Effective Evaluation of Relationship-Based Access Control Policy Mining

Padmavathi Iyer
University at Albany – SUNY
Albany, New York, USA
riyer2@albany.edu

Amirreza Masoumzadeh
University at Albany – SUNY
Albany, New York, USA
amasoumzadeh@albany.edu

ABSTRACT

Mining algorithms for relationship-based access control policies produce policies composed of relationship-based patterns that justify the input authorizations according to a given system graph. The correct functioning of a policy mining algorithm is typically tested based on experimental evaluations, in each of which the miner is presented with a set of authorizations and a system graph, and is expected to produce the corresponding ground truth policy. In this paper, we propose formal properties that must exist between the system graph and the ground truth policy in an evaluation test so that the miner is challenged to produce the exact ground truth policy. We show that failure to verify these properties in the experiment leads to inadequate evaluation, i.e., not truly testing whether the miner can handle the complexity of the ground truth policy. We also argue that following these properties would provide a computational advantage in the evaluations. We propose algorithms to identify and correct violations of these properties in system graphs. We also present our observations regarding these properties and their enforcement using a set of experimental studies.

CCS CONCEPTS

• **Security and privacy** → **Access control**; **Authorization**.

KEYWORDS

policy mining, relationship-based access control, evaluation, mining evaluation case, synthetic test generation

ACM Reference Format:
Padmavathi Iyer and Amirreza Masoumzadeh. 2022. Effective Evaluation of Relationship-Based Access Control Policy Mining. In *Proceedings of the 27th ACM Symposium on Access Control Models and Technologies (SACMAT) (SACMAT '22), June 8–10, 2022, New York, NY, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3532105.3535022

1 INTRODUCTION

Mining access control policies is an automated process of constructing high-level access control policies from low-level authorization information. Such a process is useful in refactoring existing policies or migrating existing policies to a system that uses a different policy model than the current one. Various mining approaches have been proposed in the literature for role-based access control (RBAC) [22, 23], attribute-based access control (ABAC) [12, 21, 25], and relationship-based access control (ReBAC) [4, 8, 9, 18]. In the context of ReBAC, the focus of this paper, a policy miner is given a set of authorizations (which entity can/cannot access another entity) and a *system graph* (graph of entities and their relationships in a system). Researchers need to evaluate and establish the correctness of their proposed policy mining algorithms. Proving the correctness of any policy mining algorithm theoretically would be extremely challenging. Therefore, the alternative approach that is taken is testing the correctness experimentally using simulated/synthetic test cases. Ideally, the set of test cases is comprehensive enough to truly test a miner. For example, test cases with various levels of policy complexity can be given to a miner.

The goal of a test case evaluation is to see whether the tested miner can mine an expected ground truth policy based on a given set of authorizations. The set of authorizations itself is produced based on the ground truth policy on a given system graph. We note that it is only fair to expect a miner to produce a *semantically equivalent* policy to the ground truth policy, i.e., a policy that produces the same authorizations that were given to it as an input. For example, if there are semantically equivalent policies to the ground truth policy, a policy miner should not be faulted for producing any of those policies. However, we also note that the semantic equivalency is also conditioned on the system graph that is given to the miner. As an extreme case example, a system graph with no edges does not produce any authorizations regardless of the ground truth policy. A miner that is presented with a ground truth policy alongside such a system graph, may mine any policy (including an empty policy that does not authorize anything) and still be correct semantically since all produce the same empty set of authorizations! Therefore, even when a range of policies is tested it is critical to ensure that those tests are each truly challenging for the miner.

In this paper, we propose that the pairing of ground truth policies and system graphs should be carefully considered for an effective experimental evaluation of ReBAC miners. We characterize the properties for such effective pairings that demand the miner to produce the exact ground truth policies, and present experimental evaluations to support our theoretical results. That is, we show that if these properties are violated in an evaluation scenario, the miner will not be truly tested to reconstruct the intended policy. Moreover, meeting these properties can be advantageous in large-scale experiments since correctness can be tested by matching the syntax of the policies rather than their semantics. We summarize our contributions (and their organization) as follows:

SACMAT '22, June 8–10, 2022, New York, NY, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9357-7/22/06…$15.00
https://doi.org/10.1145/3532105.3535022

- We provide an abstract definition of ReBAC miners, and formalize semantic and syntactic equivalence of ReBAC policies accordingly. We define the notions of *graph-based semantic equivalence* and *mining evaluation case* (MEC) which are fundamental in interpreting the miners' correctness (Section 3).
- We propose the notion of *strong MEC* as those MECs that challenge a miner to produce a syntactically equivalent policy to ground truth (Section 3). We characterize strong MECs by defining two formal properties for them: *minimality* and *maximality* (Section 4). Those properties ensure that a policy cannot, respectively, shrink and grow, without changing the authorizations relative to a system graph.
- We propose algorithms for identifying violations of minimality/maximality (Section 5) and updating the system graph to address them (Section 6). For this purpose, we consider a system graph schema in our reference model that supports restrictions to produce realistic synthetic graphs for a given application domain (Section 2).
- We conduct extensive experiments using two system graph generation methods and three miners from the literature. We report our results on the chance of generating strong MECs based on different sizes of system graphs and policy. We also report the performance of proposed algorithms in testing and correcting minimality/maximality violations (Section 7).

2 REFERENCE REBAC MODEL

The fundamental idea of ReBAC is to employ relationship information between the entities (users and resources) of a system to make access decisions. Accordingly, the system data model is based on the idea of a labelled graph, in which nodes represent entities within the system and edges represent relationships between entities, and sequences of edges within the graph are used when processing the authorization decisions. In this section, based on existing ReBAC models [11, 13–15, 24], we present a reference model for ReBAC that captures the necessary features in the context of this paper.

2.1 System Model/Schema

A system consists of a set of entities, which includes the sets of subjects and objects. The authorization information in a system is captured as a directed graph called *system graph* [13].

Definition 1 (System Graph)**.** A *system graph* is a directed graph denoted as $G = \langle V, E\rangle$ where $V = S \cup O$ is the set of nodes consisting of subjects and objects in a system, L is the set of edge labels and $E \subseteq V \times V \times L$ is the set of edges (relationships) labelled from L.

In a typical system, subjects indicate users and objects indicate protected resources. We employ dot notation to indicate an element within a concept (e.g., $G.V$ refers to the nodes in system graph).

In order to form meaningful system graphs, we need to follow some restrictions based on the knowledge in a particular domain. To this end, we define a *graph schema* to constrain the types of nodes and edges that can be specified in a system graph. We adopt ideas from the *OWL Web Ontology Language* [2] to describe restrictions on the relationships between various types of nodes in the schema. These restrictions are sufficient for describing the schema in the case study in this paper. Our proposed approaches for effective evaluation are applicable irrespective of the expressiveness of a schema, and so it is straightforward to extend our approach to work with a more expressive schema.

Definition 2 (Graph Schema)**.** We define *graph schema* as a tuple $\mathcal{G} = \langle \mathcal{V}, \Gamma\rangle$, where $\mathcal{V}$ indicates the set of node types and Γ indicates the restrictions on the graph edges.

We employ the notation $\tau(v)$ to denote the type of a node v. If a node v does not have a type, then we indicate that as $\tau(v)$=*null*. A restriction specifies what kinds of and how many edges can exist from a given node type to other node types defined in the schema.

Definition 3 (Graph Restrictions)**.** A *graph restriction*, or simply a *restriction*, is defined for a node type $\tau(v)$ and represented as a triple $\langle \texttt{label}, \texttt{restr\_type}, \texttt{target\_node\_type}\rangle$, where $\texttt{label} \in L$, $\texttt{restr\_type} \in \{\texttt{EXACTLY\_ONE}, \texttt{ONLY}\}$, $\texttt{target\_node\_type} \in \mathcal{V}$, and $\texttt{target\_node\_type} \neq \tau(v)$. We denote the set of graph restrictions as $\Gamma = \{\langle \tau(v_i), \langle \texttt{label}, \texttt{restr\_type}, \texttt{target\_node\_type}\rangle\rangle\}$.

Specifically, in this paper, we define two types of restrictions as:

Definition 4 (`ONLY` Restriction Type)**.** The graph restriction specified as $\langle \texttt{label}, \texttt{ONLY}, \texttt{target\_node\_type}\rangle$, on node type $\tau(v_i)$, requires that for all nodes v_i, if there exists an edge $\langle v_i, v_j, \texttt{label}\rangle$ in the graph, then it must hold that $\tau(v_j) = \texttt{target\_node\_type}$.

Definition 5 (`EXACTLY_ONE` Restriction Type)**.** The graph restriction, $\langle \texttt{label}, \texttt{EXACTLY\_ONE}, \texttt{target\_node\_type}\rangle$, on node type $\tau(v_i)$ requires that for all nodes v_i, there must exist exactly one edge $\langle v_i, v_j, \texttt{label}\rangle$ in the graph such that $\tau(v_j) = \texttt{target\_node\_type}$.

For instance, consider the following set of restrictions based on an excerpt of the graph schema from the project management case study that we use for our experiments:

$$\begin{aligned}\Gamma = \{\ &\langle \texttt{Manager}, \langle \texttt{projects}, \texttt{ONLY}, \texttt{Project}\rangle\rangle,\\ &\langle \texttt{Manager}, \langle \texttt{department}, \texttt{EXACTLY\_ONE}, \texttt{Department}\rangle\rangle,\\ &\langle \texttt{Budget}, \langle \texttt{project}, \texttt{EXACTLY\_ONE}, \texttt{Project}\rangle\rangle,\\ &\langle \texttt{Project}, \langle \texttt{department}, \texttt{EXACTLY\_ONE}, \texttt{Department}\rangle\rangle\}\end{aligned}$$

Here, `Manager`, `Budget`, `Project`, and `Department` are some node types defined in the graph schema of the application. In the above example, the `ONLY` restriction specifies that managers can *only* be associated to projects through the `projects` relationship. Note that this restriction does not require a `Manager` node to have a `projects` relation; if it does have one or more, they must all be associated to `Project` nodes. On the other hand, the `EXACTLY_ONE` restriction on a `Manager` node specifies that a manager must be associated with *exactly one* department. Similarly, the `EXACTLY_ONE` restrictions on the `Budget` and the `Project` node types indicate that a budget must be associated with exactly one project, and a project must be associated with exactly one department.

Definition 6 (Graph Well-Formedness)**.** Given schema $\mathcal{G} = \langle \mathcal{V}, \Gamma\rangle$, we say a system graph $G = \langle V, E\rangle$ is *well-formed* if for each node $v \in V$, it holds that $\tau(v) \in \mathcal{V}$, and for every edge $\langle v, v', l\rangle \in E$, it holds that either $\langle \tau(v), \langle l, \texttt{ONLY}, \tau(v')\rangle\rangle \in \Gamma$, or $\langle \tau(v), \langle l, \texttt{EXACTLY\_ONE}, \tau(v')\rangle\rangle \in \Gamma$ *and* there is no $v'' \in V$ such that $\tau(v'')=\tau(v')$ and $\langle v, v'', l\rangle \in E$.

In the rest of the paper, well-formedness is implied when we discuss system graphs.

2.2 ReBAC Policy Model

ReBAC policy rules use the relationship information in system graph for making access decisions. A ReBAC authorization policy grants accesses based on *relationship patterns* that specify different arrangements of labeled edges between entities in a system graph.

Definition 7 (Relationship Pattern). A *relationship pattern* ϕ is a sequence of relationship labels $[l_1, l_2, \ldots, l_n]$, where $l_i \in L$ and $1 \leq n \leq N$. Here, N denotes the maximum allowable length of a relationship pattern that is determined by the target application. We denote the domain of relationship patterns by $\mathcal{R}$.

We use the notation $-l$ to represent an edge with label $l \in L$ traversed in the inverse direction. In this paper, we assume that all authorization rules are about granting the same right (or action). So, to help simplify our discussions, we use relationship patterns and authorization rules interchangeably. Accordingly, we define a ReBAC policy as follows:

Definition 8 (ReBAC Policy). A *ReBAC policy* $\rho = \{\phi_i\}$ consists of a set of ReBAC authorization rules, i.e., relationship patterns, ϕ_i.

We alternatively use the term *permitted pattern* to refer to an authorization rule ϕ. Also, we refer to any pattern ϕ' that is not part of authorization policy as a *non-permitted pattern*, i.e., $\phi' \in \mathcal{R} \setminus \rho$.

An *access request*, denoted by tuple $\langle s, o \rangle \in S \times O$, consists of the requesting subject s and the requested object o. An access request will be *permitted* if it matches one of the authorization rules (i.e., relationship patterns) in policy ρ, and will be *denied* otherwise. An access request $\langle s, o \rangle$ *matches* a relationship pattern ϕ if and only if there is a path from s to o in G such that the sequence of edge labels in the path matches the sequence of the labels in ϕ. Alternatively, we say that the pattern ϕ *applies* to the request $\langle s, o \rangle$ in such a case.

We can characterize a ReBAC policy on a given system graph by enumerating its permitted access decisions or *authorizations*.

Definition 9 (Authorizations). We define a function λ that takes system graph G and ReBAC policy ρ as inputs and produces the corresponding complete set of authorizations, denoted by $\mathcal{A} = \lambda(G, \rho)$. Formally, $\forall s, o \in G.V$, $\langle s, o \rangle \in \mathcal{A}$ if and only if access request $\langle s, o \rangle$ is permitted by ρ.

3 REBAC MINING AND MINING EVALUATION CASES (MECS)

A ReBAC mining algorithm mines a ReBAC policy based on a given set of authorizations and a system graph. We formally define a *ReBAC miner* as follows:

Definition 10 (ReBAC Miner). A *ReBAC miner* is a function μ that takes system graph G and authorizations $\mathcal{A}$ as inputs and produces a ReBAC policy ρ as the mining output. We denote this by $\rho = \mu(G, \mathcal{A})$. A ReBAC miner assumes that authorizations $\mathcal{A}$ are correct and complete with respect to graph G, i.e., $\forall s, o \in G.V$, $\langle s, o \rangle \in \mathcal{A}$ if and only if access request $\langle s, o \rangle$ is supposed to be permitted.

The above definition expects a ReBAC miner algorithm to be a function, and hence *deterministic*. It means that a miner must produce the same output ρ for the same inputs $\mathcal{A}$ and G. While not a hard requirement, such deterministic behavior is embedded in the design of most mining algorithms. This property also facilitates our formal discussion of miner evaluation in the rest of this section. We also emphasize that the input authorizations to a miner are required to be complete and correct. Correctness of a miner cannot be formalized and established in presence of incorrect or missing authorization information.

For evaluating the correctness and performance of ReBAC miners, we need to set the expectations of their output, i.e., ReBAC policies. In particular, we define the syntactic and semantic equivalence of ReBAC policies for the purpose of evaluation as follows.

Definition 11 (Syntactic Equivalence). Policies ρ_1 and ρ_2 are *syntactically equivalent* (or simply, *equal*), denoted by $\rho_1 = \rho_2$, if and only if they contain the exact same set of rules.

The authorizations that a policy enforces in a system can be considered as the semantics of that policy. Two policies that might not be syntactically equivalent could be semantically equivalent if they enforce the same set of authorizations.

Definition 12 (Graph-Based Semantic Equivalence). Policies ρ_1 and ρ_2 are *semantically equivalent based on system graph G*, denoted by $\rho_1 \equiv_G \rho_2$, if and only if they both produce the same authorizations when evaluated on G:

$$\rho_1 \equiv_G \rho_2 \iff \lambda(G, \rho_1) = \lambda(G, \rho_2)$$

Definition 13 (Semantic Equivalence). Policies ρ_1 and ρ_2 are *semantically equivalent*, denoted by $\rho_1 \equiv \rho_2$, if and only if they are semantically equivalent based on any system graph:

$$\rho_1 \equiv \rho_2 \iff \forall G, \rho_1 \equiv_G \rho_2$$

Note that two policies that are not semantically equivalent in general can be semantically equivalent based on specific system graphs. As an extreme example, every pair of policies are semantically equivalent based on a system graph that has no edges. Since no paths exist in such a graph, $\lambda()$ produces an empty set of authorizations regardless of the policy. Also, note that semantic equivalence does not necessarily correspond to syntactic equivalence for policies. However, that is the case in the context of the policy model used in this paper (Definition 8).

Inspired by the above definition of semantic equivalence we devise a fair expectation for the correctness of a ReBAC miner:

Definition 14 (Correctness of ReBAC Miner). Consider inputs G and $\mathcal{A}$ to ReBAC miner μ. We say that mined policy $\rho_m = \mu(G, \mathcal{A})$ is *correct* if and only if ρ_m preserves the authorizations given in $\mathcal{A}$ with respect to input system graph G, i.e., $\lambda(G, \rho_m) = \mathcal{A}$. A ReBAC miner is *correct* if it always produces correct mined policies, i.e.:

$$\forall \langle G, \mathcal{A} \rangle (\rho_m = \mu(G, \mathcal{A}) \implies \lambda(G, \rho_m) = \mathcal{A})$$

The above definition of correctness ensures that the mined policies will have no over-assignment or under-assignment of authorizations compared to the input authorizations. Deviation from this requirement results in policies that are not capturing the exact authorizations as expected, and hence, incorrect policies. However, we note that proving the above correctness property for any proposed miner is very challenging.

In the absence of a proof of correctness for a miner, the next best option is to evaluate its correctness experimentally. The correctness of a ReBAC miner can be experimentally evaluated using test cases where the miner is provided an input pair of a system graph and a set of authorizations, and is expected to produce the corresponding ground-truth policy. We call such tests *mining evaluation cases.*

Definition 15 (Mining Evaluation Case (MEC))**.** We capture a *mining evaluation case (MEC)* as a tuple $\langle G, \rho_T \rangle$ where G and ρ_T are a system graph and a ground truth policy, respectively. Set of authorizations $\mathscr{A}$ corresponding to MEC $\langle G, \rho_T \rangle$ can be generated using the λ function, i.e., $\mathscr{A} = \lambda(G,\rho_T)$. Miner μ passes MEC $\langle G, \rho_T \rangle$ if it produces a correct output for input $\langle G, \mathscr{A} \rangle$, i.e., after mining policy $\rho_m = \mu(G, \mathscr{A})$, we must have $\lambda(G, \rho_m) = \mathscr{A}$. Alternatively, we can express passing MEC $\langle G, \rho_T \rangle$ as a graph-based semantic equivalence: $\mu(G, \lambda(G, \rho_T)) \equiv_G \rho_T$.

Therefore, an MEC can be used to experimentally evaluate a ReBAC miner by first, generating the corresponding authorizations, then, running the miner to produce a mined policy, and finally, comparing its semantics against those of the ground-truth policy. While experimental evaluations cannot prove the general correctness of a ReBAC miner, they provide experimental assurance by demonstrating correctness for the particular MECs that are tested.

We should highlight a major characteristic of an experimental evaluation using an MEC as indicated in Definition 15. One can only expect a correct miner to produce a semantically-equivalent policy to the ground truth policy, not a syntactically-equivalent policy. This distinction is very important when evaluating a miner. During an evaluation, one might assume that the miner is being tested based on the complexity of the ground truth policy that is provided in an MEC. However, in reality, an MEC is only challenging a miner to produce a semantically equivalent policy to the ground truth policy relative to the provided system graph.

Let us demonstrate this distinction using a small example. Consider MECs $c_1 = \langle G_1, \rho_T \rangle$ and $c_2 = \langle G_2, \rho_T \rangle$ where ground truth policy ρ_T has only two simple rules (single-edge patterns), and system graphs G_1 and G_2 have a few edges as follows:

$$\rho_T = \{[a], [b]\}$$
$$G_1.V = G_2.V = \{u, v, w\}$$
$$G_1.E = \{\langle u, v, a\rangle, \langle u, w, b\rangle\}$$
$$G_2.E = \{\langle u, v, a\rangle, \langle u, w, a\rangle, \langle u, w, b\rangle\}$$

The set of authorizations produced by both MECs is the same:

$$\mathscr{A}_1 = \lambda(G_1, \rho_T) = \{\langle u, v\rangle, \langle u, w\rangle\}$$
$$\mathscr{A}_2 = \lambda(G_2, \rho_T) = \{\langle u, v\rangle, \langle u, w\rangle\}$$

Observe that, in the case of c_1, a miner needs to mine policy $\rho_{m1} = \rho_T$ in order to produce the same authorizations. But, in the case of c_2, in addition to policy ρ_{m1}, policy $\rho_{m2} = \{[a]\}$ will also produce the expected authorization since $\lambda(G_2, \rho_{m2}) = \mathscr{A}_2$. In fact, between the two alternatives, ρ_{m2} is usually the preferred result of a mining algorithm since it achieves the expected authorization using a less complex policy. Therefore, in the case of c_2, we miss to fully evaluate the capability of a miner to produce the more complex policy ρ_T.

We refer to MECs that challenge a miner to produce a syntactically equivalent policy as *strong MECs.* Conversely, *weak MECs* may not challenge a miner enough. In the above example, c_1 and c_2 are strong and weak MECs, respectively.

Definition 16 (Strong MEC)**.** MEC $c = \langle G, \rho_T \rangle$ is strong if and only if for every ReBAC policy ρ_m: $\rho_m \equiv_G \rho_T \implies \rho_m = \rho_T$.

The above definition ensures that a miner that works correctly produces ρ_m that is syntactically equivalent to the given ρ_T. In the case of a weak MEC, however, there could be multiple alternative ρ_m's semantically equivalent to ρ_T based on G. In that case, it is fair to consider a miner that produces any of the alternative ρ_m's as correctly functioning. Since the intended (ground-truth) policy can be any of the semantically-equivalent policies, it becomes unclear on what basis the miner can be fairly evaluated. Therefore, if we intend to evaluate a miner on policies with varying degrees of complexity, we can only assure it is tested on truly different inputs if the corresponding MECs are strong.

In addition to truly testing the capability of a miner, strong MECs have a computational advantage in the evaluation process compared to weak MECs. As mentioned in Definition 14, the correctness of a miner's result needs to be established based on the authorizations it produces. Let us consider MEC $c = \langle G, \rho_T \rangle$, and the corresponding input that is provided to miner μ: $\langle G, \lambda(G, \rho_T)\rangle$. Suppose the miner has mined policy ρ_m. If c is a weak MEC, in order to evaluate the correctness, we need to produce $\lambda(G, \rho_m)$ and compare it against $\lambda(G, \rho_T)$. Note that this is true for testing correctness in general. However, for a strong MEC, we simply need to establish the syntactic equivalence between ρ_m and ρ_T (i.e., check whether $\rho_m = \rho_T$). This will be arguably a much more efficient test. Therefore, large-scale evaluation of miners will be more efficient and realistic if we ensure that we evaluate them against strong MECs.

4 PROPERTIES OF STRONG MECS

As discussed in Section 3, strong MECs are advantageous for both true and efficient evaluation of miners. In this section, we propose two formal properties that characterize strong MECs:

Property 1 (Minimality of MEC)**.** Given an MEC, we say that its ground-truth policy is *minimal* with respect to its system graph iff there is no subset of the policy that is semantically equivalent to it based on the system graph. We also call such an MEC a *minimal MEC.* Formally, MEC $c = \langle G, \rho_T \rangle$ is minimal iff $\nexists \rho \subset \rho_T, \rho \equiv_G \rho_T$.

Property 2 (Maximality of MEC)**.** Given an MEC, we say that its ground-truth policy is *maximal* with respect to its system graph iff there is no superset of the policy that is semantically equivalent to it based on the system graph. We also call such an MEC a *maximal MEC.* Formally, MEC $c = \langle G, \rho_T \rangle$ is maximal iff $\nexists \rho \supset \rho_T, \rho \equiv_G \rho_T$.

It follows from the above definition that if an MEC is not maximal then the authorizations as result of some non-permitted expression(s) are included in the MEC's authorizations. More formally, for some non-permitted pattern $\phi' \notin \rho_T$, it holds that $\lambda(G, \{\phi'\}) \subseteq \lambda(G, \rho_T)$. Such patterns create ambiguity for miners as they can be inferred as valid policy rules based on MEC authorizations, while they are not included in the ground truth policy.

Next, we show that simultaneous minimality and maximality of an MEC leads it to be a strong MEC.

Theorem 1. Assume that a miner has produced ρ_m based on MEC $c = \langle G, \rho_T \rangle$, where $\rho_m \equiv_G \rho_T$. If MEC c is both minimal and maximal, then ρ_m and ρ_T must be syntactically equivalent. Formally, $\rho_m \equiv_G \rho_T \land$ (c is minimal and maximal) $\implies \rho_m = \rho_T$.

Proof. We prove this by contradiction. Suppose that mined policy ρ_m and ground truth policy ρ_T are semantically equivalent based on G but are not syntactically equivalent:

$$\lambda(G, \rho_m) = \lambda(G, \rho_T) \tag{1}$$

$$\rho_m \neq \rho_T \tag{2}$$

Based on eq. (2) and since c is minimal, we have $\rho_m \not\subset \rho_T$. Therefore, ρ_m has at least one rule that is not in ρ_T:

$$\exists r_1 \in \rho_m : r_1 \notin \rho_T$$

Since $r_1 \in \rho_m$, by definition we have $\lambda(G, \{r_1\}) \subseteq \lambda(G, \rho_m)$. Therefore, considering eq. (1), we have:

$$\lambda(G, \{r_1\}) \subseteq \lambda(G, \rho_T) \tag{3}$$

Let us compose policy $\rho'_T = \rho_T \cup \{r_1\}$. Given eq. (3), the authorizations of ρ'_T based on G can be calculated as:

$$\begin{aligned} \lambda(G, \rho'_T) &= \lambda(G, \rho_T) \cup \lambda(G, \{r_1\}) \\ &= \lambda(G, \rho_T) \end{aligned}$$

Note that we just showed that ρ'_T ($\supset \rho_T$) is semantically equivalent to ρ_T based on G, which contradicts with the maximality of c. □

5 IDENTIFYING WEAK MECS

In this section, we describe our algorithms for detecting whether a given MEC, consisting of a system graph and a ground-truth policy, satisfies the minimality and the maximality properties discussed in Section 4. The algorithms output the relationship patterns in the system graph that violate those two properties.

Identifying Minimality Violations. We define the `check-min`() function that determines the set of minimality vaiolating rules in a policy, i.e., those that do not result in distinct authorizations from the rest of the rules in the policy. Given an MEC $c = \langle G, \rho_T \rangle$, for each rule $\phi \in \rho_T$, we check whether the set of authorizations based on the policy produced after removing ϕ, i.e., $\lambda(G, \rho_T \setminus \{\phi\})$, is the same as the complete set of authorizations, i.e., $\lambda(G, \rho_T)$. If positive, that rule will be part of the violating set returned by the function. We need to loop over all the rules in the policy once, and for each rule we need to calculate the λ function for all the remaining rules. Thus, the time complexity of `check-min`() is the time complexity of constructing the λ for an MEC multiplied by a factor of $\Theta(|\rho_T|)$, where $|\rho_T|$ indicates the number of rules in the ground-truth policy.

Identifying Maximality Violations. We define the `check-max`() function that determines the set of maximality violating patterns, i.e., those non-permitted patterns that spuriously behave like an authorization rule (i.e., applies to only permitted access requests) in the ground-truth policy with respect to the system graph. Given an MEC $c = \langle G, \rho_T \rangle$, we loop over all non-permitted patterns ϕ' in the set $\mathcal{R} \setminus \rho_T$, where $\mathcal{R}$ is the domain of relationship patterns (Definition 7). Then, for each ϕ', we check whether the set of authorizations corresponding to G and the policy consisting of only ϕ', i.e., $\lambda(G, \{\phi'\})$, is a subset of the complete set of authorizations, i.e., $\lambda(G, \rho_T)$. If positive, that pattern will be part of the maximality violating set returned by the function. The time complexity for checking the maximality of an input MEC depends on the number of non-permitted patterns in the system graph. The upper bound for the number of different relationship patterns of maximum length N that can exist in $G = \langle V, E \rangle$ is $O(E^N)$. Since we loop over all the non-permitted patterns ϕ' once to calculate $\lambda(G, \{\phi'\})$, the time taken by our `check-max`() function is, thus, the time taken for calculating the λ for an MEC multiplied by a factor of $O(E^N)$.

Generating Set of Authorizations. In both of the above functions, `check-min`() and `check-max`(), we need to calculate the set of authorizations corresponding to every relationship pattern in the system graph. That is, we need to calculate $\lambda(G, \{\phi\}) = \{\langle s, o \rangle\}$ for every pattern ϕ that is present in graph G during the `check-min`() and the `check-max`() procedures. To this end, we employ a graph traversal strategy such as the breadth-first search. Specifically, for obtaining access requests $\langle s, o \rangle$ corresponding to a given relationship pattern ϕ, we systematically explore the given system graph $G = \langle V, E \rangle$ for various relationship patterns defined on the set of labels L. For every path between a subject $s \in S$ and an object $o \in O$ encountered during graph traversal, we record the mapping between the relationship pattern ϕ that matches the traversed path and the access request $\langle s, o \rangle$. For a given node, there are $O(LV)$ adjacent nodes in the system graph. Since the length of the paths during the traversal is constrained by N, the total number of paths that can be enumerated from a single node will be $O(L^N V^N)$. Therefore, considering all source nodes in the system graph $G = \langle V, E \rangle$, the time complexity for obtaining the mapping between relationship patterns and their corresponding access requests is $O(L^N V^{N+1})$.

6 UPDATING WEAK MECS TO STRONG MECS

In this section, we discuss our methodology for producing a strong MEC corresponding to a given weak MEC and graph schema. Specifically, we transform the system graph component of a weak MEC to convert it to a strong MEC while ensuring that the resulting graph is well-formed with respect to the given schema. We also refer to this process as the *correction of the violating patterns.* At a high level, our correction algorithm creates a new path corresponding to every violating pattern as a separate component in the system graph. A major challenge involved during a path insertion is ensuring that the nodes and edges on the inserted path conform with the restrictions defined in the schema. However, it can be, and usually is, the case that a node type has a set of restrictions (Definition 3), whose target node type in turn has another set of restrictions, and so on. For instance, in the example discussed in Section 2.1, the `Manager` and `Budget` node types have different kinds of restrictions defined on them whose target node type is `Project`. The `Project` node type, in turn, has a restriction defined on it with `Department` as target node type. As a result, it may seem that the enforcement of restrictions never terminates while inserting a new path. To tackle such problem, we assume that there is at least one "sink" in the graph schema that is a node type for which no restrictions are defined. Thus, the process of restriction enforcement will terminate when it reaches such a "sink" node type.

6.1 Min./Max. Violation Correction Algorithm

Algorithm 1 describes the steps of our correction process. We retrieve the set of minimality/maximality violating patterns using `check-min`() and `check-max`() functions, described in Section 5.

Algorithm 1: Correcting Min./Max. Violations

```
   Inputs: Weak MEC $c = \langle G, \rho_T \rangle$, Graph Schema $\mathcal{G} = \langle \mathcal{V}, \Gamma \rangle$
   Result: Minimal and Maximal MEC $c = \langle G, \rho_T \rangle$
1  foreach $\phi \in$ check-min($\langle G, \rho_T \rangle$) $\cup$ check-max($\langle G, \rho_T \rangle$) do
2      $new\_nodes \leftarrow []$;
       /* Constructing new path for pattern $\phi$ */
3      Create $v_s$;
4      $new\_nodes$.append($v_s$);
5      for $i = 1$ to len($\phi$) do
6          Create $v_t$;
7          $new\_nodes$.append($v_t$);
8          $G.E \leftarrow G.E \cup \{\langle v_s, v_t, \phi[i] \rangle\}$;
9          $v_s \leftarrow v_t$;
       /* Assigning types to new nodes */
10     $v_s \leftarrow new\_nodes[1]$;
11     $v_t \leftarrow new\_nodes[2]$;
12     foreach $\langle \tau(v_i), \gamma \rangle \in \Gamma$ do
13         if $\gamma$.label $= \phi[1]$ then
14             $\tau(v_s) \leftarrow \tau(v_i)$;
15             $\tau(v_t) \leftarrow \gamma$.target_node_type;
16             break;
17     $v_s \leftarrow v_t$;
18     for $i = 2$ to len($\phi$) do
19         $v_t \leftarrow new\_nodes$[i+1];
20         foreach $\gamma \in$ get-restrictions($\tau(v_s)$) do
21             if $\gamma$.label $= \phi[i]$ then
22                 $\tau(v_t) \leftarrow \gamma$.target_node_type;
23                 break;
24         if $\tau(v_t) = null$ then
25             Go to Line 10;
26         $v_s \leftarrow v_t$;
27     if $\tau(v_t) = null$ then
28         exit;
       /* Checking restrictions on new nodes */
29     for $i = 1$ to len($new\_nodes$) do
30         $v \leftarrow new\_nodes$[i];
31         foreach $\gamma \in$ get-restrictions($\tau(v)$) do
32             if $\gamma$.restr_type = EXACTLY_ONE then
33                 $V \leftarrow \{v_i \mid \langle v, v_i, \gamma.\text{label} \rangle \in G.E$ &
                     $\tau(v_i)$=$\gamma$.target_node_type };
34                 if $V = \emptyset$ then
35                     Create node $v'$ s.t.
                         $\tau(v')$=$\gamma$.target_node_type;
36                     $new\_nodes$.append($v'$);
37                     $G.E \leftarrow G.E \cup \{\langle v, v', \gamma.\text{label} \rangle\}$;
38             if $\gamma$.restr_type = ONLY then
39                 $\mathcal{V} \leftarrow \{\tau(v_i) \mid \langle v, v_i, \gamma.\text{label} \rangle \in G.E\}$;
40                 Check if $\mathcal{V} \subseteq \{\gamma$.target_node_type };
```

Creating Paths for Violating Patterns. Lines 3-9 of Algorithm 1 create a new path component (i.e., a path graph consisting of both nodes and edges) that includes the violating pattern ϕ as a separate component in the system graph. For a relationship pattern ϕ of length N, we create a path component using $N + 1$ new nodes and sequence of N edges corresponding to the pattern ϕ between them. The time complexity of this step is $\Theta(N)$, where N is the maximum allowable length of a relationship pattern.

Assigning Types to New Nodes. Lines 10-28 of Algorithm 1 describe the process for assigning types to the new nodes based on schema $\mathcal{G} = \langle \mathcal{V}, \Gamma \rangle$. Initially, for all $v \in new\_nodes$, it holds that $\tau(v) = null$. So, we loop through all restrictions Γ until we find a restriction γ whose label matches the first relationship label in ϕ. Then, based on γ, we assign the types for the first two new nodes. In subsequent iterations, the source node type is simply initialized as the target node type from the immediately previous iteration. Then, we loop through all restrictions associated with the source node type and repeat the above process to assign the type for the next node in *new_nodes* list. If at any point, we are unable to find a type for a new node, then we reset the whole process (Line 25). This means that the sequence of type assignments that we had estimated so far did not conform with the given schema. However, if there is no feasible assignment of types to the new nodes, then our algorithm terminates (Line 28). This can happen when a minimality violating pattern is not compliant with schema restrictions $\mathcal{G}.\Gamma$, in which case we will not be able to correct the violation since we cannot add paths not specified in Γ. Overall, for processing the type assignments, we need to iterate over every label in the given violating pattern ϕ, and then for each of those labels we need to identify the restriction γ containing that label as specified in the graph schema. Therefore, the total time taken will be $O(N|L|)$, where N is the maximum length of a violating pattern and $|L|$ is the total number of relationship labels in the system. We note that more sophisticated search algorithms can be used for systematic search of the node type assignments. We follow the above procedure to simplify our presentation on inserting a path component in the system graph while consulting the associated graph schema.

Enforcing Restrictions on New Nodes. Lines 29-40 of Algorithm 1 demonstrate the process for checking and enforcing the restrictions given in the graph schema $\mathcal{G} = \langle \mathcal{V}, \Gamma \rangle$ associated with the type assignments determined in the previous step. For every node v in the *new_nodes* list, we determine if all the restrictions γ associated with $\tau(v)$ are enforced in system graph G. Specifically, if the restriction type in γ is EXACTLY_ONE, then we check whether there exists a node v_i whose type is γ.target_node_type and there is an edge $\langle v, v_i, \gamma.\text{label} \rangle$ in $G.E$. If there is no such node, then we create a new node v' and add an edge between v and v' according to the above specifications. Further, we append node v' to the *new_nodes* list since we need to check/enforce restrictions on this new node. Similarly, if the restriction type in γ is ONLY, then we ensure that if there is a node v_i such that edge $\langle v, v_i, \gamma.\text{label} \rangle$ exists in $G.E$, then the type of node v_i must be γ.target_node_type. The time taken by the restriction checking process is $O((N + |\mathcal{V}|^2) \times |L| \times |\mathcal{V}|)$. Looping through the set of restrictions associated with node type $\tau(v)$ takes $O(|L| \times 2 \times |\mathcal{V}|)$, based on Definition 3. The number of new nodes v' that can be created to satisfy the EXACTLY_ONE restriction type is $O(|\mathcal{V}|^2)$. As mentioned in the beginning of this section, for convergence, we consider that there is a "sink" node type in given schema $\mathcal{G}$ for which no restriction exists. Therefore, the total number of new nodes is $O(N + |\mathcal{V}|^2)$, which also considers the new nodes added while creating a new path component corresponding to the violating pattern ϕ.

Suppose the number of violating patterns be denoted as p. Aggregating the time taken by each of the individual operations in our correction algorithm, in the worst case, the total time can be bounded as $O(p \times ((N|L|) + ((N + |\mathcal{V}|^2) \times |L| \times |\mathcal{V}|)))$. We note that the number of violating patterns p is theoretically bounded as $O(E^N)$. However, in practice, this value is much smaller, which we will demonstrate through our experiments (Section 7).

6.2 Algorithm Correctness

In the following, we analyze the correctness of Algorithm 1 by showing that: (1) our technique of introducing new paths corrects MEC violations, (2) correcting one MEC violation does not lead to another MEC violation, and (3) our correction algorithm produces a system graph that is well-formed with respect to the given schema. Let ϕ' be a violating pattern in given MEC $c = \langle G, \rho_T \rangle$. Also, let G' be the final system graph after processing violation ϕ'.

Adding new paths corrects minimality/maximality violations. For correcting the violation, a new path component including ϕ' is added as a separate component to G resulting in, say, G_i. So, every new node in the added path exists only in G_i and not in G. If $v_s, v_t \in G_i.V$ are end-points of the new path, then access request $\langle v_s, v_t \rangle$ matches ϕ' only in G_i and does not match ϕ' in G. While enforcing restrictions on every node v corresponding to the new path in G_i, we always add edges of the form $\langle v, v_i, l \rangle$ to the current system graph, say, G_j such that $G_{j+1}.V \setminus G_j.V = \{v_i\}$, where $l \in L$ and G_{j+1} is the system graph resulting from adding edge $\langle v, v_i, l \rangle$. Since v_t already exists in G_i, it is impossible to have a sequence of nodes $[v_s, \dots, v_t]$ in G' due to restrictions enforcement, where G' is the final system graph. So, it holds that access request $\langle v_s, v_t \rangle$ matches only pattern ϕ' and no other pattern ϕ, where $\phi' \neq \phi$, in graph G'. Therefore, the minimality/maximality violation corresponding to ϕ' is corrected since: (1) if $\phi' \in \rho_T$, then $\lambda(G', \rho_T) \setminus \lambda(G', \rho_T \setminus \{\phi'\}) = \{\langle v_s, v_t \rangle\}$ (checking minimality violation), and (2) if $\phi' \in \mathcal{R} \setminus \rho_T$, then $\lambda(G', \{\phi'\}) \setminus \lambda(G', \rho_T) = \{\langle v_s, v_t \rangle\}$ (checking maximality violation).

Correction does not create new MEC violations. Correcting violation ϕ' in MEC $c = \langle G, \rho_T \rangle$ leads to the addition of a separate component in resulting graph G', without affecting either $G.V$ or $G.E$. As a result, the number of access requests that match a pattern in MEC $\langle G, \rho_T \rangle$ will be always less than or equal to the number of access requests that match the same pattern in MEC $\langle G', \rho_T \rangle$. Thus, if an access request $\langle s, o \rangle$ matches only rule ϕ (and no other rule in ρ_T) with respect to graph G, then $\langle s, o \rangle$ will still uniquely match ϕ with respect to graph G'. That is, if rule ϕ does not violate the minimality property in MEC $\langle G, \rho_T \rangle$, then it will not violate minimality in MEC $\langle G', \rho_T \rangle$ as well. Similar reasoning applies for not creating new maximality violations based on the observation that if $\langle s, o \rangle$ matches some non-permitted pattern ϕ in G then $\langle s, o \rangle$ will match ϕ in G' as well.

Correction produces well-formed system graph. Our correction algorithm terminates only if there is a possible assignment of types, based on restrictions Γ in schema $\mathcal{G} = \langle \mathcal{V}, \Gamma \rangle$, to the nodes in the newly added path for violation ϕ'. To show the well-formedness of graph G', we need to ensure that every edge in the newly added separate component violates neither an `ONLY` restriction nor an `EXACTLY_ONE` restriction. On the contradictory, consider that there is an edge $\langle v, v', l \rangle \in G'.E$ that does not satisfy an `ONLY` restriction. However, this is not possible, since an `ONLY`-type restriction checks the type of node v' only if the edge $\langle v, v', l \rangle$ exists in G'. Specifically, we assigned the types to the new nodes (for every new node, we insert exactly one edge corresponding to the labels in ϕ' in Lines 3-9), including v', based on the schema restrictions $\mathcal{G}.\Gamma$. Similarly, it is not possible to violate an `EXACTLY_ONE`-type restriction for any edge $\langle v, v', l \rangle \in G'.E$. This is because, along with the above argument for `ONLY` restriction, the set of nodes $\{v'\}$ satisfying the `EXACTLY_ONE` restriction on node type $\tau(v)$ will be either empty or a singleton set, since our approach always creates a new node when $\{v'\} = \emptyset$, and also adds only one edge corresponding to every new node. So, G' will be well-formed respecting given schema $\mathcal{G}$.

7 EXPERIMENTS

We conduct three types of experiments in order to empirically: (1) validate the theoretical contributions of this paper, (2) investigate the violations in MECs based on varying system graph/ground-truth policy sizes, and (3) examine the performance of the proposed identification/correction algorithms for violations. In Sections 7.2 and 7.3, we demonstrate the significance of strong MECs in effectively evaluating the performance of state-of-the-art ReBAC miners and compare with another baseline from the literature that simplifies the ground-truth policy in an MEC for evaluating miners. In Sections 7.4 and 7.5, we study the effect of varying the two inputs of an MEC on its strength, in order to have a better understanding of the nature of an MEC and its generation. Specifically, we demonstrate variation in the MEC violations caused by different sizes of the system graph and the ground-truth policy in an MEC. We note that in all, but the experiment in Section 7.5, we only modify the system graph while keeping the ground-truth policy constant during the generation of an MEC. In Sections 7.6 and 7.7, we examine the overhead caused by our algorithms given in Sections 5 and 6 for identifying the violating patterns in weak MECs and then correcting those violations to produce strong MECs. Specifically, the former demonstrates the performance of running our identification algorithm once for a given MEC, and the latter demonstrates the impact on the size of the MEC caused by our correction algorithm. We start by outlining the setup and configurations of our experiments in Section 7.1, including the different system graph generation approaches and ReBAC miners used in our experiments.

7.1 Setup

We implemented `check-min()` and `check-max()` (Section 5) and Algorithm 1 (Section 6) in Python, and our system graph schema in OWL [2] using *Protege* [1], an open-source and widely used ontology editor. All experiments are performed on a 64-bit Windows 10 machine using an Intel Core i7-7700 processor and 16 GB of RAM. In all experiments, we report the average results over 10 runs. We limit the maximum length of considered relationship patterns to 5.

Ground-Truth Policy. In order to substantiate our proposal regarding the use of strong MECs for an effective evaluation of ReBAC miners, we consider a sample policy from a project management application, which we adopted from the work by Bui et al. [8], as the ground-truth policy for our evaluation. This policy controls access by organization users to the resources associated with projects. We slightly adapted the specifications of the policy to conform with the reference policy model of this paper. An example authorization rule in our adapted policy is "`department,-department,-project`", which means *the managers of a department can access all resources associated with the projects in their department.* A second example is "`projects,-project`" which means *users can access all resources associated with the projects that they are working on.*

Graph Schema. Our graph schema for the project management application consists of various node types corresponding to the users, resources, and other logical entities in an organization. Examples of user types in this application include department managers, project leaders, employees, contractors, auditors, accountants, and planners. Similarly, the resource types include tasks, schedules, and budgets. Other logical entity types include projects, departments, and technical areas. We implemented the `ONLY`-type restriction (Definition 4) and the `EXACTLY_ONE`-type restriction (Definition 5) in our graph schema model using the *AllValuesFrom* restriction and the *Cardinality* restriction (with cardinality equal to 1) in OWL, respectively. Earlier, in Section 2.1, we showed examples of restrictions on various node types of the project management schema.

System Graph Generation. In order to demonstrate the applicability of our approach irrespective of the graph generation techniques, we implemented two approaches for generating system graphs. The first one, which we refer to as *heuristic approach*, is based on the strategy followed by Bui et al. for generating object models in their work [8]. The second approach, which we refer to as *random approach*, follows a random graph generation strategy similar to Erdos-Renyi graphs. Both of these approaches generate system graphs that are restricted based on the graph schema of the project management application. Additionally, both follow the same methodology for creating nodes of various types in the system graph based on an input parameter called *graph size parameter*. Specifically, for the node types defined in the underlying graph schema, the number of nodes that is created for each type is selected from a uniform distribution with a mean equal to the inputted graph size parameter and a standard deviation of around 0.82. However, the approaches follow different methodologies for inserting edges between the created nodes. In the case of the heuristic approach, for a particular source node and a relationship label, we randomly choose the target nodes of those edges if such an edge is allowed by the restrictions defined on the source node type. In the case of the random approach, we rely on a second input parameter called *probability of edge insertion*. For any potential edge with any relationship label that is consistent with the schema, it is inserted in the graph with the probability given by the mentioned parameter.

ReBAC Miners Studied. We experimented with three ReBAC miners from the literature. The first miner, which we refer to as `Greedy`, uses heuristic strategies to mine ReBAC policies [8]. The second miner, which we refer to as `DT`, uses decisions trees [4]. The implementation of `Greedy` and `DT` miners takes as inputs a class model, an object model, and access control rules. Along with a mined policy, these two miners also produce a "simplified ground-truth policy" for comparing with and evaluating the mined policy. Given a policy,

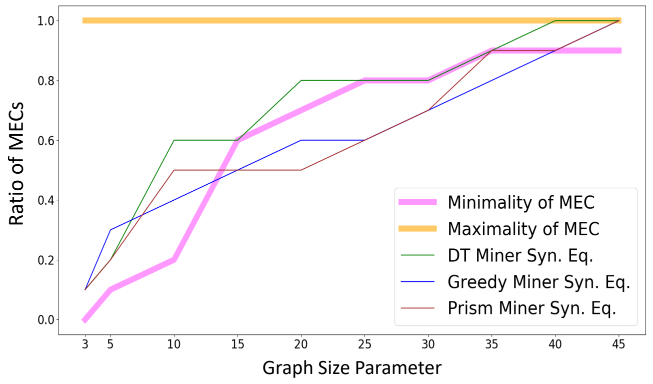


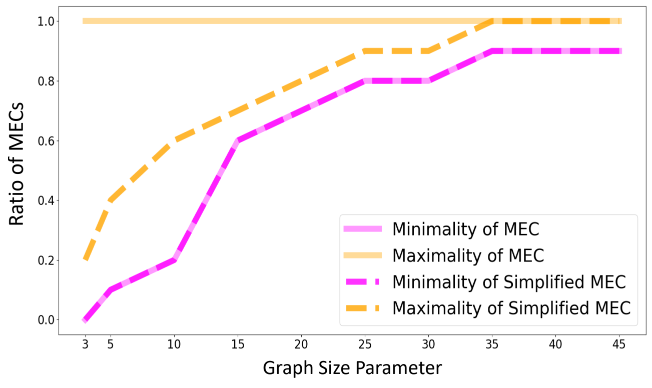
Figure 1: Impact of Strength of MEC on Performance of ReBAC Miners with respect to Graph Size Parameter for Heuristic Graph Generation Approach.

their simplification strategy aims to produce another policy that has a lower overall weight (in terms of the numbers of elements in the policy) than the original. In other words, they convert their original MEC into another MEC, which we refer to as the *simplified MEC*. These works [4, 8] report their performance metrics for `Greedy` and `DT` miners based on simplified MEC (instead of original MEC). The third miner, which we refer to as `Prism`, utilizes a combination of rule mining and pattern mining approaches to mine ReBAC authorization policies [18]. The `Prism` miner is based on a similar policy model as in this paper. There are two inputs to the implementation of `Prism` miner, namely a system graph and an access control log.

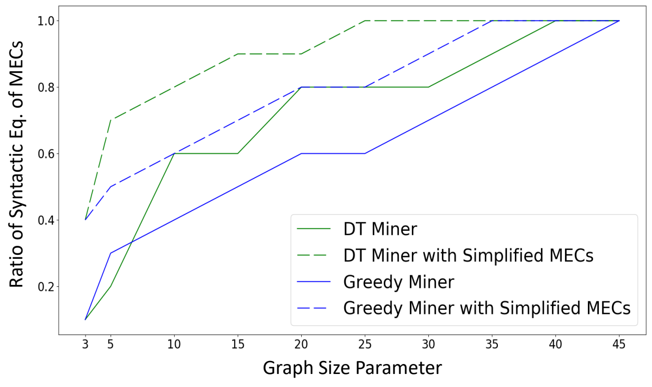
Evaluating ReBAC Miners. For `Greedy` and `DT` miners, we adapted our graph schema and our generated system graph into their class model and object model representations, respectively, and inputted those along with the ground-truth policy. For `Prism` miner, we inputted our generated system graph and the access log produced using the system graph and ground-truth policy. The miners that we experimented with are based on different ReBAC policy models, and so their mined rules have different expressive power. In order to fairly assess the policies produced by the three miners, we convert their rules into our rule format specified in Section 2.2. In particular, we convert their path expressions into relationship patterns format.

7.2 Strong MECs Challenge Miners to Produce Syntactically Equivalent Policies

We investigated the performance of the three ReBAC miners with respect to the strength of MECs. Specifically, using the ground-truth project management policy and employing the heuristic approach for generating system graphs over different graph size parameters (both discussed in Section 7.1), we recorded the semantic and syntactic equivalence of the mined policies with the ground-truth policy for given MECs. In all cases, the mined policies were semantically equivalent to the ground-truth policy. Figure 1 shows the syntactic equivalence of policies produced by the three miners and the ratio of MECs meeting the minimality/maximality properties, with respect to different sizes of the system graph (10 MECs for each size). In this experiment, we are just concerned with classifying an

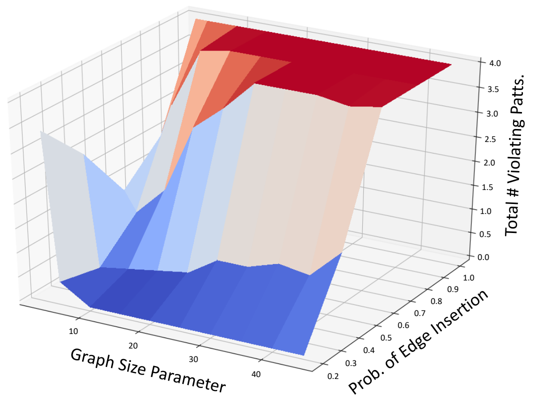


(a) Strength Comparison between MECs and Simplified MECs.

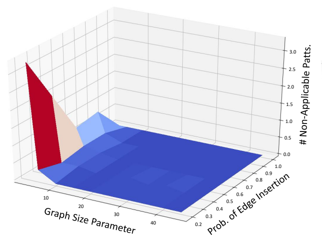


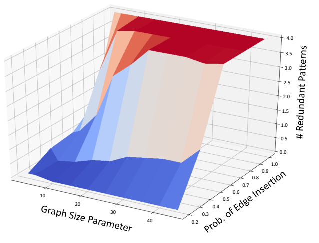
(b) Impact of Strength of MECs and Simplified MECs on Performance of `Greedy` and `DT` Miners.

Figure 2: Variations in Simplified MECs Strength and Impact with respect to Graph Size Parameter for Heuristic Graph Generation Approach.



(a) Total Number of Violating Patterns.





(b) Number of Non-Applicable Patterns Violating Minimality. **(c) Number of Redundant Patterns Violating Minimality.**

Figure 3: Variations in Strength of MECs with respect to Graph Size Parameter and Probability of Edge Insertion for Random Graph Generation Approach.

MEC as strong (when meeting both min. and max.) or weak. In the later experiments, we deal with the actual violating patterns that need to be corrected in case of weak MECs. Observe that, for all the three miners, the chances of producing a mined policy that is syntactically equivalent to the ground-truth policy increases with increasing strength of the input MEC. Therefore, strong MECs provide a systematic means for effectively evaluating a ReBAC miner. Note that the miners perform differently since they are based on different underlying algorithms (as discussed in Section 7.1). Our notion of strong MECs focuses on a fair assessment of a miner; a strong MEC does not measure a miner's performance itself.

7.3 Simplified MECs Are *Not* Always Strong

We studied the strength of simplified MECs (discussed in Section 7.1) in terms of violations of the minimality/maximality properties, where the system graph was generated according to the heuristic approach for different values of the graph size parameter. We also analyzed the impact of the strength of simplified MECs on the performance of `Greedy` and `DT` miners. Figure 2a demonstrates the chances of meeting minimality and maximality by an MEC and simplified MEC as the system graph size increases. Figure 2b demonstrates the syntactic equivalence of the mined policies with the original ground-truth policy as well as with the policy of simplified MECs, for both `Greedy` and `DT` miners. It can be observed that simplified MECs are not necessarily strong. This is evident as, along with minimality violations, maximality violations are also introduced in the case of simplified MECs. Based on our manual investigation, simplified MECs usually have at least one rule less than the original ground-truth policy, which causes an actual rule to be considered as a non-permitted pattern. Furthermore, a simplified MEC is not actually evaluating the miner against ground-truth policy, rather against a policy obtained by simplifying the ground-truth. Besides, the miners do not always produce policies that are syntactically equivalent with the policies of simplified MECs; similar to the syntactic equivalence graph when comparing with ground-truth policy (see solid lines in Figure 2b), the syntactic equivalence graph when comparing with simplified policy (see dotted lines in Figure 2b) also increases as the strength of the corresponding simplified MEC increases (see dotted lines in Figure 2a).

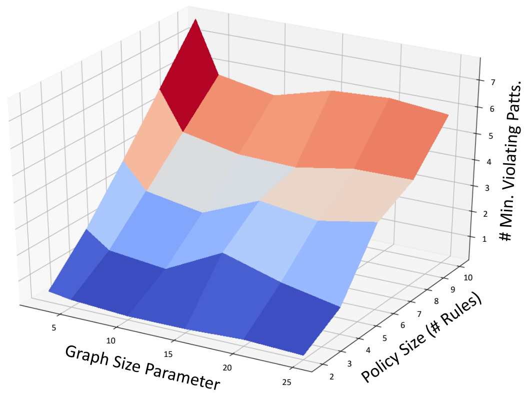
7.4 Different Types of Minimality Violations

We investigated the variation in the total number of violating patterns (returned by `check-min()`/`check-max()`) for a given MEC with respect to different sizes of the system graph. Figure 3a demonstrates the variation in the number of minimality violations with respect to the graph size parameter and the probability of edge insertion for the random approach. We did not observe any maximality violations corresponding to the inputted MECs. We can observe that the violations generally increase as the probability of edge insertions increases. They also generally grow slightly with the graph size. However, we interestingly observe relatively high violations for smaller graph sizes and lower edge probability. To be able to get a better insight into the cause of these violations, we split the total minimality violations into two: violations caused when some rules are not applicable to any access request in the system graph and violations caused when some rule's authorizations are overshadowed by another rule's authorizations in an MEC. Figures 3b and 3c demonstrate these two kinds of violations. We refer to the former as the case of *non-applicable rules*, and we refer to the latter as the case of *redundant rules*. We can observe that when the system graph size is smaller, the chances of getting non-applicable rules is much higher than that for redundant rules. However, as the system graph size increases, the redundant rules dominate over the non-applicable rules during minimality violations. We also performed the above experiment by generating the system graphs based on the heuristic approach, and observed a similar trend.

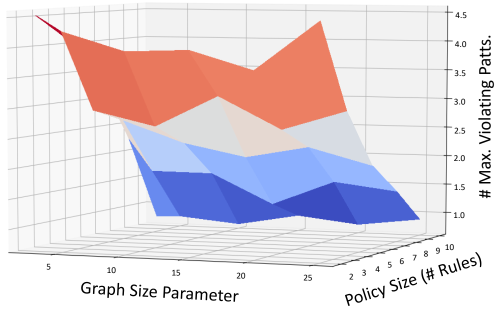
7.5 Varying Ground-Truth Policy

In all previous experiments, we generated different MECs where the ground-truth policy element was fixed and only the system graph element was changing. Here, we discuss our observations regarding the impact of varying the ground-truth policy, including its size (i.e., the number of its rules), on the corresponding MEC being strong. For every graph generated using the heuristic approach and a given graph size parameter, we generate random policies of various sizes, and record minimality/maximality violations in the resulting MECs. We repeat this procedure for different graph size parameters to investigate such behavior over various system graphs. We initially generate a set of all possible policies based on the graph schema and maximum rule length, and then randomly utilize sampled policies of various sizes as needed during the experiment. To construct a rule pattern, we add the edge labels, or their inverses, in such a sequence that does not violate any of the restrictions corresponding to the source/target node types of any label in the pattern.

Figure 4 shows the number of patterns violating minimality/maximality properties with respect to different graph size parameters and ground-truth policy sizes. Observe that the number of minimality violations (see Figure 4a) increases with increasing policy size. This is because, as the number of rules in the ground-truth policy increases, the chances of a rule being redundant (i.e., its authorizations overshadowed by another rule's authorizations) also increases, in turn increasing the chances of getting minimality violations. Also, observe that the number of maximality violations (see Figure 4b, looking at the surface from below) decreases with increasing policy size. This is because, the maximality property deals with non-permitted patterns whose authorizations are included in a given MEC's authorizations. By increasing policy size, we increase the chances of including such non-permitted patterns into the ground-truth policy, in turn reducing the chances of getting maximality violations. Based on our manual investigation of the generated policies, we speculate that policies should be somehow



(a) Variation in Number of Patterns Violating Minimality Property.



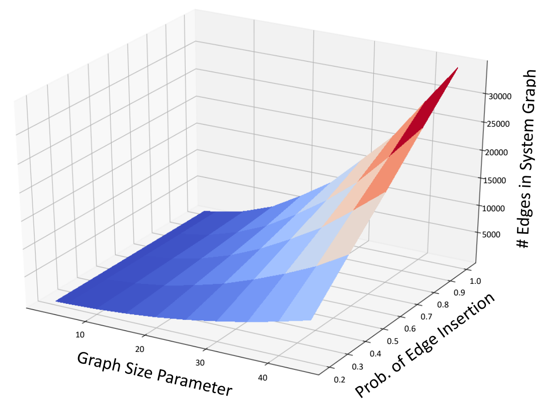
(b) Variation in Number of Patterns Violating Maximality Property.

Figure 4: Variations in Strength of MECs with respect to Graph Size Parameter and Policy Size (i.e., Number of Rules) for Heuristic Graph Generation Approach.

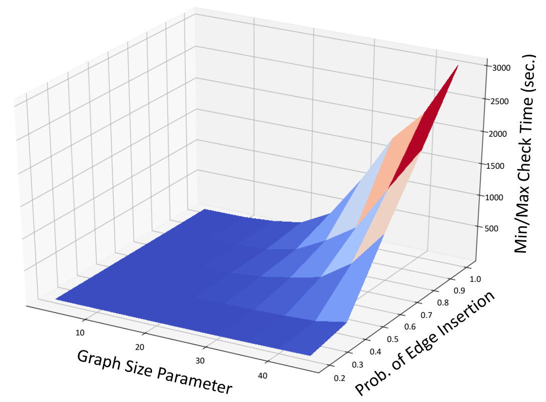
"informed" by graph schema, and it would be interesting to explore such an association for reducing violations in the generated MECs.

7.6 Increase in Min./Max. Checking Time Proportional to Number of Edges

We examined the time taken by `check-min`() and `check-max`() for identifying minimality/maximality violating patterns in a given MEC. We also studied how the number of edges varies in the system graphs corresponding to the input MECs. Figures 5a and 5b demonstrate, respectively, the number of system graph edges across different MECs and the time taken for checking the minimality/maximality of those MECs for the random graph generation approach with respect to the graph size parameter and the probability of edge insertion. We can observe that both graphs follow a similar pattern, which can be attributed to the fact that the graph traversal cost increases with the number of edges in the system graph. In our prototype implementation, graph traversal is usually the most costly phase while identifying minimality/maximality violations.



(a) Number of Edges in System Graph.



(b) Time Taken (In Seconds) for Identifying Minimality/Maximality Violating Patterns in MECs.

Figure 5: Increase in Min/Max Checking Time similar to Incr. In Number of Edges with respect to Graph Size Parameter and Probability of Edge Insertion for Random Approach.

Therefore, the time taken for identifying the minimality/maximality violations in an MEC is proportional to the number of edges in the system graph of the corresponding MEC. Again, we observed a similar trend between the minimality/maximality checking time and the number of edges in case of the heuristic graph generation approach (not presented in the paper to avoid repetition of content).

7.7 MEC Correction Grows System Graph Proportional to Number of Violations

We studied the effect of our correction algorithm, which generates a strong MEC for a given weak MEC, on the system graph size. In particular, for every graph size parameter and the probability of edge insertion in case of the random graph generation approach, we recorded the increase in the number of nodes and edges in the system graph. Such an increase is caused due to the creation of new paths corresponding to the patterns violating the minimality/maximality properties. Figure 6 demonstrates our results. We can observe that the increase in the number of nodes (and similarly for the number of edges) follows almost the same trend as the graph shown in Figure 3a. This is because, based on our Algorithm 1, we create new nodes and edges only for those patterns that violate the minimality/maximality properties. Therefore, we can infer that the increase in the system graph size is proportional to the number of violating patterns for a given MEC. Additionally, for all given MECs, executing `check-min()` and `check-max()` on the corrected MECs did not return any violations, which manifests our correction algorithm's effectiveness in producing a strong MEC for a given weak MEC. We observed that the results for the heuristic graph generation approach follow a similar pattern as in Figure 6.

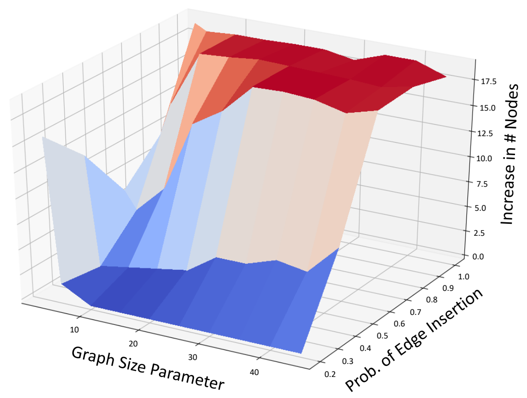


Figure 6: Increase in Number of Nodes due to Correction of Weak MECs with respect to Graph Size Parameter and Probability of Edge Insertion for Random Approach.

8 RELATED WORK

ReBAC Miners. A ReBAC miner aims to extract concise, high-level rules in terms of relationships between users and resources from given lower-level authorizations and entity relationships. Bui et al. presented two algorithms, namely a greedy algorithm guided by heuristics [7] and a grammar-based evolutionary algorithm [8]. The authors have also proposed combining neural networks and a grammar-based genetic algorithm to support additional policy language features such as set-equality and subset-equal set comparison operators [6]. More recently, they presented a simpler algorithm based on decision trees and its variant that can mine policies with negation conditions [4] and unknown attribute values [5]. Iyer and Masoumzadeh proposed a solution for mining ReBAC authorization rules in an evolving system based on rule mining and frequent graph-based pattern mining concepts [18]. Recently, researchers considered the problem of detecting the feasibility of ReBAC policy mining [9, 10]. Given the set of lower-level authorizations and the relationship graph for system entities, their approach loops through every permitted access request, identifies all possible paths between the request nodes, and deems it infeasible if the set of identified paths between the permitted request is completely satisfied by any unauthorized access request. If rule generation is feasible for every permitted request, then the mined policy is returned. The authors have also studied this problem in the context of different ReBAC policy languages, with varying expressiveness, which differ in the relationships, inverse relationships, and non-relationships used to build the policy. Recently, researchers have proposed to learn the authorization behavior of a black-box system by actively submitting access requests to and observing the corresponding decisions from the system [19]. They aim to minimize the amount of access control observations required to learn the authorization behavior.

Evaluating ReBAC Miners. For all the above works, their evaluations do not necessarily test the mining of the intended, ground-truth policies. They test their mining on different MECs, but those MECs do not follow well-defined properties for effective evaluation, which we establish in this paper. In the works by Bui et al. [4–8], the relationship data is generated by policy-specific pseudo-random algorithms that creates objects (users and resources) and selects their attribute values using appropriate probability distributions. In their works, relationships are expressed using fields that refer to other objects, and path expressions are used to follow chains of relationships between objects. The desired number of instances for the various object types is selected from a normal distribution whose mean is linear to a size parameter, which is an input of the object model generators. The values of different fields within an object are randomly chosen object(s) of the appropriate type. Due to the above procedure, there is no systematic means of determining if their generated MECs are strong. As a result, the miner may not have a fair chance of observing the policy from the given object model and lower-level authorizations; for example, if one rule overshadows another rule in the generated object model, the miner might simply ignore the latter rule. This, in turn, can cause misleading evaluation results when the mined policy is not syntactically equivalent to the ground-truth policy. To support our theory, this issue of their MEC being weak and their evaluation not testing the mining of the intended policy, is evident when the authors convert their MEC into simplified MEC whose policy is produced by simplifying the ground-truth policy. In the works by Iyer and Masoumzadeh [18, 19], the relationship data is generated pseudo-randomly by considering the information about entities and edge types, and the domain-specific constraints in place. In the works on mining feasibility detection [9, 10], determining if there exists a “mine-able” ReBAC ruleset corresponding to a weak MEC can lead to producing a policy that is different from the intended policy. Moreover, compared to their proposal, our work presents defined properties for strong MECs, which provide a more systematic understanding of the mining inputs and ensure effective evaluation.

9 CONCLUSION AND FUTURE WORK

In this paper, we consider the first-of-its-kind problem of effectively evaluating a ReBAC policy miner. In particular, we introduced the notion of a strong MEC that challenges a miner to produce a policy that is syntactically equivalent to the ground-truth policy. We propose two properties to formally characterize a strong MEC, and through theoretical and experimental results we demonstrate the significance of strong MECs for a fair assessment of a ReBAC miner's performance. Another sphere of policy mining that has been receiving great research interest deals with the problem of mining ABAC policies from lower-level authorizations and entity attributes [12, 16, 17, 20, 21, 25]. Moreover, researchers have formulated ReBAC as an “extension” of ABAC, in which the relationships are represented through the attributes of an entity that refer to other entities [3, 8]. It would be an interesting future work to explore the problem of effective evaluation in the domain of ABAC policy mining using our theory of MECs proposed for ReBAC miners.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and helpful suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 2047623.

REFERENCES

[1] 1999. Protege Ontology Editor and Framework. https://protege.stanford.edu/.
[2] 2004. OWL Web Ontology Language Guide. https://www.w3.org/TR/2004/REC-owl-guide-20040210/.
[3] Tahmina Ahmed, Ravi Sandhu, and Jaehong Park. 2017. Classifying and comparing attribute-based and relationship-based access control. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy.* 59–70.
[4] Thang Bui and Scott D Stoller. 2020. A Decision Tree Learning Approach for Mining Relationship-Based Access Control Policies. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies.* 167–178.
[5] Thang Bui and Scott D Stoller. 2020. Learning attribute-based and relationship-based access control policies with unknown values. In *International Conference on Information Systems Security.* Springer, 23–44.
[6] Thang Bui, Scott D Stoller, and Hieu Le. 2019. Efficient and Extensible Policy Mining for Relationship-Based Access Control. In *Proc. ACM SACMAT.* 161–172.
[7] Thang Bui, Scott D Stoller, and Jiajie Li. 2017. Mining relationship-based access control policies. In *Proc. ACM SACMAT.* 239–246.
[8] Thang Bui, Scott D Stoller, and Jiajie Li. 2019. Greedy and evolutionary algorithms for mining relationship-based access control policies. *Computers & Security* 80 (2019), 317–333.
[9] Shuvra Chakraborty and Ravi Sandhu. 2021. Formal analysis of rebac policy mining feasibility. In *Proc. ACM CODASPY.* 197–207.
[10] Shuvra Chakraborty and Ravi Sandhu. 2021. On Feasibility of Attribute-Aware Relationship-Based Access Control Policy Mining. In *IFIP Annual Conference on Data and Applications Security and Privacy.* Springer, 393–405.
[11] Yuan Cheng, Jaehong Park, and Ravi Sandhu. 2012. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing.* IEEE, 646–655.
[12] Carlos Cotrini, Thilo Weghorn, and David Basin. 2018. Mining ABAC rules from sparse logs. In *IEEE European Symposium on Security and Privacy.* 31–46.
[13] Jason Crampton and James Sellwood. 2014. Path conditions and principal matching: a new approach to access control. In *Proc. ACM SACMAT.* 187–198.
[14] Philip WL Fong. 2011. Relationship-based access control: protection model and policy language. In *Proc. ACM CODASPY.* 191–202.
[15] Philip WL Fong and Ida Siahaan. 2011. Relationship-based access control policies and their policy languages. In *Proc. ACM SACMAT.* 51–60.
[16] Mayank Gautam, Sadhana Jha, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. 2017. Poster: Constrained Policy Mining in Attribute Based Access Control. In *Proc. 22nd ACM SACMAT.* ACM, 121–123.
[17] Padmavathi Iyer and Amirreza Masoumzadeh. 2018. Mining positive and negative attribute-based access control policy rules. In *Proc. ACM SACMAT.* 161–172.
[18] Padmavathi Iyer and Amirreza Masoumzadeh. 2019. Generalized Mining of Relationship-Based Access Control Policies in Evolving Systems. In *Proc. 24th ACM SACMAT.* ACM, 135–140.
[19] Padmavathi Iyer and Amirreza Masoumzadeh. 2020. Active Learning of Relationship-Based Access Control Policies. In *Proc. ACM SACMAT.* 155–166.
[20] Leila Karimi and James Joshi. 2018. An unsupervised learning based approach for mining attribute based access control policies. In *2018 IEEE International Conference on Big Data (Big Data).* 1427–1436.
[21] Eric Medvet, Alberto Bartoli, Barbara Carminati, and Elena Ferrari. 2015. Evolutionary inference of attribute-based access control policies. In *International Conference on Evolutionary Multi-Criterion Optimization.* Springer, 351–365.
[22] Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. 2016. A survey of role mining. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–37.
[23] Ian Molloy, Ninghui Li, Yuan Alan Qi, Jorge Lobo, and Luke Dickens. 2010. Mining roles with noisy data. In *Proc. ACM SACMAT.* 45–54.
[24] Edelmira Pasarella and Jorge Lobo. 2017. A datalog framework for modeling relationship-based access control policies. In *Proc. ACM SACMAT.* 91–102.
[25] Zhongyuan Xu and Scott D Stoller. 2014. Mining attribute-based access control policies. *IEEE Trans. Dependable and Secure Computing* 12, 5 (2014), 533–545.