# Anomaly Discovery and Resolution in MySQL Access Control Policies

Mohamed Shehab, Saeed Al-Haj, Salil Bhagurkar, Ehab Al-Shaer
Department of Software and Information Systems

University of North Carolina at Charlotte
Charlotte, NC, USA
{mshehab,salhaj,sbhagurk,ealshaer}@uncc.edu

**Abstract.** Managing hierarchical and fine grained DBMS policies for a large number of users is a challenging task and it increases the probability of introducing misconfigurations and anomalies. In this paper, we present a formal approach to discover anomalies in database policies using Binary Decision Diagrams (BDDs) which allow finer grain analysis and scalability. We present and formalize intra-table and inter-table redundancy anomalies using the popular MySQL database server as a case study. We also provide a mechanism for improving the performance of policy evaluation by upgrading rules from one grant table to another grant table. We implemented our proposed approach as a tool called *MySQLChecker*. The experimental results show the efficiency of *MySQLChecker* in finding and resolving policy anomalies.

**Keywords:** Policy, Access Control, Policy Analysis, Anomaly Detection

## 1 Introduction

Large DBMSs used in applications such as banks, universities and hospitals can contain a large number of rules and permissions which have complex relations. Managing such complex policies is a highly error-prone manual process. Several surveys and studies have identified human errors and misconfigurations as one of the top causes for DBMS security threats [3]. This is further magnified by the adoption of fine grain access control that allows policies to be written at the database, table and row levels. In addition, once policy misconfigurations are introduced they are hard to detect.

There has been recent focus on policy anomaly detection, especially in firewall policy verification [2, 8]. The approaches proposed are not directly applicable to database policies due to several reasons. First, firewall policies are based on first-match rule semantics, in database policies other semantics such as most specific are adopted. Second, the firewall polices are flat, on the other hand database policies are hierarchical and are applied at different granularity levels. Third, firewall policies are managed in a single access control list, while database policies are managed by multiple access control lists. Therefore, new mechanisms for policy analysis and anomaly detection are required database policies.

In this paper, we use MySQL as a case study due to its wide adoption. MySQL provides a descriptive fine grain policy language which provides an expressive

policy language. First we start by introducing the details of group-based policies used in MySQL. We define and formalize the set of possible access violations. To aid the administrator in detecting and avoiding these violations we propose a framework to detect and resolve the detected violations. To the best of our knowledge, this is the first time such a policy violation detection and resolution mechanism for DBMS policies has been proposed. We implemented our proposed anomaly detection and resolution approaches as a tool (*MySQLChecker*). The efficiency and scalability of *MySQLChecker* was demonstrated through our experimental evaluations. The main contributions of the paper are:

- We formulate and model database access control policies using BDDs.
- We classified the set of possible database policy anomalies and proposed algorithms to detect and resolve the detected anomalies using BDDs.
- With a proof of concept implementation of our proposed anomaly detection and resolution, we have conducted experiments using synthetic policies.

The rest of the paper is organized as follows: In Section 2, we provide a brief background of database policies, access control in the MySQL framework and Binary Decision Diagrams. In Section 3, we define the formal representation of database grant rules, tables, and policies using BDDs. In Section 4, we define the set of possible anomalies in the MySQL context, and describe how the BDDs are used to detect and resolve these anomalies. Our implementation and experimental results are described in Section 5. Finally, we wrap up the paper with related work and conclusions.

## 2 Preliminaries

In this section we present the preliminaries related to MySQL access control, and Binary Decision Diagrams.

### 2.1 Database Policies

Most DBMS use access control lists to specify and maintain access control policies. For example, in MySQL access control lists (grant tables) are stored in multiple tables within the DBMS itself. The policy adopted is based on a closed world policy model, where access is allowed only if an explicit positive authorization is specified, otherwise access is denied. The grant tables include *user*, *db*, *tables_priv*, *columns_priv* and *procs_priv* which are used to store privileges at global, database, table, column and procedure levels respectively. Database privileges include several permissions, for example SELECT, INSERT, etc.

An access rule $R$ can be represented as a tuple that indicates the permissions granted to a user over a given object. A rule is defined as $R = \{user, host, db, table, column, privs\}$, where $R[user]$, $R[host]$, $R[db]$, $R[table]$, $R[column]$, $R[privs]$ refers to the rule's user, host, database, table column names and set of privileges respectively. To refer to a group of hosts, IP address and domain wildcards are allowed in $R[host]$. For example, *192.168.1.%* refers to all hosts in the *192.168.1* class C network, and *%.abc.com* refers to any host in the *abc.com* domain. Database names can also include wild characters.

Access control in MySQL involves both *Connection* and *Request Verification*. Connection verification verifies if a user connecting from a specific host is allowed

to login to the database. If access is granted, the request verification stage verifies if the user has access to the objects requested. Request verification checks user access from higher to lower granularity levels, for example database level, then table level, and then column level.

### 2.2 Binary Decision Diagrams (BDDs)

Binary Decision Diagrams (BDDs) are a type of symbolic model checking. A BDD is a directed acyclic graph [4, 5] used to represent boolean functions. The graph has a root and set of terminal and non-terminal nodes. Each non-terminal node represents a binary variable. Non-terminal nodes have two edges at most, high and low. High edge represents the true assignment for that variable while low edge represents the false assignment. Terminal nodes are two nodes representing the values *true* and *false*. BDDs were used efficiently in anomaly discover in access control list in Firewalls [2] and XACML [7] policies. Utilizing BDD operations such as and, or and negation can be used to efficiently implement set operations on Boolean expressions such as intersection and union.

## 3 Formal Representation

In this section, we will present the formal representation for the MySQL policy evaluation process using BDDs to encode the MySQL grant tables and rules.

### 3.1 Rules Modeling

A rule has two parts, condition and privilege vector. The matching request has to match rule conditions in order to grant a set of privileges. A rule can be represented formally as $R_i : C_i \rightsquigarrow PV_i$, where $C_i$ are the conditions for the $i^{th}$ rule that must be satisfied in order to grant privileges Vector $PV_i$. The condition $C_i$ can be represented as a Boolean expression of the filtering fields, $f_1$, $f_2$, ..., $f_k$ as $C_i = f_1 \wedge f_2 \wedge \ldots \wedge f_k$. The fields can be *host IP*, *user name*, *database name*, *table name* and *column name*.

Each grant table has different set of fields to represent the matching condition for rules in that table. For example, *users* grant table uses host IP and user name to match rules in the table. The *db* grant table uses host name, user name, and database name for condition matching. Privileges vector is the decision vector for each rule that specifies what are the privileges that will be granted when the rule is triggered. Formally as $PV = P_1 \wedge P_2 \wedge \ldots \wedge P_m$, where $m$ is the total number of permissions in the system, $P_i$ is a Boolean variable representing the $i^{th}$ permission. When a permission is granted, its corresponding Boolean variable is set to TRUE. For example, in Figure 1, the permission vector for *bob@152.150.10.2* is $|1, 0, 0|$ which means allowing SELECT and denying INSERT and UPDATE.

| Rule | User | Host | DB name | Select | Insert | Update |
|---|---|---|---|---|---|---|
| $r_1$ | alice | localhost | % | Y | Y | Y |
| $r_2$ | bob | 152.150.10.2 | Emp | Y | N | N |

**Fig. 1.** *db* grant table example

### 3.2 Grant Tables Modeling

To evaluate a request in MySQL policy, there has to be a rule in one of the grant tables that allows this request. A grant table is modeled as a Boolean formula using BDDs. Each grant table is a sequence of filtering rules, $R_1, R_2, ..., R_n$. The rules are checked in the first match semantic when a request is matched.

In the first match semantic, policy evaluation starts from the first rule, then the second rule and so on until a matching rule is found. For example, when the third rule is matched, the first and the second rule are not matched. The formal rule ordering in this is represented as $R_1 \vee (\neg R_1 \wedge R_2) \vee (\neg R_1 \wedge \neg R_2 \wedge R_3)$. The previous formula shows the rules ordering only, it did not show the how rules conditions and permissions are encoded in the grant table. Formally, each grant table will be constructed as $X_{BDD} = \bigwedge_{i=1}^{m} X_{BDD}^{i}$, where $X_{BDD}$ is the BDD representation for any of the grant tables, and $X_{BDD}^{i}$ is the BDD representation for the $i^{th}$ permission in $X$ grant table.

Each permission in the permission vector is represented by a BDD. The $k^{th}$ permission BDD is $X_{BDD}^{k} = \bigvee_{i=1}^{n} \neg C_1 \wedge \neg C_2 \ldots \neg C_{i-1} \wedge C_i \wedge P_k$, where $n$ is the total number of the rules in the $X$ grant table and $k$ is the $k^{th}$ permission in the permission vector PV. Each of the grant tables will be encoded as a BDD. Where $U_{BDD}$, $D_{BDD}$, $T_{BDD}$ and $C_{BDD}$ represent the *users*, *db*, *tables_priv*, and *columns_priv* grant tables respectively.

### 3.3 Policy Modeling

In order to determine if a request is to be granted or not, the *users* grant table is checked first. If a matching rule is found in the table, the associated privileges are granted. Otherwise, *db* grant table is checked for a matching rule. MySQL policy checks grant tables in the following order: *users*, *db*, *tables_priv* and *columns_priv*. Formally, the BDD first match semantic for MySQL policy ($G_{MySQL}$) is represented as:

$$G_{MySQL} = U_{BDD} \bigvee (\neg U_{BDD} \wedge D_{BDD}) \bigvee (\neg U_{BDD} \wedge \neg D_{BDD} \wedge T_{BDD}) \bigvee (\neg U_{BDD} \wedge \neg D_{BDD} \wedge \neg T_{BDD} \wedge C_{BDD})$$

Given a request $Q$, the decision whether to grant this request or not is evaluated by intersecting $G_{MySQL}$ and $Q$. If the resultant Boolean expression is FALSE, then the request is denied. Otherwise, it is granted. Formally this operation can be represented as $Q \wedge G_{MySQL} \Leftrightarrow action, action \in \{TRUE \mid FALSE\}$.

## 4 MySQL Policy Anomalies

The hierarchal relation between grant tables and the first match semantic in evaluating requests can introduce different types of anomalies. There two types of anomalies in MySQL policies namely intra-table and inter-table redundancy.

### 4.1 Intra-Table Redundancy

Intra-table redundancy is the redundancy within the same grant table. Subset and superset relationships are the primary causes for rule redundancy. Two rules are intra-table redundant if they grant the same privileges for the same conditions. Rule redundancy can occur in all grant tables. The following intra-table redundancy definition applies for any grant table.

**Definition 1.** *Given a grant table $X_{BDD}$, a rule $R_i$ is intra-table redundant to rule $R_j$ for $i < j$ if:*

$$(C_i \subseteq C_j) \bigwedge (PV_i = PV_j) \bigwedge (\nexists R_k (i < k < j \wedge \ C_i \subseteq C_k \wedge PV_i \neq PV_k)) .$$

Definition 1 covers intra-table redundancy case in which the preceding rule $R_i$ is a subset from the superset rule $R_j$. Unlike firewall policies, a superset rule cannot appear before a subset rule because of the pre-sorting process. A rule is intra-table redundant if there is a rule with the same privileges vector follows the redundant rule. Definition 1 excludes the case of exceptions. Exceptions are not considered as redundant rules because they intended to perform different action on a subset from the later rule. We demonstrate this condition through the example in Figure 2(a). Note that rules $R_2$ and $R_3$ are not intra-redundant because they have different permission vectors. In the case of $R_2$ and $R_5$, even both rules have the same privilege vector and subset relationship between conditions, they are not considered redundant because there is another rule, $R_3$ that has different permission vector. When deleting rule $R_2$, bob@152.150.10.1 will not be able *insert* because the *insert* privilege has been revoked in rule $R_3$. Rules $R_1$ and $R_4$ are redundant. Deleting $R_1$ will not affect the policy semantics.

| Rule | User | Host | Select | Insert | Update |
|---|---|---|---|---|---|
| $R_1$ | alice | 152.150.40.55 | Y | N | N |
| $R_2$ | bob | 152.150.10.1 | Y | Y | N |
| $R_3$ | bob | 152.150.10.% | Y | N | N |
| $R_4$ | alice | 152.150.40.% | Y | N | N |
| $R_5$ | bob | 152.150.%.% | Y | Y | N |

(a) *users* grant table example

| Rule | User | Host | DB name | Table | Select | Insert | Update |
|---|---|---|---|---|---|---|---|
| $R_1$ | bob | 152.150.10.5 | Emp | manager | Y | Y | N |
| $R_2$ | bob | 152.150.10.% | Emp | human_resources | Y | N | N |
| $R_3$ | bob | 152.150.%.% | Acc | human_resources | Y | N | Y |
| $R_4$ | alice | 169.12.%.% | Emp | manager | Y | Y | Y |

(b) *tables_priv* grant table example

| Rule | User | Host | DB name | Table | Column | Select | Insert | Update |
|---|---|---|---|---|---|---|---|---|
| $R_1$ | bob | 152.150.10.% | Emp | manager | name | Y | Y | N |
| $R_2$ | bob | 152.150.10.% | Emp | human_resources | id | Y | Y | Y |
| $R_3$ | bob | 152.150.%.% | Acc | human_resources | salary | Y | N | N |
| $R_4$ | alice | 169.12.25.% | Emp | manager | id | Y | Y | Y |

(c) *columns_priv* grant table example

**Fig. 2.** Intra and Inter-Table Redundancy Example

### 4.2 Inter-Table Redundancy

Inter-table redundancy appears between two rules in different grant tables. Considering all grant tables, we have *six* inter-table redundancy cases between all grant tables. Inter-table redundancy is *partial* or *complete*. Complete inter-table redundancy occurs when two rules in different grant tables have the same privileges vector for some common conditions. While the partial inter-table redundancy when some privileges are similar for some common conditions.

**Definition 2.** *Given two grant tables $X$ and $Y$, $X<Y$, having the BDD representation $X_{BDD}$ and $Y_{BDD}$ respectively, a rule $R_i \in Y_{BDD}$ is completely inter-table redundant by $R_j \in X_{BDD}$ if:* $(X_{BDD} \cap Y_{BDD} \neq \phi) \wedge (C_i \subseteq C_j)$.

To find complete inter-table redundancy, grant tables BDDs are compared together. Complete inter-table redundancy requires two conditions: 1) there is an overlap between grant table BDDs and 2) the superset rule appears in the upper level grant table.

**Definition 3.** *Given two grant tables $X$ and $Y$, $X<Y$, having the BDD representation $X_{BDD}$ and $Y_{BDD}$ respectively, a rule $R_i \in Y_{BDD}$ is partially inter-table redundant by $R_j \in X_{BDD}$ if:* $\exists m \exists k (X^m_{BDD} \cap \neg Y^m_{BDD} \neq \phi) \wedge (X^k_{BDD} \cap Y^k_{BDD} \neq \phi)$.

Where $X^m_{BDD}$ is the BDD representation for the $m^{th}$ permission in $X$ grant table. In partial inter-table redundancy, the privileges vectors are not the same. There are some permissions allowed in the upper level table and denied in the lower level table. Partial inter-table redundancy does not hold if a permission is denied in the upper level table and allowed in the lower level table. Partial inter-table redundancy requires at least one permission to be allowed in both rules. This is necessary to eliminate the case in which there is no rule exists in the lower level grant table. Figures 2(b)-2(c) provide inter-table redundancy examples. Rules $R_1$ in *tables_priv* and $R_1$ in *columns_priv* are not redundant because the upper table rule is not a superset rule. Rules $R_2$ in *tables_priv* and $R_2$ in *columns_priv* are not complete inter-table redundant because privileges vectors are not the same, also they are not partial inter-table redundant because the upper table rule denies the *insert* privilege while the lower table rule allows it. Rules $R_3$ in *tables_priv* and $R_3$ in *columns_priv* are partially inter-table redundant. Rules $R_4$ in *tables_priv* and $R_4$ in *columns_priv* are complete inter-table redundant.

### 4.3 Violations and Safety

A policy operation is safe if it does not cause the new policy to allow (deny) requests that were previously denied (allowed) by the original policy. Mainly safety is focused on maintaining the allow and deny space of the original policy.

**Definition 4.** *Policy $P_A$ is allow (deny) safe w.r.t policy $P_B$ iff every request allowed (denied) by $P_A$ is also allowed (denied) by $P_B$. An operation $OP$ that transforms $P_A$ to $P_B$ is safe iff $P_A$ and $P_B$ are both deny and allow safe.*

In what follows we will investigate the safety of the operations involving the removal of rules identified as redundant.

*Proof.* Let the original policy $P_A$ includes a rule $R_s$ is identified as a redundant violation. Let the policy $P_B$ be the policy generated after removing rule $R_s$ from $P_A$. Let $P_A$ be a policy with three rules: $R_1$, $R_2$ and $R_3$. Assume rule $R_1$ is an intra-table redundant with $R_3$. Using the BDD modeling described earlier, the formal representation for $P_A$ is $R_1 \vee (\neg R_1 \wedge R_2) \vee (\neg R_1 \wedge \neg R_2 \wedge R_3)$. Redundancy check requires $R_1 \subset R_3$. When $R_1 \subset R_3$, then $R_1 \cap R_3 \Rightarrow R_1$ and $R_1 \cup R_3 \Rightarrow R_3$. To simplify the formula, we can apply demorgan's law. The final result will be similar to $P_B$ that has only two rules $R_2$ and $R_3$. The formal representation for $P_B$ is $R_2 \vee (\neg R_2 \wedge R_3)$. After simplifying $P_A$ and $P_B$, the same Boolean expression is reached. Therefore, removing an intra-table redundant rule will not change the matching semantic for MySQL final policy. For the sake of simplicity we used 3 rules, the proof can be easily extend to any number of rules.

In case of inter-table redundancy anomaly, the redundant rule is located in the lower level grant table. Let the original policy $P_A$ includes a rule $R_s$ is identified as an inter-table redundant violation. Let the policy $P_B$ be the policy generated after removing rule $R_s$ from $P_A$. Let the subset rule, $R_s$, be in $Y_{BDD}$ grant table and the superset rule be in $X_{BDD}$. Before removing the redundant rule, the intersection of grant table BDDs is: $X_{BDD} \cap Y_{BDD} \neq \phi$. After removing the subset rule $R_s$ from $Y_{BDD}$ the intersection of the two grant table BDDs is $\phi$ and $R_s \cap X_{BDD} = R_s$, because of the presence of a superset rule that covers $R_s$ in $X_{BDD}$. Therefore, any request will be matched in the upper grant table after removing the inter-table redundant rule from the lower grant table. □

### 4.4 Algorithms

This section defines algorithms that are used to extract the list of intra/inter table redundant rule conditions in MySQL based on the BDD representation of

the policy. Algorithm 1 detects intra-table redundant rule by comparing each rule condition with its possible supersets succeeding it. Algorithm 2 detects complete inter-table redundant rules in the *db* grant table based on $U_{BDD}$ and $D_{BDD}$.

**Algorithm 1:** IntraRedRules

```
   Input: L_u (Sorted list of user rule BDDs)
   Output: L_r
 1 for i = 0 to |L_u| - 2 do
 2     for j = (i + 1) to |L_u| - 2 do
 3         P = L_u[i];
 4         N = L_u[j];
 5         I = N ∩ P;
 6         if P == I and PV_i == PV_j then
 7             L_r ∪ {(i, j)};
 8         end
 9         if P == I and PV_i != PV_j then
10             break;
11         end
12     end
13 end
14 return L_r
```

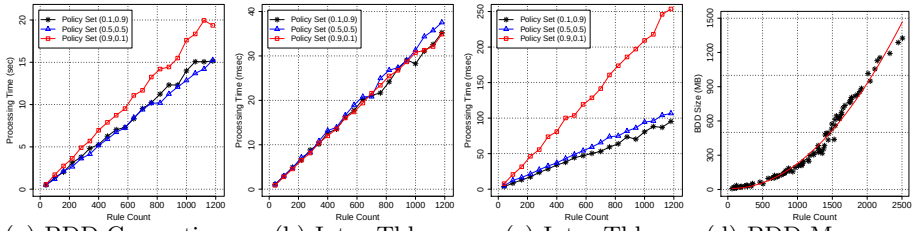
**Algorithm 2:** InterRedRules

```
   Input: U_BDD, D_BDD
   Output: L_s
 1 I = U_BDD ∩ D_BDD;
 2 L_s = φ;
 3 while (I) has satisfying assignments do
 4     b =One satisfying assignment of (I);
 5     L_s = L_s ∪ info(b);
 6     I = I ∩ (¬b);
 7 end
 8 return L_s;
```

**Fig. 3.** Algorithms.

## 5 Implementation and Evaluation

We developed and tested our framework against synthetic policies to show the scalability of the framework. It is difficult to get a large number of real-life MySQL policies as these policies are often regarded as confidential. We developed a policy generator engine that generates synthetic policies, with a specific number of rules at each level and probabilities of intra-table ($P_{intra}$) and inter-table ($P_{inter}$) redundancy. The experiments were performed on Mac OS X 10.5.5 with 4GB RAM and a 2.4GHz Dual Core, using the BuDDy library v2.4 and MySQL client library v5.1.




(a) BDD Generation (b) Intra-Tbl (c) Inter-Tbl (d) BDD Memory

**Fig. 4.** User Study Results and Algorithms Performance

The synthetic policies were analyzed by our framework and the BDD generation, intra-table and inter-table redundancy average processing times were recorded. Figure 4(a), shows the initial processing time required to build the BDD for different policy sets where ($P_{inter}, P_{intra}$) are the inter-table and intra-table redundancy probabilities respectively. The intra-table and inter-table redundancy processing times are reported in Figures 4(b) and 4(c) respectively. Note, that the BDD generation, intra-table and inter-table processing times are linear with respect to the number of policy rules. The discovery and resolution of the inter-table redundancy depends on the percentage of inter-table introduced in the synthetic policy, for a policy containing 1200 policy rules it takes around 90ms and 250ms for $P_{inter}$ values of 0.1 and 0.9 respectively, refer to Figure 4(c). In addition, we recorded the memory requirements for storing the BDD generated, the memory required is polynomial (degree 2) w.r.t the number of policy rules, the regression estimate ($R^2 = 0.983$) is plotted in Figure 4(d).

## 6 Related Work

BDDs were utilized to resolve anomalies in access control lists [2, 6, 7]. The work presented by Al-Shaer *et al.* in [2, 6] used BDDs to discover and resolve anomalies in network devices such as firewalls, IPSecs, etc. The work introduced enter-policy and intra-policy anomalies. Redundancy, shadowing, correlation and exception were resolved in this work. Web access control list anomalies were studies by Hu *et al.* in [7], where XACML policies were modeled using BDDs, which was used to discover and resolve conflicts and redundancy in both XACML policy and policy set levels.

Role based access control, which has made significant simplifications in the management of security policies. Roles represent functional roles in an enterprise and individual users acquire authorizations through their assigned roles. Research related to RBAC policy verification [1, 9] has focused on verifying RBAC implementation, Separation of Duty and role hierarchy constraints.

## 7 Conclusion

In this paper, we presented a formal approach to model and define anomalies in MySQL policies. We utilized Binary Decision Diagrams (BDDs) to encode MySQL policy and grant tables. We presented and formalized intra-table and inter-table redundancy anomalies. In addition, we provided a mechanism for improving the performance of policy evaluation by upgrading rules from one grant table to another grant table. We implemented our proposed approach as a tool called *MySQLChecker*. The experimental evaluation conducted on the *MySQLChecker* shows the efficiency and scalability of finding and resolving the presented policy anomalies.

## References


1. G.-J. Ahn and H. Hu. Towards realizing a formal rbac model in real systems. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, SACMAT '07, pages 215–224, New York, NY, USA, 2007. ACM.
2. E. S. Al-Shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM'04*, volume 4, March 2004.
3. Application Security Inc. Database security tips for 2012. *http://www.appsecinc.com/santa-breach/Database_Security_Tips_2012.pdf*, 2011.
4. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.
6. H. H. Hamed, E. S. Al-Shaer, and W. Marrero. Modeling and verification of ipsec and vpn security policies. In *ICNP 2005. 13th IEEE International Conference on Network Protocols*, pages 259–278, 2005.
7. H. Hu, G.-J. Ahn, and K. Kulkarni. Anomaly discovery and resolution in web access control policies. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 165–174, New York, NY, USA, 2011. ACM.
8. E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25:852–869, November 1999.
9. B. Shafiq, A. Masood, J. Joshi, and A. Ghafoor. A role-based access control policy verification framework for real-time systems. In *WORDS'05*, Feb. 2005.