The Computational Complexity of Enforceability Validation for Generic Access Control Rules

Vincent C. Hu, D. Richard Kuhn, and David F. Ferraiolo
National Institute of Standards and Technology, Gaithersburg, Maryland 20899-8930, USA
Email: {vhu, kuhn, dferraiolo}@nist.gov

Abstract

In computer security, many researches have tackled on the possibility of a unified model of access control, which could enforce any access control policies within a single unified system. One issue that must be considered is the efficiency of such systems, i.e., what is the computational complexity for the enforceability validation of access control rules of a system that is capable of implementing any access control policy? We investigate this question by arguing that two fundamental requirements exist for any such system: satisfiability of access rules and ensuring absence of deadlock among rules. We then show that both of these problems are NP-Complete by using some basic computational theorems applied to the components of the generic access control process.

1. Introduction

Access control policies can be as diverse as applications, and are heavily dependent on the needs of a particular environment. Some research has focused on a unified model or mechanism of access control, which could enforce any access control policies within a single unified system. As the modern theme in access control research is the separation of mechanism from policy, many researchers have provided elegant solutions for encompassing various access control models and policies in a unifying formalism. These works apply one of two approaches: The first is to provide configurable policy attributes and/or configurable enforcement mechanisms [1, 2, 3, 4]. The second is to provide policy (authentication) management systems or language to configure the authorization database [5, 6]. Thus, it is natural to speculate how efficient is a system that can enforce any arbitrary access control policy.

Endeavors have established in the computability analysis of access control systems such as: 1, finding the complexity of safety either through the use of limited access control model or the verification via constraints [10], and has shown that the safety of access control is undecidable [11]. 2, developing of flexible and efficient models for access control rule generalization [12, 13]. In our paper, we are not proposing another similar works as above, instead, we analyzed the rudimentary access control rule enforceability validation processes that is inevitable for a system that can implement any arbitrary access control policy no matter which access control mechanism is applied.

Although not every access control policy can be specified by an access control model (for example, Rule Based AC), a certain AC mechanism such as Access Control List (ACL), Access Control Matrix (ACM) are required in order to implement the policy. Therefore, to demonstrate the generic, therefore universal without being restricted to any specific model or mechanism, we surpass any known models/mechanism and worked from the elementary components that any access control policy is built upon, and for algorithms embedded in any access control mechanism operated with. These basic components are subjects, operations, and objects.

Our goal is to determine the computational complexity for a completely general access control system. Therefore, the question can be rephrased as: what is the computational complexity of the enforceability validation of a mechanism that is capable of implementing any access control policy? We investigate this question by arguing that two fundamental requirements for the enforceability validation of access control policy exist for any such system: satisfiability of access rules and ensuring absence of deadlock among rules. That is, we argue that these functions are necessary, although possibly not sufficient, for any access control policy. We then show that both of validation problems are NP Complete.

The following proof outline summarize our computational complexity argument for the rest of the paper:
1. To implement an access control policy, a mechanism must incorporate rules that are evaluated in each system state.
2. If the mechanism is capable of implementing an arbitrary policy, it must be capable of incorporating an arbitrary set of rules.
3. All rules must be satisfiable, i.e. satisfying truth assignments of Boolean expression.
4. No rule may be dependent on itself, i.e., deadlock or circular dependency is prohibited.
5. The problem of checking for satisfiability is NP-Complete.
6. The problem of checking for deadlock is represented by the AND-OR graph decision problem, which is also NP-Complete.
7. Consequently, the complexity for enforceability validation of a mechanism capable of implementing any conceivable access control policy is NP-Complete.

2. Authorization process

In access control, a privilege assignment refers to the association of a privilege to a subject, denoted by the triple <subject, operation, object>, indicating that the subject is permitted to perform the operation on the object. A subject refers to an active entity that typically includes users and system processes. An operation refers to a specific action applied to an object, such as read and write, and an object is a passive entity, such as files and printers, that require protection.

Abstractly, access control mechanisms apply a set of rules to system states for the purpose of allowing or denying a specified operation to an object by a subject. The rule set are composed according to the access control policy, such that the final process of any access control is the decision-making for a subject’s request to perform an operation on an object. To be universal, the operations must be arbitrary. Fig. 1 shows the relation mapping of an access control system from policy, model, and mechanism to the elementary algorithm. Note that not every policy can be described by a model (i.e. one policy can be modeled either by zero or one of the n known models as illustrated above of the arrows in the Figure1), however, every policy can be implemented by at least one of the n mechanisms, which can be implemented by an access control algorithm.

At an elementary level as in Figure 1 (Access Control algorithm), an access control system consists of the space of states and the space of rules. The states space contains privilege assignments permitted by the implemented policy. And for historical type of access control policies, it is required to maintain the past access states (already granted access history), therefore, each state may be in a status of already granted (+) or not-yet-granted (-). The status of a state is changed from –s to +s when the privilege of the state is granted as marked to be the past event, and changed from +s to –s when the s is required to be reset. Each state is expressed by the relation mapping a subject’s operations to access an object. The rules space contains information about the specified rules and/or constraints enforced by the implemented policy; each rule is generically expressed by the logic relations between security attributes with two types: the dynamic, which is the same as in the states space, we call them “dynamic”, because they can be either already-granted (+) or not-yet-granted (-), which is in contrast with the static attributes such as specific time, locations and other physical attributes related to security. The static attributes are set according to the access control policy or the access environment, such as requiring that an object can only be read at a particular time frame at a certain location. As examples, Table 1 shows the states and rules mapping of a simple MLS (Multilevel Security) policy with users and objects secrecy ranked by High (h), Medium (m), and Low (l), and any object access can only be accessed at the certain time frame t1, the policy is modeled by the Bell-LaPadule model [14]. Where read operation is denoted as r and write operation is denoted as w.

Table 1. The states and rules mapping of a MLS policy

<table>
<thead>
<tr>
<th>states</th>
<th>rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1=(h, r, h)</td>
<td>No restriction</td>
</tr>
<tr>
<td>s2=(h, r, m)</td>
<td>No restriction</td>
</tr>
<tr>
<td>s3=(h, r, l)</td>
<td>t1</td>
</tr>
<tr>
<td>s4=(m, r, m)</td>
<td>t1</td>
</tr>
</tbody>
</table>
Table 2 shows the states and rules mapping of a simple Chinese Wall policy [15] with only two users (groups) a and b, and two Conflict Of Interest (COI) classes x and y each contains objects x1, x2, and y1, y2. For the simplicity, we use only one read operation r.

Table 2. The states and rules mapping of a Chinese Wall policy

<table>
<thead>
<tr>
<th>states</th>
<th>rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1=(a, r, x)</td>
<td>¬s2</td>
</tr>
<tr>
<td>s2=(a, r, x2)</td>
<td>¬s1</td>
</tr>
<tr>
<td>s3=(a, r, y)</td>
<td>¬s4</td>
</tr>
<tr>
<td>s4=(a, r, y2)</td>
<td>¬s3</td>
</tr>
<tr>
<td>s5=(b, r, x)</td>
<td>¬s6</td>
</tr>
<tr>
<td>s6=(b, r, x2)</td>
<td>¬s5</td>
</tr>
<tr>
<td>s7=(b, r, y)</td>
<td>¬s8</td>
</tr>
<tr>
<td>s8=(b, r, y2)</td>
<td>¬s7</td>
</tr>
</tbody>
</table>

Table 3 shows an example of a simple Work Flow policy [16] that allows user c to read object z only after user a has read object x, and user b has written object y, and the access can only be granted on the business location l1 or l2. User b can write object y only after user a has read object x, and the access can only be granted on the business location l1.

Table 3. The states and rules mapping of a Work Flow policy

<table>
<thead>
<tr>
<th>states</th>
<th>rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1=(a, r, x)</td>
<td>No restriction</td>
</tr>
<tr>
<td>s2=(b, w, y)</td>
<td>s1 ∧ l1</td>
</tr>
<tr>
<td>s3=(c, w, z)</td>
<td>s1 ∧ s2 ∧ (l1 ∨ l2)</td>
</tr>
</tbody>
</table>

In addition to the well-known access control policies as the shown in Table 2, 3, and 4, other (known and unknown) access control policies may have different relation mapping of their states and rules elements; Assume SR is the set of all states in R. Most of the access control policies are in the case of SR ⊂ S and SR = S, i.e. all or some states in S are also in R, however, the cases of S ⊂ SR, SR ⊂ S = ∅, and SR ⊂ S = C (where C is the set of common states) represents some (or all) rules may contain state(s) that are not in S, means that the mapping states for those rules will never be granted until the states in R that are not covered in S are included in S in the future. Also, by different algorithm implemented, other type of relations of states and rules are possible, for example, instead of being the allowable states, S contains restricted states such as a popular implementation of Rule-Based Access Control that S contains the prohibited privileges. Further, The relation of the states and rules spaces can be either a one-to-many mapping if a rule can be shared by states or one-to-one, otherwise. The one-to-many relation can be transformed to one-to-one relation if allowing the rules to be duplicated in the mapping. Note that the elements in rules are finite with maximum number equal to the total number of dynamic and static attributes.

Formally, any access control policy, PO, can be described by the mapping of access control rules AC = S → R, where S is the domain of all possible states (i.e. privilege assignments) space, and R is the range of all possible rules space. S = {s1...sn}, and s1 = (u1, p11, o1) is any privilege assignment described by u1 ∈ U, a set of all subjects, p1i ∈ P, a set of all access operations, and o1i ∈ O, a set of all objects covered by the policy PO. Thus, subject u1 is allowed to access object o1i with the p1 operation under the policy PO.

A rule r1 ∈ R = {r1...rn} is a set of Boolean expressions. Each r1j is expressed by the variables in the sets of dynamic attributes S1 ⊂ S and a set of static attributes A = {a1...an}, unlike S1, A only exists in the rules space. The Boolean expression of r1j is arbitrarily connected by logic operators l ∈ L = {∧ (AND), ∨ (OR), ¬ (NOT)}. The states attribute sj in a rules are active only when the status of sj is + in the states, because obviously, a rule is realistic only when the covered states are True. Thus, when sj in the states is changed from + to -, the same sj in the rules will be treated as “don’t care” or “null”, and the Boolean operators associated with it will be ignored in evaluating the Boolean result. This means sj has not yet authorized, and should not be a decision.
factor of the authorization process. (e1) is an example of a rule $r_x$, which contains dynamic attributes $s_1$, and $s_2$, and static attributes $a_1$ and $a_2$. $r_x$ will change to (e2) if $s_2$ has not yet happened (thus nulled) when evaluated.

\[
(s_1 \land (\neg s_2)) \lor a_1 \land a_2) \quad (e1)
\]
\[
(s_1 \lor a_1) \land a_2) \quad (e2)
\]

In conclusion, the fundamental authorization process of an access control system is first check if the user’s access request is permitted under the organization’s access control policy. The operation is checking if the requested user, operation, and object are presented in the states. Second, check what rule(s) in the organization’s access control policy the request is regulated by. And the authorization of the access request is evaluating the Boolean result in the rules match the request state.

3. Rule validation

Based on the generic access control process as described in Section 2, any access control mechanism according to the access control policy implemented should be capable of (1) configuring the states; and (2) checking the validation of rules to make sure the rules are enforceable; Obviously, a rule is enforceable only if the rule can generate a result of grant or deny of an access request, and the calculation of the result is within finite steps of logical evaluation. Thus, the operation steps of granting an access request will first search for a matched state $s_x$ in $S$ for the request, then verify the rule $r_x$ mapped to the state $s_x$, then evaluate the decision, and update the status of $s_x$ from - to + (from not-yet-granted to granted) if it was - before the authorization. The configuration of states is either straight privilege assignment such as ACL or ACM, or the disseminated results of groups and roles assignments for policies such as role based access control (RBAC) [1]. Note that the states dissemination is usually performed when the states is initialized or rebuild, that is “off-line” of the authorization process, so, does not affect the efficiency of the “run-time” process (i.e. processing the access request), and in general, the efficiency of the access control mechanism. Hence, the enforceability validation check of a rule involves two functions: one is checking for satisfaction: if the rule generates result in respond to the values of it’s state parameters, and the other one is checking for deadlocks: if the result generates within finite steps of calculation. We define both properties as follow:

3.1. Satisfiability

We define a rule as unsatisfiable when its result will never be True no matter what the variables’ truth-values of the rule are, i.e. the mapped state $s_i$ will never happen (be True) under the rule.

Since any rule $r_x$ has an unique truth value once the truth values of its elementary constituents are known, it is a well-formed Boolean expression over a finite set of elements {$S, A$} and the set of logical operations {\land (AND), \lor (OR), \neg (NOT)}). And a well-formed Boolean expression is not Satisfiable when every possible instantiation of its variables evaluates to False. Therefore, a rule is unsatisfiable when the Boolean expression of the rule $r_1$ has no True result by any assignments of its variables, for example the rule

\[
r_1 = (s_1 \land s_2) \lor (\neg s_3 \land s_1 \land \neg s_2) \land \neg s_1 \quad (e3)
\]

will never be evaluated to be “true” no matter what the truth assignments of $s_1, s_2, s_3$ are.

3.2. Deadlock

We define a rule $r_x$ as deadlocked when it has a dependency on other rule(s), which eventually depends back on $r_x$ itself such that the mapped state $s_x$ will never happen because of the cyclic referencing.

A Boolean function can be represented efficiently using a data structure called an AND/OR graph [17]; A rule $r_x$ as a Boolean function is constructed to an AND/OR Graph by connecting all the non-terminal (has dependency of other states) $s_x$’s in $r_x$. The AND/OR Graph is an undirected graph $G = (V, A)$, where $V$ is the set of vertexes represented by the $s_x$’s in $r_x$ and $A$ is the set of links represented by the \land or \lor logical relations between $s_x$’s, with a single vertex $s_0 \in V$ have in-degree 0, for each $v \in V$ having out-degree weight $w(a) \in Z^+$ for each $a \in A$. An $s_x$ is non-terminal when there is other rule in $R$ contains $s_x$ as its Boolean variable, i.e. there is a link $a$ from the non-terminal $s_x$. $G$ can introduce dummy node such that each of the descendents of the dummy node starts a sub-graph of Boolean rule that require to be solved before it’s ancestor, for example, $s_2 \land s_3$ need to be solved before the rule $s_1 \lor (s_2 \land s_3)$ represented by the dummy node $s_d$. For checking the deadlock validation, $w(a) = 1$ (see Section 4). Note that we leave static attributes $a_x$’s for the graph construction, because there are no dependencies.
between them. We also ignored the unary operator $\neg$, since it does not invoke dependency between states. For example, some rules in the following $R$.

$$
\begin{align*}
  s_1 & \rightarrow r_1 = s_2 \lor s_3 \\
  s_2 & \rightarrow r_2 = s_4 \land s_5 \land s_6 \\
  s_3 & \rightarrow r_3 = \text{no restriction} \\
  s_4 & \rightarrow r_4 = s_2 \\
  s_5 & \rightarrow r_5 = \text{no restriction} \\
  s_6 & \rightarrow r_6 = l_1
\end{align*}
$$

(e4)

can be converted to an AND-OR Graph as Figure 2 when $s_1$ access request is made ($r_1$ is evaluated). The graph is construct by using the requested state ($s_1$) as the beginning note of the graph and links the states in the rules until the terminal condition (no restriction for access in this example) is met. Where $s_3$, $s_5$, $s_6$ are terminal nodes, and $\lor$ represent the "$\lor$" Boolean relation.

![AND-OR Graph converted from (e4).](image)

Figure 2. AND-OR Graph converted from (e4).

The satisfaction-check makes sure a rule is satisfiable, which means a rule is evaluated to a result according to its variables’ values at the give state (instead of a fixed result no matter what the values of it variables are). Even though an unsatisfiable produces unchangeable result, we consider this check necessary when a well-defined access control policy implementation is required. The deadlocks-check makes sure that a rule generates a result within a finite number of evaluation steps; rules in deadlock will cause infinite loops when the rule is evaluated.

4. Computational Complexity

Different from the states configuration, which need to be set up before the access control system is operational, the enforceability validation of rules is usually performed at run-time when an access request is processed, because rules may be valid (or invalid) before an access request but invalid (or valid) when the status of the states changes from $+$ to $-$ (or vice versa), or other rules are modified/deleted at the time when processing the next access request. For example, when processing the $s_x$ access request, rule $r_x$ may be invalid because it is unsatisfiable under the current status of states, but might be satisfiable when the status of the state $s_x$ is later changed from $+$ to $-$ (therefore, removed from $r_x$) in $r_x$. As shown in (e3), $r_1$ is satisfiable when the status of $s_1$ is changed from $+$ to $-$, and therefore becomes “don’t care” in $r_1$. Or, a rule $r_x$ may be free from a deadlock when other rule(s) is (are) later removed from $R$. As shown in (e4), $r_1$ are no longer in deadlock when $r_x$ is removed, or changed such that contains no non-terminal states by the time when processing the next $s_1$ request. So, it is reasonable to compose a rule without validation check, which will be performed at run-time when an access request triggered the authorization process.

Checking for satisfaction as we have defined it is checking if every possible Boolean values of $s_x$s and $a_x$s in the $r_x$ evaluated to False. This evaluation is a problem of Satisfiability of Boolean expression, which is NP-Complete [18]. And in a worst case, checking for satisfiability takes all the possible truth assignments of all variables in $S$ (i.e. $S_i = S$) and $A$, therefore requires $2|S|!|A|$ operations.

If each connection represents the reference from one state (node) to another in an AND-OR graph, then cycle exists when the number of connections between nodes is equal or greater to the number of nodes in a graph when traversing the graph form the root to any of its terminal note. Therefore, checking for deadlock of $r_x$ is equivalent to determining if all the costs for the solutions of the AND-OR graph $G$ constructed from $r_x$ are at most $k$, where a cost is assigned to a link (connection) between two nodes in the graph. Since our purpose is to determine the number of steps when traversing $G$, the cost is represented by the nodes in $G$ is fixed to 1 and $k = |S| - 1$. Thus a cycle therefore deadlock exists in $G$ when the cost accumulated exceeds the number of nodes in $G$ minus the root note. For example, in Fig. 5, one solution for the determination of the cost to reach the terminal notes from the root is ($s_1$, $s_2$) = 1, and the other solution ($s_1$, $s_2$, $s_4$, $s_5$, $s_6$) is extended to ($s_1$, $s_2$, $s_4$, $s_5$, $s_6$) and further extended to ($s_1$, $s_2$, $s_4$, $s_2$, $s_4$, $s_5$, $s_6$) with accumulated cost equal to 6, which exceeds $|S| - 1 = 5$, the cost of this solution indicates that there exists a deadlock in $r_1$. According to [19, 17], the AND-OR graph decision problem is in the class of NP-Complete.

A rule $r_x$ may contain the number of elements in $S$ and $A$ minus the mapping state in $S$: $|S| + |A| - 1$
variables in a worst case (maximum number of variables), and $|S|!$ expanding steps are required to construct an equivalent AND-OR graph; note there is no expansion required for the static attributes $a_x$ in $A$ as there is no dependency between them. The AND-OR graph construction steps can be explained by the fact that any switching functions of $\land$, $\lor$, and $\neg$, $r_x$ can be expanded into a canonical DNF (Disjunctive Normal Form)[17] product of $m+n-1$ variables in $S = \{s_1...s_m\}$ and $A = \{a_1...a_n\}$ except the $s_x$ state that the $r_x$ is mapped to. So, in maximum, a rule $r_x$ is finally expanded into $\bigvee_{i=1}^{k} c_i$, where the $c_i$ is a clause represented as $\bigwedge_{j=1}^{m+n-1} l_j$, the $l_j$ is literal, which is either a variable or its negation of $s_x$ or $a_x$, and $k$ is equal to $|A|$ plus all the possible combination of $S$, i.e. the permutation of $|S|! - 1$. In other words, the final expanded $r_x$ may contain $(|S|! - 1)/|A|$ clauses after $(|S|! - 1)/|A|$ states plus $|A|$ steps, where the $c_i$ is a literal of $S$ or $A$. And both functions are in the complexity of NP-Complete.

As stated previously, 1, we are considering the worst cast of enforceability validation of a generic access control mechanism or algorithms, by generic, we mean that mechanism or algorithms can be implemented to cover all known and unknown access control policies. 2, we also mentioned that to be truly dynamic (allow access control policy to be changed any time), the validation check has to be performed at run-time, i.e. processed at the time when access request needs to be authorized. 3, it is obvious that states in the states space and rules in rules spaces are interrelated when historical and workflow type of policies are implemented. However, most of the popular and practical access control mechanism/algorithm in use are not geared for meeting the above three requirements, therefore, are not bond to the complexity of our discoveries.

5. Conclusion

Diversity and flexibility of access control is becoming essential with the growth of global and distributed computing environments such as the Internet and Grid computing networks. Many research efforts have focused on the solution for the mechanisms of access control policy composition and combination, but have not studied the issue of efficiency such as lower complexity bounds in their efforts. In this paper we have shown that the problem of enforcing an arbitrary access control policy, regardless of the kind of mechanism, is NP-Complete. The computational complexity is critical if the access control policy is historical or workflow related, however, for most of the popular and practical access control systems today are not related to historical or workflow types of policies, therefore does not require enforceability validation of the rules at run-time, so the invalid check can be preformed “off-line” of the authentication process. Further, the enforceability validation only required when creating or changing a state or rule and they occurs less frequently than a users’ access request, if the validation has been done before.

6. References

[12], Bertino E., Catanis B., Ferrari E., Perlasca P., “A Logical Framework for Reasoning about Access Control


