Detecting Incompleteness in Access Control Policies Using Data Classification Schemes

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 Inc.
125 Commerce Valley DR W, Thornhill
Ontario, Canada.
Email: serge.mankovski@ca.com

***Abstract*—In a set of access control policies, incompleteness is the existence of situations for which no policy applies. Some of these situations can be exploited by attackers, to obtain unintended access or to compromise integrity. Such cases can be difficult to foresee, since typical policy sets consist of thousands of rules. In this paper, we adopt data classification techniques widely used in the machine learning community for detecting incompleteness in sets of access of control policies. To the best of our knowledge, we are the first ones to use data classification algorithms to detect incompleteness in sets of access control policies. We show that our proposed solution is simple, efficient and practical.**

***Index Terms*—Access control, Data classification, Incompleteness, Policy validation.**

I. Introduction

In enterprise environments, permission to access resources is protected by employing various access control methodologies. Based on the enterprise security requirements, a policy administrator selects an appropriate access control method, such as RBAC [1] or OrBAC [2]. According to the access control model, the policy administrator defines policies. The complexity of an access control system is dependent on number of factors, such as number of subjects, number of resources, security requirements, access control model etc. Because of these factors, it is highly possible that a system may contains anomalies, such as *inconsistencies* and *incompleteness* in access control policies.

Incompleteness is the existence of situations for which no policy applies. Some of these situations can be exploited by attackers, including the author of the policy or the owner of the database, to obtain unintended access or to compromise integrity. Because of the complexity, distribution and size of typical policy sets, resolving the incompleteness detection problem is difficult and challenging.

Existing research work has mainly focused on the detection of inconsistencies in access control policies [3], [4], [5], [6], [7], [8]. However, less importance has been given to resolving the completeness problem. Traditionally, completeness checks can be performed by the policy administrator manually, and completeness can be achieved by adding new rules in the rule set. For example, in the case of the OrBAC model [2], the policy administrator may be able to achieve this goal by adding negative authorizations. In some systems, incompleteness is resolved by denying access in the unspecified cases. However, these meta-rules may not reflect the intention of the security administrator. Therefore, these anomalies should be brought to the attention of the security administrator. Detecting incompleteness manually in sets of policies is a very cumbersome job for large sets of policies. The difficulty increases when contextual conditions (e.g. time and location) are included in policies. Therefore, an automated mechanism or tool is highly needed to assist policy administrators in detecting anomalies, and validating policies.

In this paper, we adopt data classification techniques widely used in the machine learning community for detecting incompleteness in sets of access of control policies. We have applied three different data classification algorithms such as Limited Search Induction Algorithm (LSIA) [9], C4.5 [10] and ASSISTANT'86 [11]. We show that the LSIA and C4.5 data classification algorithms (*with some extension that we have proposed*) are very efficient in detecting incompleteness in sets of access control policies. We show that our solution is simple and practical. To the best of our knowledge, we are the first ones to use data classification algorithms to detect incompleteness in sets of access control policies.

The rest of the paper is organized as follows. Section II presents concepts and definitions. Section III contains a description about our proposed incompleteness detection strategy. Section IV shows detailed demonstration of proposed solution. Section V contains discussion. Finally, Section VI concludes the paper and discusses future work.

II. Concepts and Definitions

A policy set must contain at least one rule. In terms of data mining, 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 feature of a particular rule and contains some discrete or continuous value. On the other hand, a category attribute represent the class to which a rule belongs. Typically, a category attribute takes only the values {*Allowed*, *Denied*, *Not Applicable*}, or similar.

From the perspective of data classification, we can formally define incompleteness in following manner. Let $\Re$ be a set of rules ($\Re = \{R_1, R_2, \ldots, R_n\}$), where $\Re \neq \phi$. Rules $R \in \Re$ have an 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 class attribute of the rule.

Let $\Upsilon(A_j)$ denote the set of all values assigned to an attribute A_j. Let $\upsilon(R_i.A_j)$ denote the value assigned to an attribute A_j in rule R_i.

Definition 1: For each decision-making attribute $A_j \in A$ if

$$\bigcup_{i=1,..,m} \upsilon(R_i.A_j) \subset \Upsilon(A_j)$$

then $\Re$ is *incomplete* with respect to attribute A_j (where $\subset$ denotes a *proper subset*).

In the above definition, each attribute defined in a rule has only one value. If more than one value is assigned to a single attribute then each case will be handled individually.

Example 1. Let us assume that a set of rules only uses one attribute called *Day*. It can take seven possible values.

$$Day = \{Mon, Tue, Wen, Thu, Fri, Sat, Sun\}$$

If no rule is defined for a particular *Day* (say *Friday*), then according to our definition, the rule set $\Re$ is *incomplete* with respect to attribute *Day*.

III. Incompleteness Detection Strategy

Our proposed incompleteness detection method consists of the five steps shown in Figure 1. Details about each step is given below.

Step#1: *Get policies for each resource set.*
Access control rules are classified with respect to the resource set. Here, the resource set is anything (soft or hard) that has an identity and requires permission for access. A resource set may contain single or multiple things but could not be empty. For example, a resource set could be medical records etc. Our algorithm separates the policies according to the resources they refer to, since policies that refer to different resources cannot be in conflict.

Step#2: *For each resource set, we need to define non-category attributes.*
As defined earlier, non-category attributes are the decision making attributes, such as users (e.g. Doctor, Alice), and context (e.g. location, time). Each resource set can be described by different non-category attributes. These attributes can be either defined by the policy administrator or can be obtained by applying some rule mining or parsing technique on each set.

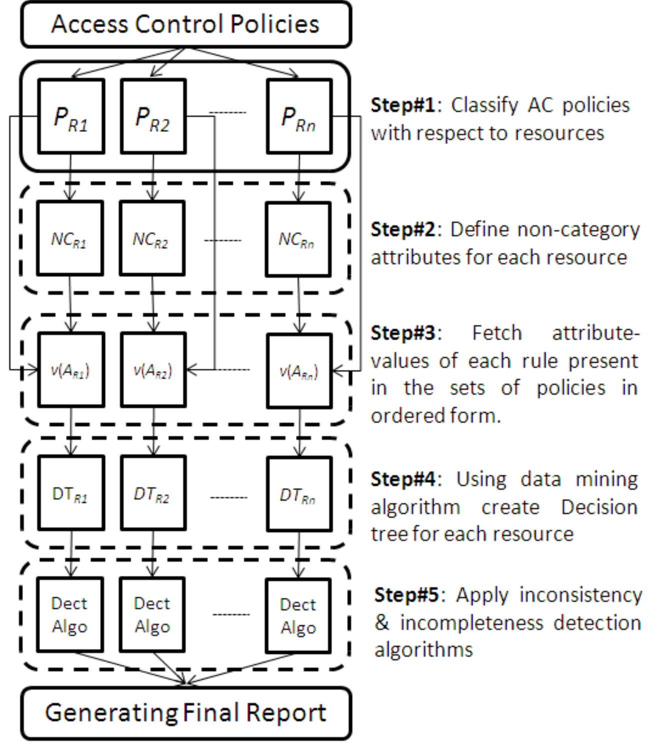


Fig. 1. Incompleteness Detection Method

Step#3: *Fetch attribute-values of each rule.*
For each rule related to a particular resource set, we need to fetch attribute values. In order to provide valid input to the data mining algorithm, all attributes must be in an ordered collection. For example, Subject, Operation, Location, Time etc. If a specific attribute is not used in a particular rule, then we can use a default value.

Step#4: *Decision tree generation for each resource set.*
In order to detect incompleteness in the policy set that controls access to a resource, we first generate a decision tree by applying a data mining algorithm. By default, standard data classification algorithms such as C4.5 generates compact or optimized trees, which may not contains all attributes. In order to ensure that all non-category attributes are present in a decision tree, we need to ensure that the following condition holds:

$$T_\alpha \geq A_{r_i} + 1 \tag{1}$$

where, T_α represent the depth (levels) of the decision tree, A_{r_i} represents the total number of non-category attributes defined for resource i and 1 is for the root node which represents the category attribute. This condition can be obtained by a modification in the standard data classification algorithm such

TABLE I
POLICY SET FOR EMPLOYEE‘S RECORD

	Role	Location	Time	Permission
R_1	Doctor	-	-	Denied
R_2	Admin Staff	General ward	9:00-17:00	Denied
R_3		Emergency ward	17:01-8:59	Denied
R_4		Admin office	9:00-17:00	Allowed
R_5			17:01-8:59	Allowed

as C4.5 algorithm. The algorithm so modified will be said to be *extended*.

Step#5: *Incompleteness Analysis*
In a decision tree, each branch b_i (from the root to a terminal node) represents one rule. In order to detect incompleteness in the decision tree, we will apply Algorithm 1. First we check the terminal node of each branch (Lines: 3-4). If any terminal node t_{node} does not contains any category (C) attribute value (Line: 4), this means that no explicit rule is defined in the specific context for particular user(Line:7). The information about the user and context will be fetched from the complete branch (root to terminal node) (Line: 5). If a category value is assigned to all the terminal nodes then this means that the policy set is complete (Lines: 11-13).

Algorithm 1 Incompleteness Detection Algorithm
Input: Decision tree
Output: Context of incompleteness

1: Let $A(b_i)$ be the set of all attributes present in one branch.
2: Bool $complete = true$;
3: **for** each branch b_i in Decision tree **do**
4:    **if** no category attribute is assigned to terminal node $b_i.t_{node}$ **then**
5:       A = fetch_all_attributes_of_branch(b_i);
6:       Policy set is incomplete w.r.to $label(b_i.t_{node})$;
7:       Complete context: $A(b_i)$;
8:       $complete = false$;
9:    **end if**
10: **end for**
11: **if** $complete = true$ **then**
12:    No incompleteness found;
13: **end if**

IV. EXAMPLE

A. Example 1

Let us assume that we have one resource called employee‘s records. A sample set of rules that includes five rules is given in Table I.

This table shows that the resource is accessible to the two set of users (doctors and admin staff) in different contexts, such as location and time. The set of rules includes three non-category attributes: A_1 = Role , A_2 = Location, and A_3 = Time. Possible values for these attributes are given in Table III. Also,

TABLE II
NON-CATEGORY ATTRIBUTES FOR RESOURCE: EMPLOYEE‘S RECORD

Attribute	Possible values
Role	Role-1: Doctor Role-2: Admin staff ...
Location	L1: General Ward L2: Emergency ward L3: Admin office ...
Time	T1: 9:00-17:00 T2: 17:01-8:59 ...

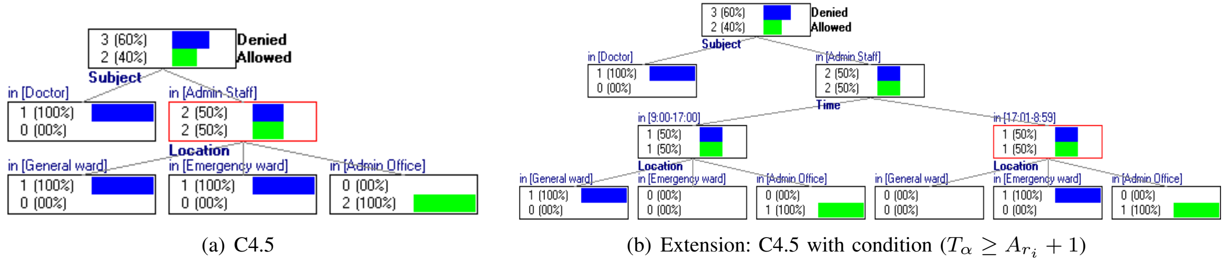
we have two permission category attribute values: *Allowed* and *Denied*.

In order to generate the decision tree for this policy set, we have applied three data mining algorithms: 1) C4.5 [10], 2) Limited Search Induction Algorithm (LSIA) [9], and 3) ASSISTANT86 [11]. For this purpose, we have used the Sipina data mining software package developed by Ricco Rakotomalala in the ERIC Research laboratory [12].

Figure 2(a) shows the decision tree generated by the C4.5 algorithm. One can see that the depth of the decision tree is three. In this tree, not all defined attributes are present, for example the attribute time is not present. When we add the condition ($T_\alpha \geq A_{r_i} + 1$) in the C4.5 algorithm, then we get the decision tree shown in Figure 2(b). In this tree, all essential attributes are present and the most important thing that we need to notice is that no category attribute (*Allowed* or *Denied*) exists at two terminal nodes (2nd and 4th from left at the last level). This shows that the policy set of this resource is incomplete with respect to the location attribute in certain time contexts. This small extension (enforcement of condition: $T_\alpha \geq A_{r_i} + 1$) in C4.5 algorithm, provides a more clear and complete picture of the domain.

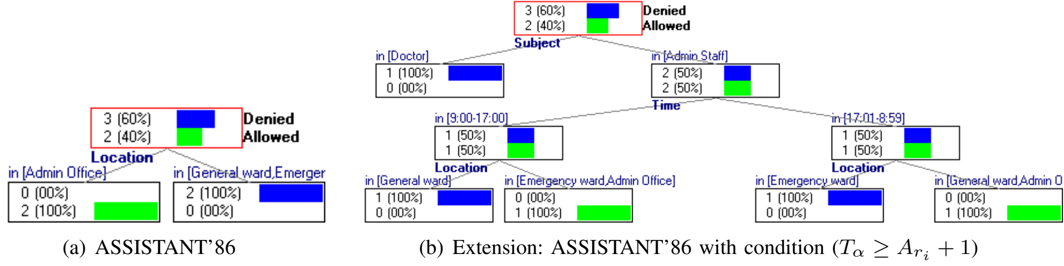
When we applied the algorithm ASSISTANT’86 on the same policy set, we obtained a more compact tree, shown in Figure 3(a). Similar to the C4.5, this algorithm does not provide a complete tree. When we modified the algorithm to ensure condition 1, we obtained the complete tree shown in Figure 3(b). However, this algorithm is still not useful to detect incompleteness.

When we applied the standard limited search induction algorithm (LSIA) and modified form of the LSIA on the same policy set, we obtained the decision trees shown in Figure IV-A. The LISA algorithm generates more condensed trees (Figure 4(a)) as compared to the C4.5 algorithm (Figure 2(a)). In Figure 4(a) , two essential attributes, role and time are not present. In order to get a complete tree, we applied the proposed modified form of LSIA shown in Figure 2(b). This figure shows that the extended form of LISA is also capable of detecting incompleteness in set of access control policies. However, it requires more iterations as compared to the extended form of the C4.5 algorithm.



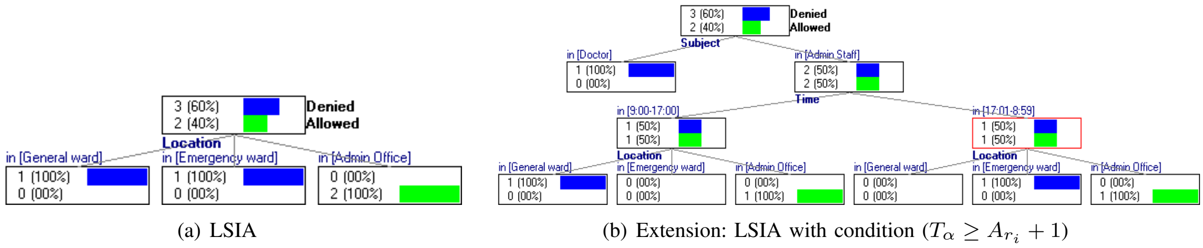
(a) C4.5 (b) Extension: C4.5 with condition ($T_\alpha \geq A_{r_i} + 1$)

Fig. 2. C4.5: Decision trees for employee's record



(a) ASSISTANT'86 (b) Extension: ASSISTANT'86 with condition ($T_\alpha \geq A_{r_i} + 1$)

Fig. 3. ASSISTANT'86: Decision trees for employee's record



(a) LSIA (b) Extension: LSIA with condition ($T_\alpha \geq A_{r_i} + 1$)

Fig. 4. LSIA: Decision trees for employee's record

B. Example 2

Let us assume that we have two subjects *Alice* and *Bob*. Both are allowed to work in a departmental store on different week days as shown in Table III.

TABLE III
POLICY SET FOR EXAMPLE 2

	Subject	Day	Permission
R_1	Alice	MON	Allowed
R_2	Bob	MON	Denied
R_3	Bob	TUE	Allowed
R_4	Alice	WEN	Allowed
R_5	Bob	WEN	Denied
R_6	Alice	THU	Denied
R_7	Alice	FRI	Allowed
R_8	Bob	FRI	Denied

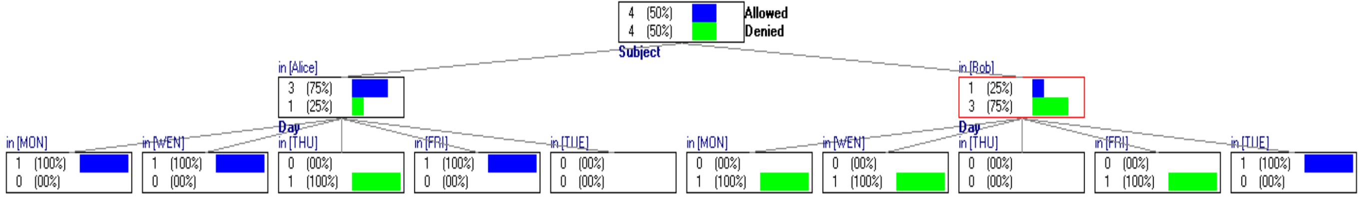
In order to find the any incompleteness in this policy set, we have applied three (C4.5, ASSISTANT and LSIA) data classification algorithms we have just seen. Due to page limit restrictions, we have only shown the decision trees generated by the extended version of these three algorithm. In Figure 5, one can see that only the extended form of the C4.5 algorithm found two incompleteness situations in the policy set.

TABLE IV
COMPARISON OF DATA CLASSIFICATION ALGORITHMS

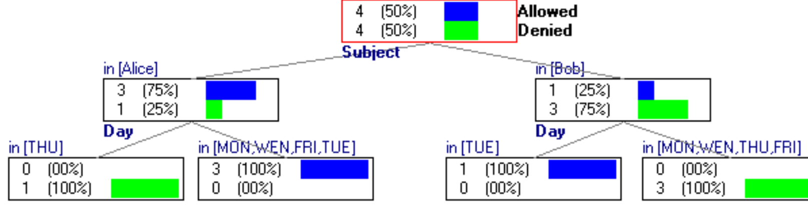
	Incompleteness Detection			
	Exp. 1	Extra iterations	Exp. 2	Extra iterations
C4.5	No	-	No	-
C4.5 with extension	Yes	1	Yes	1
LSIA	No	-	No	-
LSIA with extension	Yes	2	No	0
ASSISTANT'86	No	-	No	-
ASSISTANT'86 with extension	No	2	No	2

V. DISCUSSION

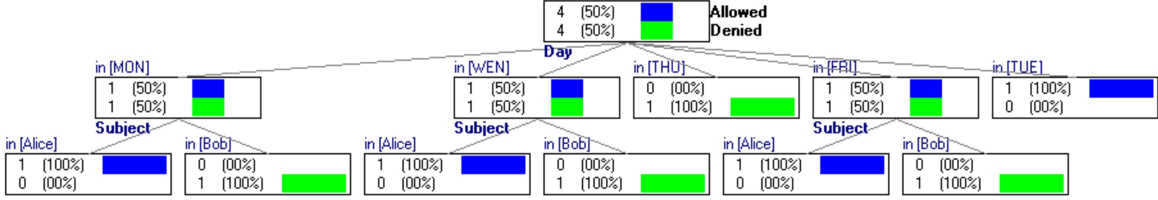
Table IV summarizes the comparison of three data classification algorithms. In this table, extra iterations represents



(a) Extension: C4.5 with condition ($T_\alpha \geq A_{r_i} + 1$)



(b) Extension: ASSISTANT'86 with condition ($T_\alpha \geq A_{r_i} + 1$)



(c) Extension: LSIA with condition ($T_\alpha \geq A_{r_i} + 1$)

Fig. 5. Decision trees for example 2

additional iterations that are required to generate complete decision trees by the extended version of the algorithm as compared to the original one. From experiments, we here concluded that not all data mining algorithms are capable of detecting incompleteness in sets of access control policies, for example the ASSISTANT'86 algorithm cannot. If we enforce condition ($T_\alpha \geq A_{r_i}+1$) then some data classification algorithms such as C4.5 (in general) and LSIA (in some cases) are useful to detect incompleteness in sets of access control policies. Also, in terms of detecting incompleteness and computation (number of iterations), the extended form of the C4.5 is the best. The values of 'extra iterations' that are given in the Table IV are only for the examples presented in previous section. These can change for different data sets.

Witten and Frank [13] have calculated an ordered complexity of the C4.5 algorithm in following manner.

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

where n is the size of the training data (in our case the number of rules) and m is the number of attributes. $O(m\ n\ log\ n)$ represent the complexity for building complete decision tree and $O(n\ (log\ n)^2)$ is required for sub-tree raising (pruning). In our proposed incompleteness detection method, we are only interested in building a complete decision tree. So the complexity of our method for building decision trees is $O(m\ n\ log\ n)$.

VI. Conclusions and Future work

In this work, we have investigated the use of data classification techniques for detecting incompleteness in access control policies. We applied three different data classification algorithms such as, Limited Search Induction Algorithm (LSIA) [9], C4.5 [10] and ASSISTANT'86 [11]. We show that the C4.5 data classification algorithm (*with some modification that we have proposed*) is very efficient in detecting incompleteness in sets of access control policies. We show that our solution is simple, efficient and practical. To the best of our knowledge, we are the first ones to use data classification algorithms to detect incompleteness in sets of access control policies. Also, the computational complexity of our proposed method is linear. Note however that the current method does not taken into consideration complex boolean conditions that are often used in access control policies. Those conditions are the subject of future work.

Acknowledgments

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) specially to Hemanth Khambhammettu and Ji Ma for providing useful comments and suggestions.

REFERENCES

[1] D. Ferraiolo, D. Kuhn, and R. Chandramouli, *Role-based access control*. Artech House Publishers, 2003.

[2] A. Kalam, R. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin, "Organization based access control," in *Proceedings of the IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*. Los Alamitos, CA, USA: IEEE Computer Society, 2003, pp. 120–131.

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

[4] N. Dunlop, J. Indulska, and K. Raymond, "Dynamic conflict detection in policy-based management systems," in *Proceedings of the Sixth international Enterprise Distributed object Computing Conference (EDOC'02)*. Los Alamitos, CA, USA: IEEE Computer Society, 2002, p. 15.

[5] S. Benferhat, R. El Baida, and F. Cuppens, "A stratification-based approach for handling conflicts in access control," in *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*. New York, NY, USA: ACM, 2003, pp. 189–195.

[6] 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.

[7] 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.

[8] 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.

[9] J. Catlett, "Megainduction : Machine learning on very large databases," PhD Thesis, School of Computer Science, University of Technology, Sydney, Australia, 1991.

[10] J. R. Quinlan, *C4.5: Programs for Machine Learning*. USA: Morgan Kaufmann Publishers, 1993.

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

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

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