Inconsistency Detection Method for Access Control Policies

Riaz Ahmed Shaikh, Kamel Adi, Luigi Logrippo
Department of Computer Science and Engineering
Université du Québec en Outaouais
Quebec, Canada
Email: {riaz.shaikh, kamel.adi, luigi.logrippo}@uqo.ca

Serge Mankovski
CA Technologies
125 Commerce Valley DR W, Thornhill
Ontario, Canada.
Email: serge.mankovski@ca.com

***Abstract*—In enterprise environments, the task of assigning access control rights to subjects for resources is not trivial. Because of their complexity, distribution and size, access control policies can contain anomalies such as inconsistencies, which can result in security vulnerabilities. A set of access control policies is inconsistent when, for specific situations different incompatible policies can apply. Many researchers have tried to address the problem of inconsistency using methods based on formal logic. However, this approach is difficult to implement and inefficient for large policy sets. Therefore, in this paper, we propose a simple, efficient and practical solution for detecting inconsistencies in access control policies with the help of a modified C4.5 data classification algorithm.**

***Index Terms*—Access control, Data classification, Decision tree, Inconsistency, Policy validation.**

I. Introduction

A set of access control policies is inconsistent when, for specific situations, different incompatible policies can apply. The problem of detecting inconsistencies has been studied intensively in the access control community [1], [2], [3], [4], [5], [6]. Most researchers have tried to solve this problem using methods based on formal logic. However, this approach suffers from exponential growth of computational complexity, as well as difficulties with continuous values (for example time), which are quite common in access control policies. Some researchers have tried to address the issue of inconsistency by adding special meta-rules in access control systems. For example, in firewalls the rules are executed top-down and only the first rule applicable is executed [7], later rules that may contradict it are ignored. In XACML, conflict resolution meta-rules (override) can be provided by the user: first override, deny override, etc. However, the results of the application of these meta-rules may not reflect the intention of the security administrator. Therefore, potential inconsistencies should be brought to the attention of the security administrator in an efficient autonomic manner.

In this paper, we adopt classification techniques widely used in data mining and in the machine learning community for detecting inconsistencies in sets of access of control policies. Several well known data classification algorithms exist, such as ID3 [8], C4.5 [9], ASSISTANT 86 [10] etc. The C4.5 algorithm is a state-of-the-art algorithm and one of the most widely used in the machine learning community. We have chosen to use the C4.5 algorithm in this work because of its ability to handle both *continuous* and *missing* attribute values. In order to detect the highest possible number of inconsistencies in the access control policies, we have made modifications in the standard C4.5 algorithm. We found that our proposed solution is very efficient in detecting inconsistencies in various complex scenarios. Also, our solution is simple and practical. To the best of our knowledge, we are the first to use a modified C4.5 data classification algorithm to detect inconsistencies in sets of access control policies.

The rest of the paper is organized as follows. Section II presents related work. Section III contains formal definitions of inconsistencies in access control policies. Section IV presents the decision tree construction process. Section V describes the proposed inconsistency detection method. Section VI shows how to detect inconsistencies in various scenarios using our proposed technique. Finally, Section VIII concludes the paper and discusses future work.

II. Related Work

Fisler *et al.* [11] 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. In our work, we use a data mining algorithm to generate decision trees. Decision trees are different from binary decision diagrams (BDD) in many ways. Some of them are: 1) In order to create BDDs, we need complete sets of policies; this is not mandatory in the case of decision trees. 2) Because of the first difference, BDDs do not provide efficient mechanisms to detect incompleteness in sets of policies. 3) For BDDs, the policies have to be encoded in terms of binary variables. No such encoding is necessary in our case. 4) The MTBDD technique does not provide a method for reducing the size of policy sets. A separate algorithm is necessary.

Gouda *et al.* [7] 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. In enterprises, policies can be generated and changed dynamically, therefore inconsistency and incompleteness should be detected as they are generated.

Some researchers [12], [13] have used formal verification techniques such as satisfaction algorithms and the Alloy toolset [14] for policy validation. For example, Hu *et al.* [12] 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 SAT and Alloy. Similarly, Mankai *et al.* [13] 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 set of access control policies. However, the use of Alloy presents some limitations [11], 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. In addition, Alloy has severe limitations for numeric values, which creates problems in case of conditions involving hours of the day or monetary amounts, among others.

III. Concepts and Definitions

In terms of data mining, 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 represents some important features of a particular rule and contains some discrete or continuous value. On the other hand, a category attribute represents the class of the rule to which it belongs. Typically, a category attribute takes only the values {*Allowed*, *Denied*}, perhaps *Not Applicable*, etc.

We have a *direct inconsistency* when two rules present in the same policy set lead to contradictory conclusions. For example, 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 in the same context. Formally, we can define inconsistency in following manner.

Let $\Re$ be a set of rules ($\Re = \{R_1, R_2, \ldots, R_n\}$), where $\Re \neq \phi$. All rules $R \in \Re$ have uniform structure, consisting of a number of attribute/value pairs. This can be realistically expected, if one assumes that default values can be used. Each rule $R \in \Re$ comprises a set of non-category attributes $A = \{A_1, A_2, \ldots, A_n\}$ and one category attribute C. Formally, a rule R_i can be written as follow:

$$R_i : A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow C$$

For example, consider the following rule.

$$R : role(Doctor) \wedge resource(Medicalrecord) \wedge$$
$$action(Write) \rightarrow Allowed$$

In this example, *Doctor*, *Medical record* and *Write* operation are the non-category attributes of the rule and *Allowed* is the category attribute of the rule.

Let $\upsilon(R_i.A_j)$ denote the value assigned to an attribute A_j in rule R_i.

Definition 1: Rules $R_i, R_j \in \Re$ are *mutually inconsistent* if and only if

1) $\forall A_k \in A \;\; \upsilon(R_i.A_k) = \upsilon(R_j.A_k)$ and
2) $\upsilon(R_i.C) \neq \upsilon(R_j.C)$

Informally, condition 1 in the above definition states that all decision-making attribute-values of R_i are same as the corresponding attribute-values of rule R_j and condition 2 states that the category attribute-value of rule R_i is different from the category attribute-value of rule R_j. Note that we assume that decision-making attributes will be in the same order for all rules, a condition that can be easily satisfied.

We have an *indirect inconsistency* if two rules present in different policy sets lead to contradictory conclusions. Such inconsistencies are difficult to see because they may be not visible at the time of defining policies and can be triggered only when some specific event occur. For example, Alice is allowed to create accounts and Bob is allowed to delete accounts. A policy may state that create and delete operation cannot be performed by the same entity. Inconsistency could occur if Alice delegates her rights to Bob. From the perspective of data mining, we can formally define inconsistency in following manner.

Definition 2: Rule $R_i \in \Re$ and $R_j \in \Re'$ are mutually inconsistent if and only if

1) $\forall A_k \in A \;\; \upsilon(R_i.A_k) = \upsilon(R_j.A_k)$ and
2) $\upsilon(R_i.C) \neq \upsilon(R_j.C)$

IV. Decision Tree Construction

In the standard C4.5 implementation, the process of decision tree creation starts with a single node representing all data [15]. If all cases in a data set belong to the same class then the node becomes a leaf labeled with a class label. Otherwise, an algorithm will select an attribute according to the following criteria [16].

---

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 those nodes as children of current node.*

TABLE I
SAMPLE RULES

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

Let us assume that a data set S contains two classes P and N. Then, the information gain for an attribute A is calculated in the following manner [15]:

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

Here, $I(S_P, S_N)$ represents the amount of information needed to decide if an arbitrary example in S belongs 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 [15]:

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

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 will partition 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 in following manner [15].

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

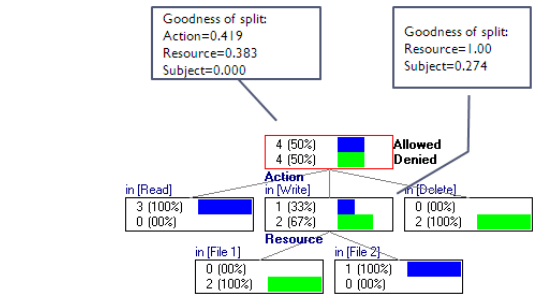
Let us assume that the policy administrator has defined the rules given in Table I. On this policy data set, by applying the standard C4.5 algorithm, we get the decision tree shown in Figure 1(a). In this figure, one can see that the attribute that provides the most information gain appears first in the decision tree. Consequently, we always get optimized or compact trees in which some attributes may not appear. However, we are interested in to detect anomalies such as inconsistencies in the policy data set. Therefore, all attributes must be present in the decision tree so that we can get a complete picture of the domain. Without having a complete decision tree, we may not able be to detect all inconsistencies in the policy set.

V. INCONSISTENCY DETECTION STRATEGY

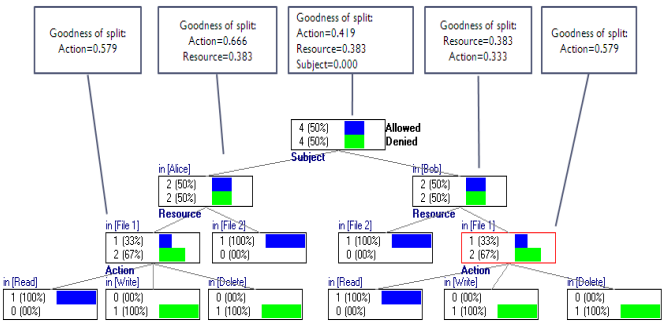
Our proposed inconsistency detection method consists of the following two steps.

A. *Step#1: Creation of complete tree*

In order to assure that all attributes must be present in the decision tree, we have made the following changes in the standard C4.5 algorithm.



(a) Standard C4.5



(b) Modified C4.5

Fig. 1. Decision trees

For each attribute that does not appear already in the decision tree

2) *Calculate the information gain that results from splitting on that attribute.*
3) *Split on the attribute that gives the lowest information gain.*

When we applied the modified C4.5 algorithm on the policy data set of Table I, we get the decision tree shown in Figure 1(b). In this figure, one can note that the attributes that give the lowest information gain appear on the top. Because of this we obtain a more complete tree as compared to the standard C4.5 algorithm. Note that the subject attribute of the policy does not appear in Figure 1(a). Having complete decision tree is more useful for detecting inconsistencies in access control policies as discussed in detailed in Section VI.

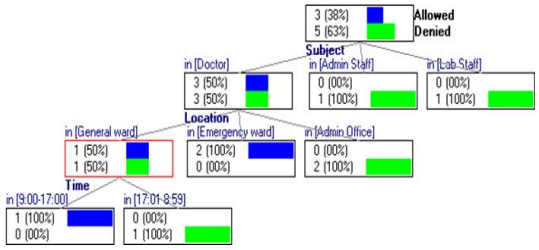
B. *Step#2: Inconsistency analysis*

In a decision tree, each branch b_i (from the root to a terminal node) represents one rule. In order to detect inconsistency, we will apply Algorithm 1. First we check the terminal node of each branch (Lines: 3-4). If any terminal node t_{node} contains more than one category (C) attribute value (Line: 4), this means that some rules in the policy set are mutually inconsistent. In order to determine which particular rules in the policy are mutually inconsistent, first we fetch all the attributes of the particular branch (Line: 5). After that the algorithm will start searching the attribute-values in the actual policy set (Lines: 6-10). All the rules in the policy set that contain those attribute-values will be highlighted as inconsistent (Lines: 7-9). If in a decision tree, no terminal node has more than one

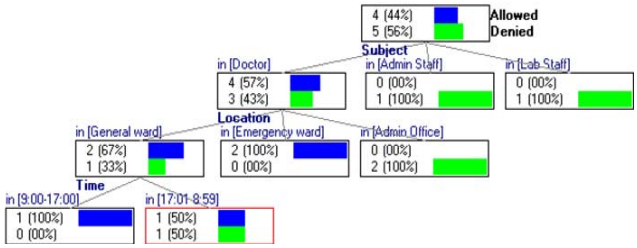
Algorithm 1 Inconsistency Detection Algorithm

Input: Decision tree
Output: Context of inconsistency

1: Let $A(b_i)$ be the set of all attributes present in one branch.
2: Bool $consistent = true$;
3: **for** each branch b_i in Decision tree **do**
4: **if** more than 1 category attribute is assigned to terminal node $b_i.t_{node}$ **then**
5: $A(b_i)$ = fetch_all_attributes_of_branch(b_i);
6: **for** each actual rule R_a in the policy set **do**
7: **if** $v(A(R_a)) = v(A(b_i))$ **then**
8: Highlight: $Ra : A_1 \wedge \ldots \wedge A_n \rightarrow C$;
9: **end if**
10: **end for**
11: $consistent = false$;
12: **end if**
13: **end for**
14: **if** $consistent = true$ **then**
15: No inconsistency found;
16: **end if**



(a) contains no inconsistency



(b) contains inconsistency

Fig. 2. Decision tree for medical record

category attribute-value then this means that no inconsistency has been found in the policy set (Lines: 14-16).

VI. Demonstration

In this section, we will discuss two scenarios. In the first scenario, we demonstrate how to detect inconsistencies in a single access control policy set. In the second scenario, we will show an example of inconsistency detection in two sets of access control policies.

A. Single Policy Set

Let us assume that we have one resource called medical records. A sample policy set that includes five rules is given in Table II.

TABLE II
Policy set for medical records

Role	Location	Time	Permission
Doctor	L1: General ward	T1: 9:00-17:00	Allowed
		T2: 17:01-8:59	Denied
	L2: Emergency ward	T1: 9:00-17:00	Allowed
		T2: 17:01-8:59	Allowed
	L3: Admin Office	T1: 9:00-17:00	Denied
		T2: 17:01-8:59	Denied
Lab staff	-	-	Denied
Admin staff	-	-	Denied

On this policy set, we have applied the modified C4.5 data mining algorithm to generate the decision tree. For this purpose, we have used the Sipina data mining software package developed by Ricco Rakotomalala in the ERIC Research laboratory [17].

The root node in Figure 2(a) shows that out of eight rules, three lead towards the *Allowed* decision and five lead towards the *Denied* decision. At the 2^{nd} level of a decision tree, we find that *Lab Staff* and *Admin Staff* are denied access to medical records. The remaining six rules are associated with *Doctor*. This node (*Doctor*) gives us the information that in three contexts *Doctors* are allowed access to medical records and in three contexts *Doctors* are denied access to medical records. One can see that each terminal node contains one non-category attribute value. This means that no direct inconsistency exists in this policy set. Let us now assume that the policy administrator adds the following new rule for resource "medical records".

$$R: \quad Subject(Doctor) \wedge Location(Generalward) \wedge Time(17:01-8:59) \rightarrow Allowed$$

In this case, we get the decision tree shown in Figure 2(b). We see that two contradictory permissions (allowed and denied) exist at a single terminal node (4th level, 2nd node from left). This shows that two rules are mutually inconsistent in the policy set. In order to point out which two rules conflict in the policy set, we apply Algorithm 1. First we fetch all the non-category attributes of the particular branch. In this case the attributes are: Subject (Doctor), Location(General ward) and Time (17:01-8:59). After that we start searching the policy set. All the rules that contain the above mentioned attributes will be highlighted. After reading the policy set of the medical records, the following two rules are found to be in conflict:

$$R_1: \quad Subject(Doctor) \wedge Location(Generalward) \wedge Time(17:01-8:59) \rightarrow Denied$$

$$R_2: \quad Subject(Doctor) \wedge Location(Generalward) \wedge Time(17:01-8:59) \rightarrow Allowed$$

TABLE III
POLICY SET FOR WORKING HOURS

Subject	Location	Time	Permission
Alice	General ward	9:00-12:00	Allowed
Alice	General ward	12:00-15:00	Denied
Alice	Emergency ward	9:00-12:00	Denied
Alice	Emergency ward	12:00-15:00	Allowed
Bob	-	-	Allowed

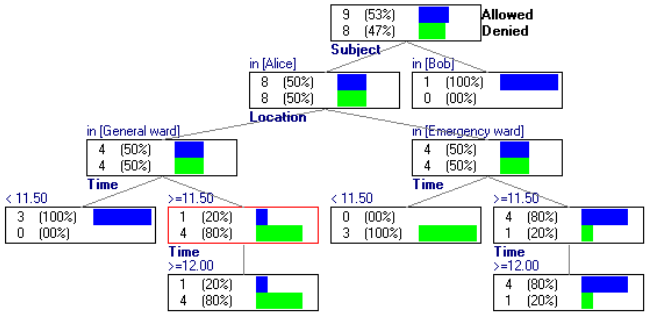


Fig. 3. Decision tree for continuous & overlapping attribute values: Time interval = 1.

So far we have only considered discrete and non-overlapping attribute values. Let us assume that time is a continuous attribute and its values are overlapping as shown in Table III. By applying the modified form of the C4.5 algorithm, we get the decision tree shown in Figure 3. We see that Alice is simultaneously allowed and denied to work in the general and emergency wards at time 12:00. Note that the C4.5 algorithm detected a split point in the time intervals which are placed at the time 11.50, i.e 11:30 in the usual time notation. This enabled the algorithm to detect two problems for Alice at time ≥ 12.

B. Multiple Policy Sets

In this scenario, we will focus on detecting inconsistencies between two different sets of policies. Indirect inconsistencies can be not visible at the time of defining policies, since they can occur after some specific event. For example, we have the following set of policies.

P_1: Administrator can *read*, *write* and *delete* database.
P_2: Technician can *read*, and *write* database.
P_3: Administrator can delegate his rights to technician.

So far, there are no conflicts in the above set of policies. A conflict can occur, if the policy administrator introduces the following requirement:

RQ_1: Action *delete* can only be performed by administrators.

If a delegation occurs one can see that the technician acquires the action *delete* through policy 3 which contradicts requirement 1.

Now we show how can one detect such conflicts with the help of data mining techniques. Let us assume that policy 1 and policy 2 are stored in the policy database shown in Table IV.

TABLE IV
AUTHORIZATION RULES

Role	Permission	Action
Admin	Allowed	Read
Admin	Allowed	Write
Admin	Allowed	Delete
Technician	Allowed	Read
Technician	Allowed	Write

TABLE V
CONSTRAINT ENFORCING RULES

Role	Permission	Action
Admin	Allowed	Delete
Technician	Denied	Delete

TABLE VI
NEW RULES AFTER DELEGATION

Role	Permission	Action
Technician	Allowed	Read
Technician	Allowed	Write
Technician	Allowed	Delete

According to requirement 1, only administrators are *allowed* to perform action *delete*. This implicitly means that other users who are not Administrators are *denied* action *delete*. These constraints should also be stored in the policy database in the form of rules as shown in Table V.

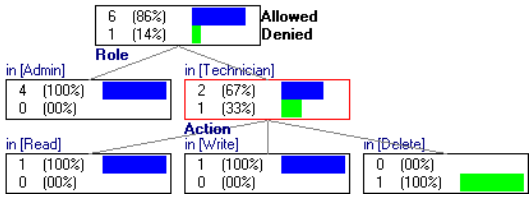
In order to find inconsistencies between authorization rules and constraint enforcing rules, we have applied our proposed strategy on the combined sets of policies of (Table IV and Table V). After applying the modified C4.5 algorithm, we get the following decision tree shown in Figure 4(a). One can see that all terminal nodes have only one action class attribute. This means that no inconsistency exists between authorization rules and constraints enforcing rules. Let us now assume that the administrator has performed the delegation operation. After performing delegation, the *Technician* is allowed to perform the following actions that are defined in Table VI. In order to find inconsistencies after the delegation operation, we will first combine all the rules defined in Table IV, V and VI and then generate the decision tree. After applying the C4.5 algorithm (with enforcement of condition), we get the decision tree shown in Figure 4(b). One can see that the *Technician* is simultaneously *allowed* and *denied* to perform the *delete* operation.

VII. COMPLEXITY

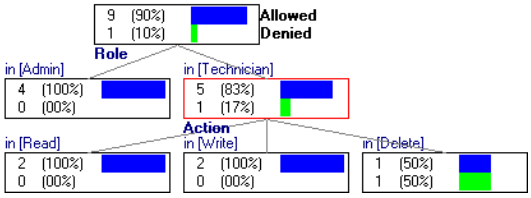
The order of complexity of the C4.5 algorithm is [18]:

$$O(m\ n\ log\ n) + O(n\ (log\ n)^2)$$

where n is the size of the training data (in our case number of rules) and m is the number of attributes. $O(m\ n\ log\ n)$



(a) Decision tree before delegation



(b) Decision tree after delegation

Fig. 4. Decision trees for multiple sets

represent the complexity for building complete decision trees and $O(n\ (log\ n)^2)$ is required for sub-tree raising (pruning). In our proposed inconsistency detection method, we are only interested in building a complete decision tree. So the complexity of our method for building decision tree is $O(m\ n\ log\ n)$. Figure 5 shows that when the number of rules increases, the complexity of the proposed method also increases linearly.

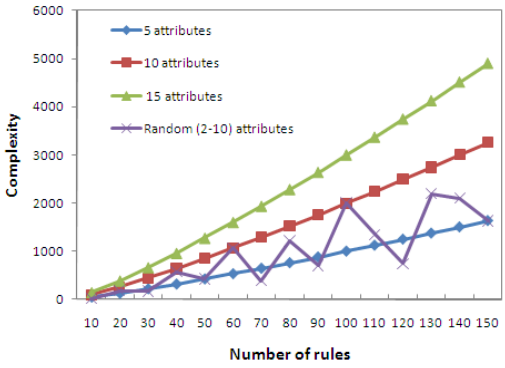


Fig. 5. Complexity of proposed method

VIII. Conclusion and Future Work

Many researchers have tried to solve the problem of detecting inconsistencies in access control policies using various methods, mostly based on formal logic. However, this approach suffers from exponential growth of computational complexity, and cannot deal easily with numerical or continuous values. In this paper, we have come up with a simple, efficient and practical solution for detecting such inconsistencies. We use a modified form of the C4.5 data mining algorithm that is very efficient and has linear computational complexity.

In this paper, we have not considered situations in which users belong to two or more groups or roles and by this fact can be simultaneously permitted and prohibited to access given resources, leading to conflicts. Detecting inconsistencies in such situations is our next goal. We are also working on techniques to deal with rules involving complex logical conditions.

Acknowledgment

The work reported in this article was partially supported by the Natural Sciences and Engineering Research Council of Canada and CA Labs. The authors would like to thank all members of the Computer Security Research Lab (UQO, Canada) for providing useful comments and suggestions.

References

[1] B. Stepien, S. Matwin, and A. Felty, "Strategies for reducing risks of inconsistencies in access control policies," in *Proc. of the 5th Int. Conference on Availability, Reliability and Security (AReS 2010)*, Feb 2010, pp. 140–147.

[2] N. Dunlop, J. Indulska, and K. Raymond, "Dynamic conflict detection in policy-based management systems," in *Proc. of the 6th international Enterprise Distributed object Computing Conference*, CA, USA, 2002, p. 15.

[3] S. Benferhat, R. El Baida, and F. Cuppens, "A stratification-based approach for handling conflicts in access control," in *Proc. of the 8th ACM Symp. on Access control models and technologies*. NY, USA: ACM, 2003, pp. 189–195.

[4] C.-J. Moon, W. Paik, Y.-G. Kim, and J.-H. Kwon, "The conflict detection between permission assignment constraints in role-based access control," *Lecture notes in computer science*, vol. 3822, pp. 265–278, 2005.

[5] F. Cuppens, N. Cuppens-Boulahia, and M. B. Ghorbel, "High level conflict management strategies in advanced access control models," *Electronic Notes in Theoretical Computer Science*, vol. 186, pp. 3–26, July 2007.

[6] K. Adi, Y. Bouzida, I. Hattak, L. Logrippo, and S. Mankovskii, "Typing for conflict detection in access control policies," *Lecture Notes in Business Information Processing*, vol. 26, pp. 212–226, 2009.

[7] M. G. Gouda and A. X. Liu, "Structured firewall design," *Computer Networks*, vol. 51, no. 4, pp. 1106–1120, 2007.

[8] J. R. Quinlan, "Induction of decision trees," *Mach. Learn*, vol. 1, no. 1, pp. 81–106, March 1986.

[9] ——, *C4.5: Programs for Machine Learning*. USA: Morgan Kaufmann Publishers, 1993.

[10] B. Cestnik, I. Kononenko, and I. Bratko, "Assistant 86: A knowledge elicitation tool for sophistical users," in *Proc. of the 2nd European Working Session on Learning*, 1987, pp. 31–45.

[11] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proc. of the 27th Int. conference on Software engineering*, NY, USA, 2005, pp. 196–205.

[12] H. Hu and G. Ahn, "Enabling verification and conformance testing for access control model," in *Proc. of the 13th ACM Symp. on Access control models and technologies*. New York, NY, USA: ACM, 2008, pp. 195–204.

[13] M. Mankai and L. Logrippo, "Access control policies: Modeling and validation," in *Proc. of NOTERE 2005*, 2005, pp. 85–91.

[14] D. Jackson, "Automating first-order relational logic," *ACM SIGSOFT Software Eng. Notes*, vol. 25, no. 6, pp. 130–139, Nov 2000.

[15] O. Zaiane, "Chapter7: Data classification," http://webdocs.cs.ualberta.ca/~zaiane/courses/cmput690/slides/Chapter7/index.htm, 1999.

[16] S. Kotsiantis, "Supervised machine learning: A review of classification techniques," *Informatica*, vol. 31, pp. 249–268, 2007.

[17] R. Rakotomalala, "Sipina data mining software," http://eric.univ-lyon2.fr/~ricco/sipina.html, 2009.

[18] I. H. Witten and E. Frank, *Data mining: Practical machine learning tools and techniques with java implementations*. USA: Morgan Kaufmann Publishers, 1999.