizations policy.

In general terms, we note that our method provides clear identification of the existing anomalies, and thus will facilitate the resolution effort.

5 Normalization of input data

Normalizing and formatting input data is the first phase of our method. In order to generate valid decision trees, we may need to perform the following three manipulations on the policy data set.

1. Ordering of attributes,
2. Normalization of Boolean expressions, and
3. Interval manipulation.

They will be described in the following three subsections.

5.1 Ordering

As mentioned, rules are ordered collections of attribute values. For our method, in all rules, all attribute values

Table 4: Output of the transformation of the XML file of Figure 5

Role	Resource	Action	Permission
Doctor	Patient File	Read	Allowed
Doctor	Patient File	Write	Allowed
Nurse	Patient File	Read	Allowed
Nurse	Patient File	Write	Denied

must be ordered according to the same ordering of attributes. It does not matter what actually the order is. In our example, all rules define role firstly, object secondly, action thirdly, and permission lastly. This may not be the case if the policy set is imported from a policy database in another form, such as XML. In this second case, a parser is required to fetch the attribute values from the XML file and place them in uniformly ordered form.

Note that the objective of this step is to convert policies into tabular form. Any order of attributes can be used in a table since the entropy heuristics of the algorithm C4.5 will reorder them attempting to minimize the size of the resulting trees.

Let us assume that the XML file contains rules shown in Figure 5[2]. This XML file is given as an input to the parser. Then the parser will generate the output shown in Table 4. This tabular structure of the policy data set is a valid input for a data classification algorithm to generate a decision tree.

From this table we obtain the information about which attributes are present in the rule set and what are their values. For example, from Table 4, we obtain the following information:

A = {Role, Resource, Action}
Permission={Allowed, Denied}
Υ(Role) = {Doctor, Nurse}
Υ (Object)={Patient File}
Υ(Action)={Read, Write}

Based on the above-mentioned information, we create a local reference model which is used to see whether all cases are covered in the decision tree. Note that it is also possible that some valid attribute-value pairs are not present in the given rule set. For example, *Admin* may be an additional user, or *Payment File* may be an additional object. The policy administrator can provide such additional attribute-value pairs, to yield a global reference model. As mentioned, the local reference model should be included in the global reference model. Based on the given reference model, whether local or global, a decision tree is created.

5.2 Normalization of Boolean Expressions

Policy Administrators sometimes specify contextual conditions such as location and time on subjects, objects and actions. For example, subject can access object during working hours only; Objects can only be accessed from office; Action write can only be performed during specified time intervals. Permission is granted when all such contextual conditions are satisfied. In access control rules, these contextual conditions are defined in the form of Boolean expression. Arbitrary Boolean expressions can be handled by our method, after a process of expression transformation and rule splitting. We note that a decision tree which is generated by a data classification algorithm naturally encodes a Disjunctive Normal Form (DNF) Boolean formula [38]. Therefore, first, the Boolean expression is transformed into its disjunctive normal form ($B_1 \vee B_2 \vee \ldots \vee B_k$), and then the rule R_i is split into k rules.

$$R_{i.1} : A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge B_1 \rightarrow Permission$$
$$\vdots$$
$$R_{i.k} : A_1 \wedge A_2 \wedge \cdots \wedge A_n \wedge B_k \rightarrow Permission$$

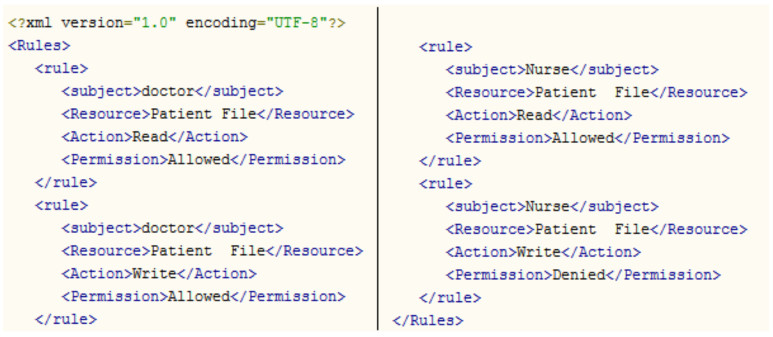
Let us assume that we have a rule which states that the subject *Alice* is permitted to perform action *Read* on object *Database* if the following Boolean expression is satisfied.

R_1: IF Subject={Alice} AND Action={Read} AND Object={Database} AND [((Project={P1} OR Project={P2}) AND Experience={$>$ 2 yr}) OR Role={Admin}] THEN Permission={Allowed}

Transformation of this Boolean expression into DNF leads to three alternative rules, shown in Table 5. These three rules can be represented in tabular form as shown in Table 6. Data classification algorithms can then be applied on the contents of this table.

It is also possible that a rule may contain negation. In that case, the NOT operator should be resolved first. This can be done in many ways. It is up to the implementer to select the best approach. In this work, we have adopted the following simple approach. If an attribute contains k unique values then $k-1$ rules will be generated to delete the negation operator. Let us assume that Subject = {Alice, Bob, Eve} and the rule says: If Subject is not Alice then Deny. In order to delete the NOT operator from this rule, two rules will be generated: 1) If the subject is Bob then Deny, 2) If the

[2] XAMCL, a standard policy specification language, was defined for this purpose. However, for ease of understanding and simplicity we have specified rules in simple XML language.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<Rules>
   <rule>
      <subject>doctor</subject>
      <Resource>Patient File</Resource>
      <Action>Read</Action>
      <Permission>Allowed</Permission>
   </rule>
   <rule>
      <subject>doctor</subject>
      <Resource>Patient  File</Resource>
      <Action>Write</Action>
      <Permission>Allowed</Permission>
   </rule>
   <rule>
      <subject>Nurse</subject>
      <Resource>Patient  File</Resource>
      <Action>Read</Action>
      <Permission>Allowed</Permission>
   </rule>
   <rule>
      <subject>Nurse</subject>
      <Resource>Patient  File</Resource>
      <Action>Write</Action>
      <Permission>Denied</Permission>
   </rule>
</Rules>
```

Fig. 5: Samples rules written in XML form

Table 5: Transformation of Boolean expression

R1.1:	IF Subject={Alice} AND Action={Read} AND Object={Database} AND Role={Admin} THEN Permission={Allowed}
R1.2:	IF Subject={Alice} AND Action={Read} AND Object={Database} AND Project={P1} AND Experience={>2 yr} THEN Permission={Allowed}
R1.3:	IF Subject={Alice} AND Action={Read} AND Object={Database} AND Project={P2} AND Experience={>2 yr} THEN Permission={Allowed}

Table 6: Rules in Tabular form

Subject	Action	Object	Project	Experience	Role	Permission
Alice	Read	Database	-	-	Admin	Allowed
Alice	Read	Database	P1	> 2 yr	-	Allowed
Alice	Read	Database	P2	> 2 yr	-	Allowed

subject is Eve then Deny. In this approach, the number of rules depends on the size of the domain. In section 6.2, we have presented the effects of the normalization and negation elimination approach at computation time as well as the resulting growth in the number of rules.

5.3 Interval finding method

Policy administrators sometimes specify attribute values as intervals. If all defined intervals are non-overlapping then each interval can be treated as a unique value. However, if the intervals are overlapping then we need to explore all possible unique intervals. This is necessary to detect any potential inconsistencies, which may exist at the overlapping intervals.

For example, policy 1 allows user x to access resource y if time is between 9 and 12, and policy 2 denies the same user access to the resource if time is between 11 and 13. Note that in the two policies, time intervals are overlapping. Before detecting any potential inconsistencies, first we need to create non-overlapping intervals. This can be done in many ways. One possible solution is described in Algorithm 4.

Algorithm 4 Non-overlapping Interval finding Algorithm

```
function IntervalFinder
Input: Boundary values
Output: List of Intervals;
1: Let $b = \{b_1, b_2, b_3, ...b_k\}$ is set containing boundary values in ascending order;
2: Let $I$ represent a set of non-overlapping intervals;
3: for  $i$=1 to $k-1$ do
4:     $I_i = [b_i, b_{i+1}) = \{x \mid b_i \leq x < b_{i+1}\}$
5: end for
6: return I;
```

In this algorithm, first we take all boundary values (Line 1), which in this example are: 9, 12, 11, and 13. After that we sort these values in ascending order: 9, 11, 12, and 13. Now we create non-overlapping intervals (Lines 3-5) in the following manner:

1. I_1: [9, 11)
2. I_2: [11, 12)

3. I_3: [12, 13)

After that the policies can be rewritten as follows:

- Policy 1 allows user x to access resource y in time interval I_1.
- Policy 1 allows user x to access resource y in time interval I_2.
- Policy 2 denies user x to access resource y in time interval I_2.
- Policy 2 denies user x to access resource y in time interval I_3.

This algorithm works well only for bounded (minimum and maximum) continuous values. For unbounded continuous attributes, this algorithm will not be able to identify intervals.

6 Analysis And Evaluation

6.1 Complexity Analysis

In the proposed method, complexity is mainly dependent on three factors: 1) Ordering of attributes 2) Transformation of Boolean expression into DNF and 3) Creation of decision trees.

1. Ordering of attributes can be done with the help of any efficient sorting algorithm algorithms, such as Timsort, Mergesort, and Heapsort. The computational complexity of these algorithms is $O(x \times \log\ x)$. Here x represents the number of items. Let us assume that a rule contains m attributes. In order to sort these attributes in a given order, the complexity will be $O(m \times \log\ m)$ which is linearithmic. If we have n rules, then the total complexity will be $O(n \times m \times \log\ m)$. Note that, in practice, $n > m$.
2. Any Boolean expression can be transformed into DNF by using logical transformation techniques, such as the double negation elimination, De Morgan's laws, and the distributive law. In some cases conversion to DNF requires exponential time $O(2^k)$, where k is the length of the Boolean expression. Let us assume that the policy set contains n rules and rule i contains a Boolean expression of length k_i then the total complexity will be $\sum_{i=0}^{n} O(2^{k_i})$. However, in practice, the length of Boolean expressions in an access control rule is usually small (e.g. k=4).
3. The ordered complexity of the C4.5 algorithm is $O(mn\ \log\ n)+O(n\ (\log\ n)^2)$ [39]. Here, $O(mn\ \log\ n)$ represent the complexity for building complete decision tree and $O(n\ (\log\ n)^2)$ is required for subtree raising (pruning). In our method, we primarily focus on building the complete decision tree. Therefore, the complexity for creating decision tree is $O(mn \log\ n)$.

The total complexity of the proposed method is:

$$\overbrace{O(n \times m \times \log\ m)}^{\text{for ordering}} + \overbrace{\sum_{i=0}^{n} O(2^{k_i})}^{\text{for normalization}} + \overbrace{O(m \times n \times \log\ n)}^{\text{for tree generation}} \tag{4}$$

6.2 Implementation and Evaluation

In order to evaluate the efficiency and effectiveness of the proposed solution, first we need large policy data sets. Unfortunately, no benchmarks have been published in this area and real industrial data are impossible to obtain because of confidentiality constraints. For this purpose, we have developed a simple *Random Policy Generator Tool* in Java. A typical screenshot of the tool is shown in Figure 6.

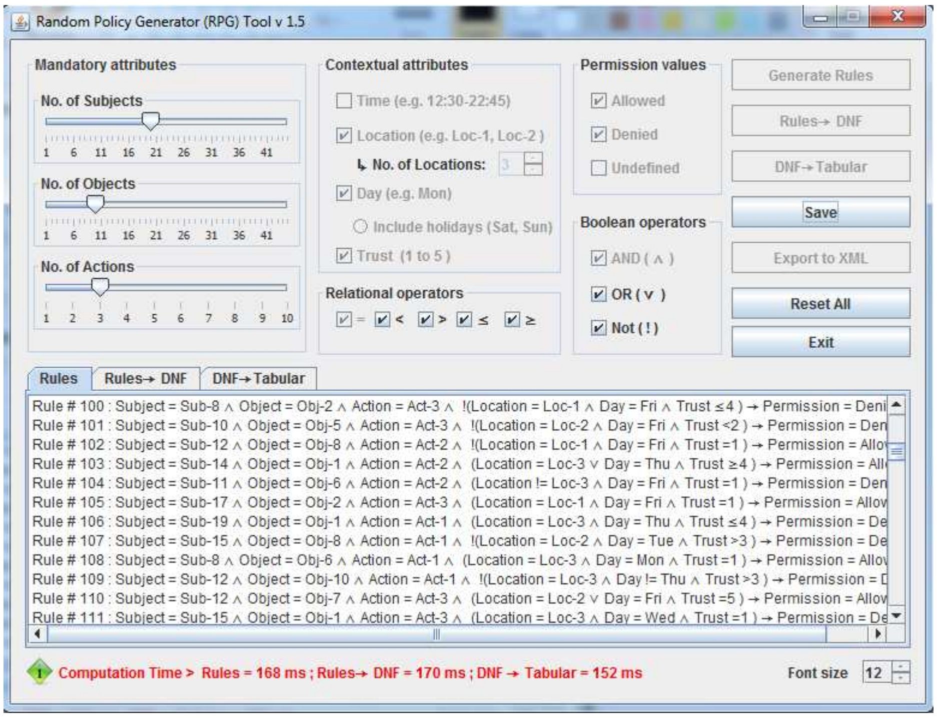
With the help of this tool, a user can generate the rules in Boolean expression format. In our conditional expressions, whole Boolean subexpressions can be negated. A sample Boolean formula that is generated by the tool is shown below.

Subject = Sub-7 $\wedge$ Object = Obj-2 $\wedge$ Action = Act-4 $\wedge$!(Location = Loc-1 $\wedge$ Day = Mon $\wedge$ Trust $\geq$ 4) $\rightarrow$ Permission = Allowed

In this tool, a user can specify any Boolean operators (AND, OR, NOT) and relational operators ($>, <, \leq, \geq$). Also, a user can define the domain size for the attributes like Subject, Object, Action, Day, Time, Location, Trust, and Permission. By using this tool, a user can transform rules into DNF and tabular form. Note that the size of the rules generated by the effect of the transformation depends on the size of the domain. This tool will also display the computation time for each processing stage. With the help of this tool, we have generated 10 data sets. The details about each data set are shown in Table 7.

In order to generate decision trees and detect anomalies in these data sets, we have also implemented a policy validation tool called *PVTool* in Java. It takes policies written in the tabular format. Based on the proposed solution, it first generates decision trees and then highlights the three types of anomalies (inconsistencies, incompleteness and redundancies) that are present in the given policy sets. A sample screenshot of the PVTool is shown in Figure 7.

We have evaluated the efficiency and effectiveness of the PVTool for policy analysis on policies that are generated by the RPG tool. Our experiments were performed on a Intel Core i5-2400 CPU 3.10 GHz with 4 GB RAM running on Windows 7 SP1. Table 8 shows



☐ Time (e.g. 12:30-22:45)
☑ Location (e.g. Loc-1, Loc-2)
No. of Locations: 3
☑ Day (e.g. Mon)
○ Include holidays (Sat, Sun)
☑ Trust (1 to 5)

☑ Allowed
☑ Denied
☐ Undefined

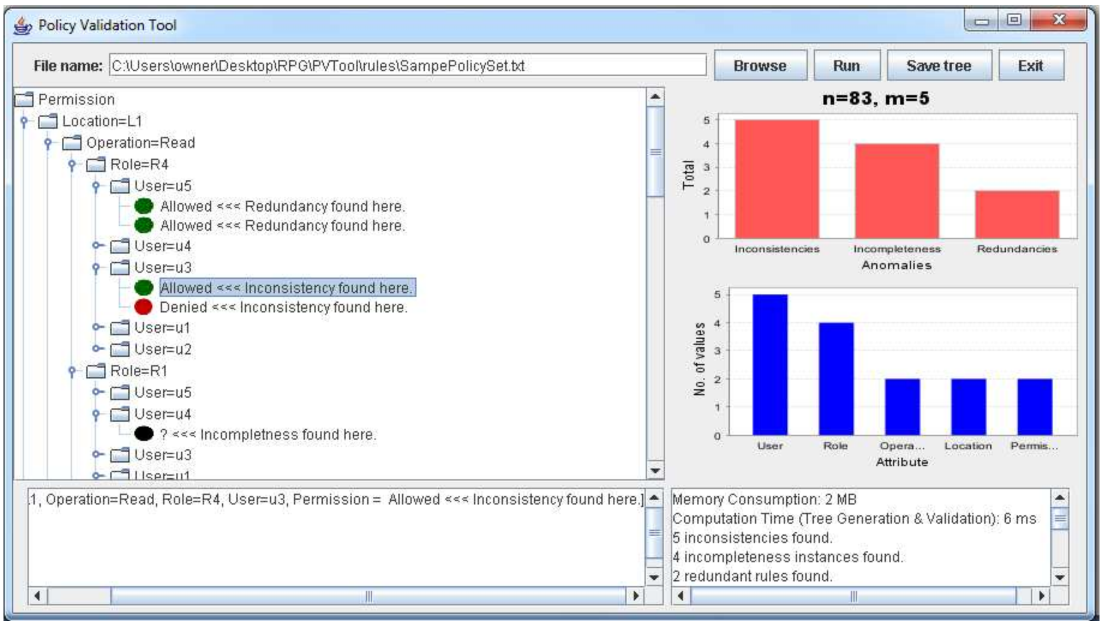
☑ AND (∧)
☑ OR (∨)
☑ Not (!)

☑ = ☑ < ☑ > ☑ ≤ ☑ ≥

```
Rule # 100 : Subject = Sub-8 ∧ Object = Obj-2 ∧ Action = Act-3 ∧  !(Location = Loc-1 ∧ Day = Fri ∧ Trust ≤4 ) → Permission = Deni
Rule # 101 : Subject = Sub-10 ∧ Object = Obj-5 ∧ Action = Act-3 ∧  !(Location = Loc-2 ∧ Day = Fri ∧ Trust <2 ) → Permission = Den
Rule # 102 : Subject = Sub-12 ∧ Object = Obj-8 ∧ Action = Act-2 ∧  !(Location = Loc-1 ∧ Day = Fri ∧ Trust =1 ) → Permission = Allo
Rule # 103 : Subject = Sub-14 ∧ Object = Obj-1 ∧ Action = Act-2 ∧  (Location = Loc-3 ∨ Day = Thu ∧ Trust ≥4 ) → Permission = All
Rule # 104 : Subject = Sub-11 ∧ Object = Obj-6 ∧ Action = Act-2 ∧  (Location != Loc-3 ∧ Day = Fri ∧ Trust =1 ) → Permission = Den
Rule # 105 : Subject = Sub-17 ∧ Object = Obj-2 ∧ Action = Act-3 ∧  (Location = Loc-1 ∧ Day = Fri ∧ Trust =1 ) → Permission = Allo
Rule # 106 : Subject = Sub-19 ∧ Object = Obj-1 ∧ Action = Act-1 ∧  (Location = Loc-3 ∧ Day = Thu ∧ Trust ≤4 ) → Permission = De
Rule # 107 : Subject = Sub-15 ∧ Object = Obj-8 ∧ Action = Act-1 ∧  !(Location = Loc-2 ∧ Day = Tue ∧ Trust >3 ) → Permission = De
Rule # 108 : Subject = Sub-8 ∧ Object = Obj-6 ∧ Action = Act-1 ∧  (Location = Loc-3 ∧ Day = Mon ∧ Trust =1 ) → Permission = Allo
Rule # 109 : Subject = Sub-12 ∧ Object = Obj-10 ∧ Action = Act-1 ∧  !(Location = Loc-3 ∧ Day != Thu ∧ Trust >3 ) → Permission = D
Rule # 110 : Subject = Sub-12 ∧ Object = Obj-7 ∧ Action = Act-3 ∧  (Location = Loc-2 ∨ Day = Fri ∧ Trust =5 ) → Permission = Allo
Rule # 111 : Subject = Sub-15 ∧ Object = Obj-1 ∧ Action = Act-3 ∧  (Location = Loc-3 ∧ Day = Wed ∧ Trust =1 ) → Permission = De
```

Computation Time > Rules = 168 ms ; Rules→ DNF = 170 ms ; DNF → Tabular = 152 ms

Fig. 6: Random Policy Generator Tool's Screenshot



File name: C:\Users\owner\Desktop\RPG\PVTool\rules\SampePolicySet.txt

```
Permission
  Location=L1
    Operation=Read
      Role=R4
        User=u5
          Allowed <<< Redundancy found here.
          Allowed <<< Redundancy found here.
        User=u4
        User=u3
          Allowed <<< Inconsistency found here.
          Denied <<< Inconsistency found here.
        User=u1
        User=u2
      Role=R1
        User=u5
        User=u4
          ? <<< Incompletness found here.
        User=u3
        User=u1
```

```
1, Operation=Read, Role=R4, User=u3, Permission =  Allowed <<< Inconsistency found here.]
```

```
Memory Consumption: 2 MB
Computation Time (Tree Generation & Validation): 6 ms
5 inconsistencies found.
4 incompleteness instances found.
2 redundant rules found.
```

Fig. 7: A screenshot of the PVTool

the memory consumption and computation time. Results show that the memory that is required to store the decision trees is not much. For example, 51 MB are needed for the largest policy set which contains 18471 DNF rules. Furthermore, the computation time trend also verifies the effectiveness of the proposed solution.

For scalability, it is also important to note that creating a single decision tree for a large policy data set is an inefficient approach. It is better to divide the policies according to the objects they refer to, since policies that refer to different objects cannot be in conflict. For example, if the policy set contains reference to 10 objects then 10 rule subsets are created, and, for each subset

Table 7: Information about data sets

Data Set	|Boolean formulas|	|Rules in DNF|	|Subject|	|Action|	|Object|	|Trust|	|Location|	|Days|	|Permission|
1	200	1655	10	4	5	5	3	5	2
2	400	3354	10	4	10	5	3	5	2
3	600	5658	15	4	10	5	3	5	2
4	800	6932	20	4	10	5	3	5	2
5	1000	8934	20	5	10	5	3	5	2
6	1200	11145	20	5	12	5	3	5	2
7	1400	12502	20	5	14	5	3	5	2
8	1600	14772	20	5	16	5	3	5	2
9	1800	15815	20	5	18	5	3	5	2
10	2000	18471	20	5	20	5	3	5	2

Table 8: Time and Memory Consumption Analysis

Data Set no.	Computation Time (ms)				Memory (MB)
	Rules→DNF	DNF→Tabular	Validation	Total	
1	57	41	49	147	7
2	119	62	97	278	12
3	198	81	123	402	17
4	266	94	150	510	21
5	345	120	228	693	26
6	299	200	268	767	31
7	305	192	291	788	36
8	362	243	372	977	41
9	371	236	406	1013	46
10	458	302	475	1235	51

a decision tree is created. In this way, smaller decision trees can be generated, that are easy to process and store.

6.3 Case Study

In this section, we demonstrate the use of our method for sets of policies in a policy language allowing positive and negative permissions, roles and conditional expressions. Several access control models using policy languages of this type have been proposed in the literature. The main inspiration for our case study has been the Organizational-based Access Control (OrBAC) model [18]. However the same method can be extended to other models, such as RB-RBAC [2] and ABAC-XACML [22]. We refer to [2] for a discussion of policy conflicts in the presence of roles and role hierarchies. Assume that an organization contains five users, four roles, and four objects as defined below.

Users = $\{u_1, u_2, u_3, u_4, u_5\}$
Roles = $\{R_1, R_2, R_3, R_4\}$
Resources = $\{obj_1, obj_2, obj_3, obj_4\}$

Users are assigned to roles and permissions (*Allowed* / *Denied*) are assigned to roles in various contextual or non-contextual situations. Contextual conditions are defined in the form of Boolean expressions. Contextual attributes contain qualitative or quantitative (numeric) values. Assume that the policy administrator has defined three contextual attributes: 1) Geo-spatial (e.g. Location), 2) Temporal (e.g. Time) and 3) Customized (e.g Risk value). The *Location* attribute contains two possible values: L_1 and L_2 while the *Time* attribute contains two possible intervals: T_1 and T_2. The time intervals are non-overlapping in this case study, therefore the time attribute will be considered here as a discrete attribute. The third customized contextual attribute is *Risk* whose range is 1 to 7. With this configuration, the policy administrator has defined the following policies.

1. Users belonging to role R_1 are *Denied* to perform *Read* and *Write* operations on the resource obj_2.
2. Users belonging to role R_1 are *Denied* to perform *Write* operations on the resource obj_3.
3. Users belonging to role R_2 are *Allowed* to perform *Read* and *Write* operations on the resource obj_2.
4. Users belonging to role R_2 are *Denied* to perform *Write* operations on the resource obj_3.
5. Users belonging to role R_3 are *Allowed* to perform *Read* and *Write* operations on the resource obj$_3$.
6. Users belonging to role R_4 are *Allowed* to perform *Read* operations on the resource obj_2.

Also, the policy administrator has added the following three policies containing Boolean expressions for accessing resource obj$_4$.

7. Users belonging to role R_4 are *Allowed* to perform *Read* and *Write* operations on the resource obj_4 if $[(L_1 \text{ AND } T_1) \text{ OR } (L_2 \text{ AND } T_2)]$.
8. Users belonging to role R_3 are *Denied* to perform *Write* operations on resource obj_4 if $[(L_2 \text{ AND } (T_1 \text{ OR } T_2)) \text{ OR } (L_1 \text{ AND } T_1)]$.

For access to resource obj$_1$, the policy administrator has added one condition of risk assessment as follows.

9. Users belonging to role R_1 are *Allowed* to perform *Read* and *Write* operations on the resource obj_1 if the risk value ($R(v)$) is between 1 and 4.

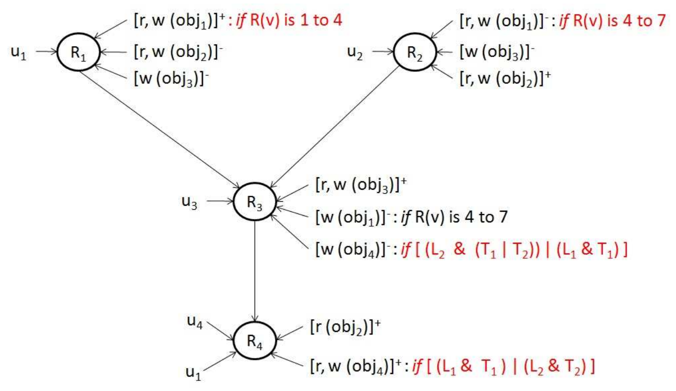

u1
R1
[r, w (obj1)]+ : if R(v) is 1 to 4
[r, w (obj2)]-
[w (obj3)]-
u2
R2
[r, w (obj1)]- : if R(v) is 4 to 7
[w (obj3)]-
[r, w (obj2)]+
u3
R3
[r, w (obj3)]+
[w (obj1)]- : if R(v) is 4 to 7
[w (obj4)]- : if [(L2 & (T1 | T2)) | (L1 & T1)]
u4
u1
R4
[r (obj2)]+
[r, w (obj4)]+ : if [(L1 & T1) | (L2 & T2)]

Fig. 8: Sample Scenario

10. Users belonging to role R_2 are *Denied* to perform *Read* and *Write* operations on the resource obj_1 if the risk value ($R(v)$) is between 4 and 7.
11. Users belonging to role R_3 are *Denied* to perform *Write* operations on the resource obj_1 if the risk value ($R(v)$) is between 4 and 7.

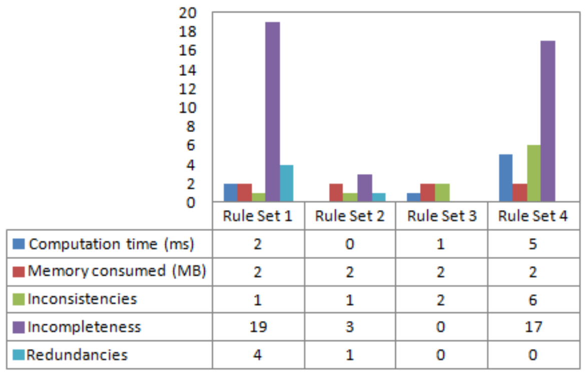
In some situations, users can be assigned to two or more roles. Because of this, users may be simultaneously permitted and prohibited to access given resources, leading to conflicts. Detecting conflicts in such situations becomes more challenging when the concept of role hierarchy is introduced in the access control model. Assume that the policy administrator has defined a role hierarchy / ordering in the following fashion:

$$R_1 \geqslant R_3 \geqslant R_4$$
$$R_2 \geqslant R_3 \geqslant R_4$$

This ordering shows that R_1 and R_2 inherit all the permissions that are assigned to role R_3 and role R_3 inherits all the permissions that are assigned to role R_4. Figure 8 shows the user assignment to role, role hierarchy and permission inheritance. Because of hierarchy, each user will get additional positive or negative authorizations as shown in Table 9.

In order to detect anomalies, we first organize access control rules in tabular form. After this we separate the access control rules (See Tables 10, 11, 12, 13) according to the resources they refer to, since the rules that refer to different resources cannot be in conflict. In this example, rules are divided into four rule sets. Note that rule set number four (Table 13) is generated after transformation of Boolean expressions in DNF form. Also, during analysis of rule set one (Table 10) we set the interval value 1 for the *Risk* attribute.

Our objective here is to find the anomalies in these rule sets that are caused by the role hierarchy. There-



	Rule Set 1	Rule Set 2	Rule Set 3	Rule Set 4
🟦 Computation time (ms)	2	0	1	5
🟥 Memory consumed (MB)	2	2	2	2
🟩 Inconsistencies	1	1	2	6
🟪 Incompleteness	19	3	0	17
🟦 Redundancies	4	1	0	0

Fig. 9: Analysis of OrBAC case study

Table 10: Rule set for obj_1

User	Role	Action	Risk	Permission
u_3	R_3	Write	[4,7]	Denied
u_1	R_1	Read	[1,4]	Allowed
u_1	R_1	Write	[1,4]	Allowed
u_1	R_3	Write	[4,7]	Denied
u_2	R_2	Read	[4,7]	Denied
u_2	R_2	Write	[4,7]	Denied
u_2	R_3	Write	[4,7]	Denied

Table 11: Rule set for obj_2

User	Role	Action	Permission
u_5	R_4	Read	Allowed
u_4	R_4	Read	Allowed
u_3	R_4	Read	Allowed
u_1	R_1	Read	Denied
u_1	R_1	Write	Denied
u_1	R_3	Read	Allowed
u_2	R_2	Read	Allowed
u_2	R_2	Write	Allowed
u_2	R_3	Read	Allowed

fore, we will not include the Role attribute in the decision tree. We analysed all four rule sets using our *PVTool* and the results are shown in Figure 9. This figure shows that the role hierarchy introduce inconsistencies, incompleteness and redundancies in the policy sets. Note that, dividing the rules in four sets reduces memory and processing time requirements. Decision trees for all four rule sets are given in the Appendix where precise information about the anomalies can be found.

7 Related Work and Comparison

The problem of possible inconsistency in access control policies was clearly stated in a paper by Lupu and Sloman in 1999 [25]. Completeness was mentioned in 2005 in [9]. Since then, considerable work has been

Table 9: Example: User Assignment and Authorization

Users	Role Assignment	Authorized roles	Assigned Privileges	Authorized Privileges
u_1	R_1	R_1, R_3, R_4	$[r,w(obj_1)]^{+}\text{if}\, R(v)=[1,4]$, $[r,w(obj_2)]^{-}$, $[w(obj_3)]^{-}$	$[r,w(obj_1)]^{+} if\, R(v)=[1,4]$, $[r,w(obj_2)]^{-}$, $[w(obj_3)]^{-}$, $[r,w(obj_3)]^{+}$, $[w(obj_1)]^{-}\text{if}\, R(v)=[4,7]$, $[w(obj_4)]^{-}\text{if}\, [(L_2\&(T_1\|T_2))\|(L_1\&T_1)]$, $[r,w(obj_4)]^{+}\text{if}\, [(L_1\&T_1)\|(L_2\&T_2)]$, $[r(obj_2)]^{+}$
u_2	R_2	R_2, R_3, R_4	$[r,w(obj_1)]^{-}\text{if}\, R(v)=[4,7]$, $[w(obj_3)]^{-}$, $[r,w(obj_2)]^{+}$	$[r,w(obj_1)]^{-}\text{if}\, R(v)=[4,7]$, $[w(obj_3)]^{-}$, $[r,w(obj_2)]^{+}$, $[r,w(obj_3)]^{+}$, $[w(obj_1)]^{-}\text{if}\, R(v)=[4,7]$, $[w(obj_4)]^{-}\text{if}\, [(L_2\&(T_1\|T_2))\|(L_1\&T_1)]$, $[r,w(obj_4)]^{+}\text{if}\, [(L_1\&T_1)\|(L_2\&T_2)]$, $[r(obj_2)]^{+}$
u_3	R_3	R_3, R_4	$[r,w(obj_3)]^{+}$, $[w(obj_1)]^{-}\text{if}\, R(v)=[4,7]$, $[w(obj_4)]^{-}\text{if}\, [(L_2\&(T_1\|T_2))\|(L_1\&T_1)]$	$[r,w(obj_3)]^{+}$, $[w(obj_1)]^{-}\text{if}\, R(v)=[4,7]$, $[w(obj_4)]^{-}\text{if}\, [(L_2\&(T_1\|T_2))\|(L_1\&T_1)]$, $[r,w(obj_4)]^{+}\text{if}\, [(L_1\&T_1)\|(L_2\&T_2)]$, $[r(obj_2)]^{+}$
u_4 & u_5	R_4	R_4	$[r,w(obj_4)]^{+}\text{if}\, [(L_1\&T_1)\|(L_2\&T_2)]$, $[r(obj_2)]^{+}$	$[r,w(obj_4)]^{+}\text{if}\, [(L_1\&T_1)\|(L_2\&T_2)]$, $[r(obj_2)]^{+}$

Here & and | symbols represent AND and OR logical operators respectively.

Table 12: Rule set for obj$_3$

User	Role	Action	Permission
u_3	R_3	Read	Allowed
u_3	R_3	Write	Allowed
u_1	R_1	Write	Denied
u_1	R_3	Read	Allowed
u_1	R_3	Write	Allowed
u_2	R_2	Write	Denied
u_2	R_3	Read	Allowed
u_2	R_3	Write	Allowed

Table 13: Rule set for obj$_4$

User	Role	Action	Location	Time	Permission
u5	R4	Read	L1	T1	Allowed
u5	R4	Read	L2	T2	Allowed
u5	R4	Write	L1	T1	Allowed
u5	R4	Write	L2	T2	Allowed
u4	R4	Read	L1	T1	Allowed
u4	R4	Read	L2	T2	Allowed
u4	R4	Write	L1	T1	Allowed
u4	R4	Write	L2	T2	Allowed
u3	R3	Write	L1	T1	Denied
u3	R3	Write	L2	T1	Denied
u3	R3	Write	L2	T2	Denied
u3	R4	Read	L1	T1	Allowed
u3	R4	Read	L2	T2	Allowed
u3	R4	Write	L1	T1	Allowed
u3	R4	Write	L2	T2	Allowed
u1	R3	Write	L1	T1	Denied
u1	R3	Write	L2	T1	Denied
u1	R3	Write	L2	T2	Denied
u1	R4	Read	L1	T1	Allowed
u1	R4	Read	L2	T2	Allowed
u1	R4	Write	L1	T1	Allowed
u1	R4	Write	L2	T2	Allowed
u2	R3	Write	L1	T1	Denied
u2	R3	Write	L2	T1	Denied
u2	R3	Write	L2	T2	Denied
u2	R4	Read	L1	T1	Allowed
u2	R4	Read	L2	T2	Allowed
u2	R4	Write	L1	T1	Allowed
u2	R4	Write	L2	T2	Allowed

done, with different aims and results, which are often not comparable to ours. Some authors have dealt with the issue of synthesizing or constructing valid policy sets; others have dealt with the issue of composing policy sets, for example in the context of transactions and collaborative systems; others yet have dealt with the issue of adding policies to existing policy sets, for example in the context of administrative systems. All these problems are different with the problem discussed in this paper, which deals with checking policy sets that already exist and may have accumulated defects over time. As mentioned, one of the practical applications of our technique is in security auditing.

Researchers have used various approaches such as Boolean satisfaction (SAT) algorithms, modeling (e.g. UML), and decision trees (e.g. MTBDDs) for policy validation. Each approach has its own advantages and disadvantages that are highlighted below.

Some researchers [14,26] have proposed techniques based on satisfaction algorithms or the Alloy toolset [17] for policy validation. For example, Hu *et al.* [14] have proposed an approach for verifying formal specifications of a role-based access control model and corresponding policies with selected security properties by means of satisfaction algorithms of the Alloy logic checker. Similarly, Mankai *et al.* [26] have proposed a method that translates sets of policies written in XACML to the first order logic modeling language Alloy, to detect and visualize possible conflicts within sets of access control policies. However, the use of Alloy presents some limitations [12], of which the most important is the fact that the Alloy logic checker requires that signatures be bound to small values, and so the results may not be true in general. For the same reason, Alloy has severe limitations for expressing numeric values, which creates problems in case of conditions involving hours of the day or monetary amounts, among others.

Armando and Ranise [3] have developed a sophisticated logical theory to support consistency checking of policies with conditions on structured attribute domains by using Satisfiability Modulo Theories (SMT) logic solvers. The SMT logic solvers separate the Boolean part of satisfiability checking from algorithms used to check properties in specialized domains, such as the theory of real numbers or time indications. The authors provide performance statistics from test runs showing that their method is practically feasible. However it is not obvious that their method can be used by auditors or administrators without specific training and sophisticated interfaces. Their method does not deal with incompleteness.

Adi *et al.* [1] have proposed a type system to check the specified access control rules for consistency. The authors consider a fairly general model taking into account different access control specification properties including abstract rules, and positive and negative authorizations as well as context expressions. However, this work is limited to consistency checking.

Das *et al.* [8] have proposed a distributed system named *Baaz* for detecting misconfigurations in access control systems. The *Baaz* uses two techniques: 1) Group Mapping, and Object Clustering - for detecting misconfigurations in access permissions. However, authors have not considered continuous values, Boolean expressions and context.

Other methods for conflict detection, such as the one of [19] are based on theorem-proving principles that may be difficult to implement automatically for the general problem.

Fisler *et al.* [12] have developed a software suite called MARGRAVE for analyzing role-based access control policies. This tool analyzes policies written in XACML format and then translates them into multi-terminal binary decision diagrams (MTBDD) to verify security properties. This is a very efficient scheme, but its use with continuous values is not clear.

Gouda *et al.* [13] have proposed a very efficient structured firewall design method to prevent inconsistency, incompleteness, and compactness problems. The policy administrator needs to designs firewalls using firewall decision diagrams (FDD) instead of simply generating rules in sequence as it is usually done. This method eliminates the problem of inconsistency, which does not arise if a firewall is designed in this way. Also, incompleteness does not arise since the syntactic requirements of FDDs force policy administrators to consider all types of traffic. However, for large scale enterprises, it may be impossible for a policy administrator to design policies in terms of decision diagrams manually and to take care of all contexts to eliminate all possible anomalies.

Hu *et al.* [15] have proposed a novel anomaly management framework that can detect and resolve anomalies in firewall policies. In that framework, they adopted a rule-based segmentation mechanism and grid-based representation technique. The proposed framework provides two core functions: 1) conflict detection and resolution, and 2) redundancy discovery and removal. Hu *et al.* [16] have also proposed a policy-based segmentation technique to detect policy anomalies in XACML-based access control policies. The proposed technique adopts a binary decision diagram-based data structure to perform set operations, for policy anomaly detection and resolution. The proposed scheme is very efficient in detecting inconsistencies, and redundancies. It also handles Boolean expressions and continuous values. However, this method does not deal with incompleteness.

Lin et al. [24] present a comprehensive tool for access control policy analysis, called EXAM. This tool was designed to do analysis of relationships among policies, such as equivalence, refinement, redundancy, and conflict. It can produce an analysis of similarities and differences between policy sets. It handles continuous values, such as time intervals, but, according to its purpose, it does not do completeness analysis. Similar to Fisler et al. [12], the tool uses a variant of Binary Decision Diagrams, called Multi-Terminal Binary Decision Diagrams (MTBDD). The user of EXAM is provided with a flexible query language, which makes it possible to query sets of policies for many different properties. The use of EXAM for consistency checking is illustrated in Example 5 of the paper. Suppose that a patient, who has specified in a policy its privacy requirements, wishes to join a hospital, and suppose that the hospital has its own privacy policy. Then the tool enables the patient to query whether there are inconsistencies between her own policy and the hospital's policy. For example, a type of query will allow her to determine whether there are situations where her own policy would result in a denial, but the hospital's policy would result in a permission. The semantics of EXAM' query language is fairly complex and needs to be learned. In contrast, our tool is designed to efficiently find inconsistencies and incompleteness in a set of rules, which could be the union of several sets, without the need of specific queries. Therefore, the aim and structure of EXAM are quite different with the ones of our proposed method.

Some researchers [4,29] have used data mining techniques to solve the issue for policy validation in access control systems. For example, Bauer *et al.* [4] used association rule mining technique to predict potential misconfigurations or situations where no rule is defined. However, in that paper, authors have not considered complex Boolean expressions or numeric attributes. Also, Mukkamala *et al.* [29] have proposed a data mining method for detecting and correcting misconfigurations for RBAC systems. However, this work is limited to identification and correction of over-privileged and under-privileged roles.

Some researchers [31,36] have used the Unified Modeling Language (UML) and the Object Constraint Language (OCL) for the specification and validation of authorization constraints for the RBAC model. In this approach, authors have specified reusable RBAC policies using UML diagram templates. With these templates, application specific RBAC policies are generated. Further, RBAC Constraints can be specified by using OCL that is based on first order logic. However, the major limitation of this approach is that it can only specify static structures and check current snap shots of RBAC configurations.

Stepien *et al.* [37] have developed a method for detecting inconsistency by using Constraint Logic Programming with an encoding of rule sets in the Prolog programming language. This method is efficient but does not deal with incompleteness.

7.1 Qualitative Comparison and Discussion

A high level qualitative comparison of the proposed method with eight existing methods is presented in Table 14. The following five parameters are used for the comparison.

1. Inconsistency detection: It indicates whether a method can detect inconsistencies in access control policies or not.
2. Incompleteness detection: It indicates whether a method can detect incompleteness in access control policies or not.
3. Flexibility: It indicates whether a method can detect anomalies either at the design time or at a runtime (Note our ability to detect inconsistency after delegation). If a method detects anomalies at the design time only then we say that the method is inflexible.
4. Handling Boolean expressions: It indicates whether a method provides a mechanism for evaluating Boolean expressions that are commonly used for defining contextual conditions in access control policies.
5. Handling continuous values: It indicates whether a method provides a mechanism to handle continuous values (for example time), which are quite common in access control policies.

Table 14 highlights our contributions as compared to existing work. It shows that our work not only simultaneously detects inconsistencies and incompleteness in access control policies but also has ability to handle Boolean expressions and continuous values.

The purpose of our work is to provide a method that will perform efficiently for policies that can be expressed mostly as conjunctions of positive terms, or which can be reduced to this form easily. In practice, such policy sets are common. Complex Boolean expressions are not necessary in many cases and may be beyond the capabilities of the average security administrator. For policy sets of this type, more complex methods will not be as efficient as ours.

How to compare the efficiency of our method to the efficiency of others? We believe that millisecond counts are not very useful, since they are dependent on practical factors such as programming language used, OS overhead, type of computer etc. It is nearly impossible to reproduce the same environment for a meaningful comparison. For this reason, we have provided analytic estimates, and we note that other papers in the area do not provide them, probably because the techniques they propose are too complex to be evaluated analytically. We hope that our example will be followed by other authors, and this will lead to greater clarity in this research area.

8 Conclusions

Existing policy validation schemes have some shortcomings, chiefly the inability to deal efficiently with continuous values, Boolean expressions or detection of incom-

Table 14: Qualitative Comparison

	Technique	Inconsistency detection	Incompleteness detection	Flexibility	Handling Boolean expressions	Handling continuous values
Fisler *et al.* [12]	Multi-terminal binary decision diagrams	Yes	No	Yes	Yes	No
Lin et al. [24]	Multi-terminal binary decision diagrams	Yes	No	Yes	Yes	Yes
Mankai *et al.* [26]	Satisfaction algorithms	Yes	No	Yes	Yes	No
Hu *et al.* [14]	Satisfaction algorithms	Yes	No	Yes	Yes	No
Armando & Ranise [3]	Satisfiability Modulo Theories	Yes	No	No	Yes	Yes
Kamoda *et al.* [19]	Free Variable Tableaux	Yes	No	No	Yes	No
Hu *et al.* [16]	policy-based segmentation technique	Yes	No	Yes	Yes	Yes
Gouda *et al.* [13]	Decision diagrams	Yes	Yes	No	No	No
Bauer *et al.* [4]	Rule mining	Yes	Yes	Yes	No	No
Das *et al.* [8]	Group mapping and Object clustering techniques	Yes	Yes	Yes	No	No
Stepien *et al.* [37]	Constraint Logic Programming	Yes	No	No	Yes	Yes
Our proposed method	Data classification	Yes	Yes	Yes	Yes	Yes

pleteness situations. In order to overcome these shortcomings, we have proposed a new anomaly detection method that is based on data classification technique. It consists of the three main phases: 1) normalization and formatting of an input policy data; 2) decision tree construction based on the proposed DTG and D-DTG algorithms; and 3) Execution of the proposed anomaly detection algorithm, which detect inconsistencies and incompleteness in the access control policy sets. After the execution of our algorithm, then anomalies can be eliminated. To the best of our knowledge, we are the first to propose the use of data classification technique to detect anomalies in access control policies. Our proposal is generic, i.e. independent of any underlying policy specification language. Our experimental results prove the effectiveness of the proposed solution.

Security software is becoming increasingly complex. Complexity leads to human errors and human error may lead to breach of security, privacy and compliance, involving unauthorized access to sensitive data. Loss of assets and prestige are some of the possible consequences. Our solution will help prevent, detect and correct human errors and increases the capacity for handling larger and more complex access control policy sets. By adopting our solution, access management security products will become more robust and will gain useful diagnostics characteristics.

ACKNOWLEDGMENT

The work reported in this article was partially supported by the Natural Sciences and Engineering Research Council of Canada, PROMPT Quebec, and CA Technologies. We would like to thank Serge Mankovski of CA Technologies for having helped our effort. The authors would also like to thank all members of the Computer Security Research Lab (UQO,Canada), and Bernard Stepien for providing useful comments and suggestions.

References

1. Adi, K., Bouzida, Y., Hattak, I., Logrippo, L., Mankovskii, S.: Typing for conflict detection in access control policies. In: G. Babin, P. Kropf, M. Weiss (eds.) E-Technologies: Innovation in an Open World, *Lecture Notes in Business Information Processing*, vol. 26, pp. 212–226. Springer Berlin Heidelberg (2009)
2. Al-Kahtani, M.A., Sandhu, R.: Rule-based RBAC with negative authorization. In: 20th Annual Computer Security Applications Conference, pp. 405–415. IEEE (2004)
3. Armando, A., Ranise, S.: Automated and efficient analysis of role-based access control with attributes. In: N. Cuppens-Boulahia, F. Cuppens, J. Garcia-Alfaro (eds.) Data and Applications Security and Privacy XXVI, *Lecture Notes in Computer Science*, vol. 7371, pp. 25–40. Springer Berlin Heidelberg (2012)
4. Bauer, L., Garriss, S., Reiter, M.K.: Detecting and resolving policy misconfigurations in access-control systems. In: SACMAT '08: Proc. of the 13th ACM symposium on Access control models and technologies, pp. 185–194. ACM, New York, NY, USA (2008)

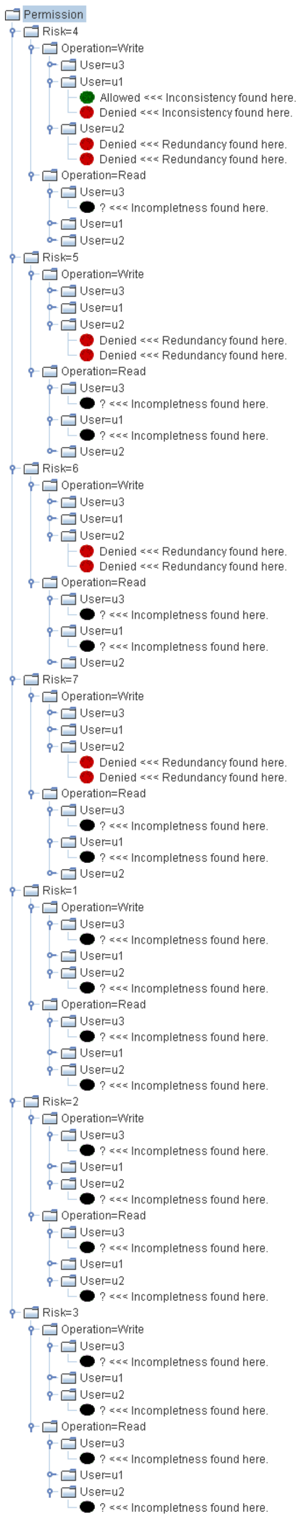
5. Benferhat, S., El Baida, R., Cuppens, F.: A stratification-based approach for handling conflicts in access control. In: SACMAT'03: Proc. of the eighth ACM symposium on Access control models and technologies, pp. 189–195. ACM, New York, NY, USA (2003)
6. Chinaei, A., Chinaei, H., Tompa, F.: A unified conflict resolution algorithm. In: W. Jonker, M. Petkovi (eds.) Secure Data Management, *Lecture Notes in Computer Science*, vol. 4721, pp. 1–17. Springer Berlin Heidelberg (2007)
7. Cuppens, F., Cuppens-Boulahia, N., Ghorbel, M.B.: High level conflict management strategies in advanced access control models. Electronic Notes in Theoretical Computer Science **186**, 3–26 (2007)
8. Das, T., Bhagwan, R., Naldurg, P.: Baaz: A system for detecting access control misconfigurations. In: Proc. of the 19th USENIX Security Symposium (USENIX) (2010)
9. De Capitani di Vimercati, S., Samarati, P., Jajodia, S.: Policies, models, and languages for access control. In: S. Bhalla (ed.) Databases in Networked Information Systems, *Lecture Notes in Computer Science*, vol. 3433, pp. 225–237. Springer Berlin Heidelberg (2005)
10. Dong, C., Russello, G., Dulay, N.: Flexible resolution of authorisation conflicts in distributed systems. In: F. De Turck, W. Kellerer, G. Kormentzas (eds.) Managing Large-Scale Service Deployment, *Lecture Notes in Computer Science*, vol. 5273, pp. 95–108. Springer Berlin Heidelberg (2008)
11. Dunlop, N., Indulska, J., Raymond, K.: Dynamic conflict detection in policy-based management systems. In: Proc. of the Sixth international Enterprisestributed object Computing Conference (EDOC'02), p. 15. IEEE Computer Society, Los Alamitos, CA, USA (2002)
12. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: ICSE '05: Proc. of the 27th international conference on Software engineering, pp. 196–205. ACM, New York, NY, USA (2005)
13. Gouda, M.G., Liu, A.X.: Structured firewall design. Computer Networks **51**(4), 1106–1120 (2007)
14. Hu, H., Ahn, G.: Enabling verification and conformance testing for access control model. In: SACMAT'08: Proc. of the 13th ACM symposium on Access control models and technologies, pp. 195–204. ACM, New York, NY, USA (2008)
15. Hu, H., Ahn, G.J., Kulkarni, K.: Detecting and resolving firewall policy anomalies. Dependable and Secure Computing, IEEE Transactions on **9**(3), 318–331 (2012). DOI 10.1109/TDSC.2012.20
16. Hu, H., Ahn, G.J., Kulkarni, K.: Discovery and resolution of anomalies in web access control policies. Dependable and Secure Computing, IEEE Transactions on **10**(6), 341–354 (2013). DOI 10.1109/TDSC.2013.18
17. Jackson, D.: Automating first-order relational logic. ACM SIGSOFT Software Engineering Notes **25**(6), 130–139 (2000)
18. Kalam, A.A.E., Baida, R.E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miége, A., Saurel, C., Trouessin, G.: Organization based access control. In: Proc. of the IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), pp. 120–131. IEEE Computer Society, Los Alamitos, CA, USA (2003)
19. Kamoda, H., Yamaoka, M., Matsuda, S., Broda, K., Sloman, M.: Access Control Policy Analysis Using Free Variable Tableaux. Information and Media Technologies **1**(2), 1155–1169 (2006)
20. Karp, A.H., Haury, H., Davis, M.H.: From ABAC to ZBAC: The Evolution of Access Control Models. Tech. report HPL-2009-30, HP Labs, `http://www.hpl.hp.com/techreports/2009/HPL-2009-30.pdf` (2009)
21. Kotsiantis, S.: Supervised machine learning: A review of classification techniques. Informatica **31**, 249–268 (2007)
22. Lang, B., Foster, I., Siebenlist, F., Ananthakrishnan, R., Freeman, T.: A flexible attribute based access control method for grid computing. Journal of Grid Computing **7**(2), 169–180 (2009)
23. Leung, K.M.: Decision trees and decision rules. `http://cis.poly.edu/~mleung/FRE7851/f07/decisionTrees.pdf` (2007). Accessed: 2014-01-07
24. Lin, D., Rao, P., Bertino, E., Li, N., Lobo, J.: EXAM: a comprehensive environment for the analysis of access control policies. International Journal of Information Security **9**(4), 253–273 (2010)
25. Lupu, E.C., Sloman, M.: Conflicts in policy-based distributed systems management. IEEE Transactions on Software Engineering **25**(6), 852–869 (1999)
26. Mankai, M., Logrippo, L.: Access control policies: Modeling and validation. In: Proceedings of NOTERE 2005, pp. 85–91 (2005)
27. Masoumzadeh, A., Amini, M., Jalili, R.: Conflict detection and resolution in context-aware authorization. In: Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on, vol. 1, pp. 505–511. IEEE (2007)
28. Moon, C.J., Paik, W., Kim, Y.G., Kwon, J.H.: The conflict detection between permission assignment constraints in role-based access control. In: D. Feng, D. Lin, M. Yung (eds.) Information Security and Cryptology, *Lecture Notes in Computer Science*, vol. 3822, pp. 265–278. Springer Berlin Heidelberg (2005)
29. Mukkamala, R., Kamisetty, V., Yedugani, P.: Detecting and resolving misconfigurations in role-based access control (short paper). In: A. Prakash, I. Sen Gupta (eds.) Information Systems Security, *Lecture Notes in Computer Science*, vol. 5905, pp. 318–325. Springer Berlin / Heidelberg (2009)
30. Rakotomalala, R.: Sipina data mining software. `http://eric.univ-lyon2.fr/~ricco/sipina.html` (2010)
31. Ray, I., Li, N., France, R., Kim, D.K.: Using UML to visualize role-based access control constraints. In: SACMAT '04: Proc. of the 9th ACM symposium on Access control models and technologies, pp. 115–124. ACM, New York, NY, USA (2004)
32. Rissanen, E.: eXtensible Access Control Markup Language (XACML) Version 3.0 OASIS Standard. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf` (2013). Accessed: 2014-01-07
33. Rokach, L., Maimon, O.: Decision trees. In: O. Maimon, L. Rokach (eds.) Data Mining and Knowledge Discovery Handbook, pp. 165–192. Springer US (2005)
34. Shaikh, R.A., Adi, K., Logrippo, L., Mankovski, S.: Detecting incompleteness in access control policies using data classification schemes. In: Proc. of the 5th International Conference on Digital Information Management (ICDIM 2010), pp. 417–422. IEEE (2010)
35. Shaikh, R.A., Adi, K., Logrippo, L., Mankovski, S.: Inconsistency detection method for access control policies. In: Proc. of the 6th International Conference on Information Assurance and Security (IAS 2010), pp. 204–209. IEEE (2010)
36. Sohr, K., Ahn, G.J., Gogolla, M., Migge, L.: Specification and validation of authorisation constraints using

UML and OCL. In: Computer Security (ESORICS 2005), LNCS 3679, pp. 64–79. Springer, New York, NY, USA (2005)

37. Stepien, B., Matwin, S., Felty, A.P.: Strategies for reducing risks of inconsistencies in access control policies. In: Proc. of the Fifth International Conference on Availability, Reliability and Security (AReS 2010), pp. 140–147 (2010)
38. Szörényi, B.: Disjoint DNF Tautologies with Conflict Bound Two. Journal on Satisfiability, Boolean Modeling and Computation **4**, 1–14 (2007)
39. Witten, I.H., Frank, E.: Data mining: Practical machine learning tools and techniques with java implementations. Morgan Kaufmann Publishers, USA (1999)
40. Yuan, Y., Shaw, M.J.: Induction of fuzzy decision trees. Fuzzy Sets and systems **69**(2), 125–139 (1995)
41. Zaiane, O.: Chapter 7: Data classification. `http://webdocs.cs.ualberta.ca/~zaiane/courses/cmput690/slides/Chapter7/index.htm` (1999). Accessed: 2014-01-07

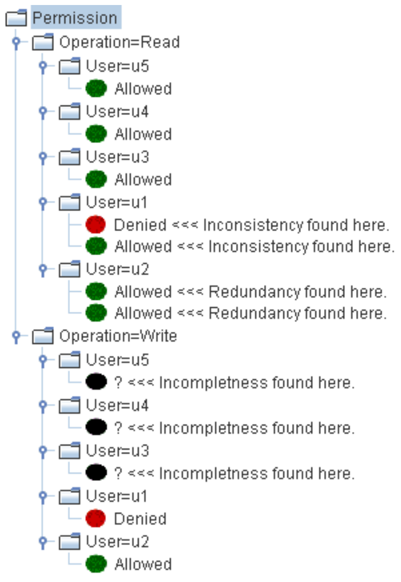
Appendix

Decision trees for the rule set 1, 2, 3 and 4 are given in Figure 10, 11, 12, and 13 respectively.



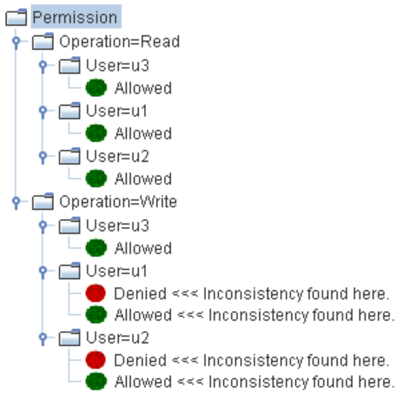
```
Permission
  Risk=4
    Operation=Write
      User=u3
      User=u1
        🟢 Allowed <<< Inconsistency found here.
        🔴 Denied <<< Inconsistency found here.
      User=u2
        🔴 Denied <<< Redundancy found here.
        🔴 Denied <<< Redundancy found here.
    Operation=Read
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
      User=u2
  Risk=5
    Operation=Write
      User=u3
      User=u1
      User=u2
        🔴 Denied <<< Redundancy found here.
        🔴 Denied <<< Redundancy found here.
    Operation=Read
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
        ⚫ ? <<< Incompletness found here.
      User=u2
  Risk=6
    Operation=Write
      User=u3
      User=u1
      User=u2
        🔴 Denied <<< Redundancy found here.
        🔴 Denied <<< Redundancy found here.
    Operation=Read
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
        ⚫ ? <<< Incompletness found here.
      User=u2
  Risk=7
    Operation=Write
      User=u3
      User=u1
      User=u2
        🔴 Denied <<< Redundancy found here.
        🔴 Denied <<< Redundancy found here.
    Operation=Read
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
        ⚫ ? <<< Incompletness found here.
      User=u2
  Risk=1
    Operation=Write
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
      User=u2
        ⚫ ? <<< Incompletness found here.
    Operation=Read
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
      User=u2
        ⚫ ? <<< Incompletness found here.
  Risk=2
    Operation=Write
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
      User=u2
        ⚫ ? <<< Incompletness found here.
    Operation=Read
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
      User=u2
        ⚫ ? <<< Incompletness found here.
  Risk=3
    Operation=Write
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
      User=u2
        ⚫ ? <<< Incompletness found here.
    Operation=Read
      User=u3
        ⚫ ? <<< Incompletness found here.
      User=u1
      User=u2
        ⚫ ? <<< Incompletness found here.
```

Fig. 10: Decision tree for rule set 1



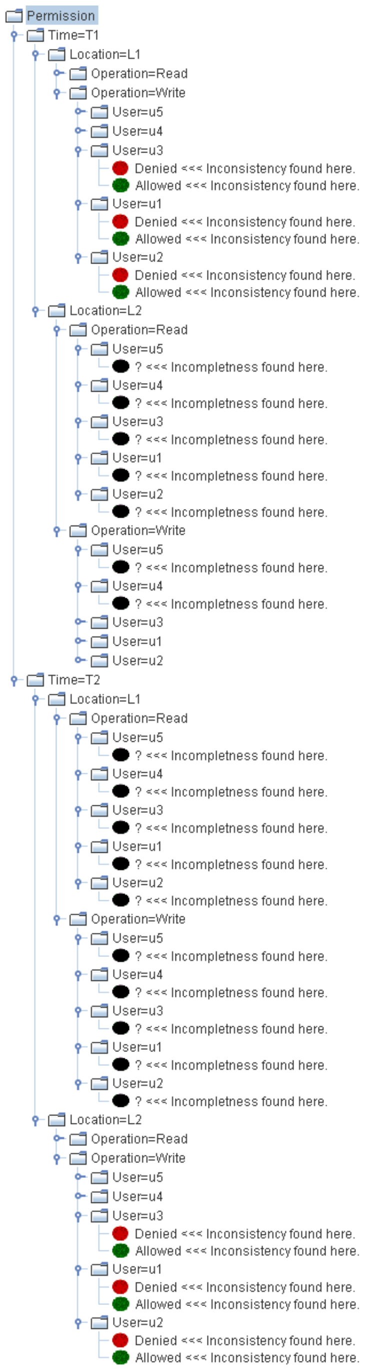
```
Permission
  Operation=Read
    User=u5
      Allowed
    User=u4
      Allowed
    User=u3
      Allowed
    User=u1
      Denied <<< Inconsistency found here.
      Allowed <<< Inconsistency found here.
    User=u2
      Allowed <<< Redundancy found here.
      Allowed <<< Redundancy found here.
  Operation=Write
    User=u5
      ? <<< Incompletness found here.
    User=u4
      ? <<< Incompletness found here.
    User=u3
      ? <<< Incompletness found here.
    User=u1
      Denied
    User=u2
      Allowed
```

Fig. 11: Decision tree for rule set 2



```
Permission
  Operation=Read
    User=u3
      Allowed
    User=u1
      Allowed
    User=u2
      Allowed
  Operation=Write
    User=u3
      Allowed
    User=u1
      Denied <<< Inconsistency found here.
      Allowed <<< Inconsistency found here.
    User=u2
      Denied <<< Inconsistency found here.
      Allowed <<< Inconsistency found here.
```

Fig. 12: Decision tree for rule set 3



```
Permission
  Time=T1
    Location=L1
      Operation=Read
      Operation=Write
        User=u5
        User=u4
        User=u3
          Denied <<< Inconsistency found here.
          Allowed <<< Inconsistency found here.
        User=u1
          Denied <<< Inconsistency found here.
          Allowed <<< Inconsistency found here.
        User=u2
          Denied <<< Inconsistency found here.
          Allowed <<< Inconsistency found here.
    Location=L2
      Operation=Read
        User=u5
          ? <<< Incompletness found here.
        User=u4
          ? <<< Incompletness found here.
        User=u3
          ? <<< Incompletness found here.
        User=u1
          ? <<< Incompletness found here.
        User=u2
          ? <<< Incompletness found here.
      Operation=Write
        User=u5
          ? <<< Incompletness found here.
        User=u4
          ? <<< Incompletness found here.
        User=u3
        User=u1
        User=u2
  Time=T2
    Location=L1
      Operation=Read
        User=u5
          ? <<< Incompletness found here.
        User=u4
          ? <<< Incompletness found here.
        User=u3
          ? <<< Incompletness found here.
        User=u1
          ? <<< Incompletness found here.
        User=u2
          ? <<< Incompletness found here.
      Operation=Write
        User=u5
          ? <<< Incompletness found here.
        User=u4
          ? <<< Incompletness found here.
        User=u3
          ? <<< Incompletness found here.
        User=u1
          ? <<< Incompletness found here.
        User=u2
          ? <<< Incompletness found here.
    Location=L2
      Operation=Read
      Operation=Write
        User=u5
        User=u4
        User=u3
          Denied <<< Inconsistency found here.
          Allowed <<< Inconsistency found here.
        User=u1
          Denied <<< Inconsistency found here.
          Allowed <<< Inconsistency found here.
        User=u2
          Denied <<< Inconsistency found here.
          Allowed <<< Inconsistency found here.
```

Fig. 13: Decision tree for rule set 4