Generalized Mining of Relationship-Based Access Control Policies in Evolving Systems

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

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

ABSTRACT

Relationship-based access control (ReBAC) provides a flexible approach to specify policies based on relationships between system entities, which makes them a natural fit for many modern information systems, beyond online social networks. In this paper we are concerned with the problem of mining ReBAC policies from lower-level authorization information. Mining ReBAC policies can address transforming access control paradigms to ReBAC, reformulating existing ReBAC policies as more information becomes available, as well as inferring potentially unknown policies. Particularly, we propose a systematic algorithm for mining ReBAC authorization policies, and a first of its kind approach to mine *graph transition policies* that govern the evolution of ReBAC systems. Experimental evaluation manifests efficiency of the proposed approaches.

CCS CONCEPTS

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

KEYWORDS

relationship-based access control; policy mining; graph transition

ACM Reference Format:
Padmavathi Iyer and Amirreza Masoumzadeh. 2019. Generalized Mining of Relationship-Based Access Control Policies in Evolving Systems. In *The 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19), June 3–6, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3322431.3325419

1 INTRODUCTION

In relationship-based access control model (ReBAC) [8, 7, 10, 11, 18, 19], system authorization information is modeled as a graph comprising of entities (nodes) and relationships (edges), and policies are specified based on relationship patterns between entities. For example, a ReBAC policy to protect medical records could indicate that *Primary doctors can access their patients' medical records.*

In this paper, we are interested in mining ReBAC policies from lower-level authorization information. One of our motivations is to facilitate intact migration from legacy access control models such as access control lists (ACL) to ReBAC in a time-efficient manner. ReBAC has been shown to be advantageous in domains beyond online social networks, such as medical record systems [18], due to its flexibility and expressiveness power. However, migrating from existing models to ReBAC can be a tedious and error-prone task if performed manually. Furthermore, ReBAC mining can be employed to make existing policies more concise based on supplementary relationship information extracted from entities over time. Moreover, inferring enforced access control policy when the full specification enforced by a system is not available [14] can be achieved through mining.

Assuming that we can acquire a graph of entities in a system, the problem of mining ReBAC *authorization policies* can be seen as extracting authorization patterns (rules) from a given authorization information such as a history of successful and unsuccessful access requests. Moreover, we note that in a real-world system the graph keeps evolving as new users and resources are added or removed from the system and relationships form or disappear among them. Many modern applications such as online social networks essentially use the same system graph both for protecting their data as well as serving their application. Therefore, capturing access control policies becomes even more challenging since the permissions that were once granted may be denied in future or vice versa. As information contained in system graph is used in authorization decisions, there needs to be also policies that authorize modification of system graph itself which we call *graph transition policies.* We note that such rules have also been considered as part of recent administrative ReBAC models [19, 7]. Our main contributions in this work can be summarized as:

- Proposing an optimal algorithm for mining ReBAC authorization policy rules, avoiding heuristic procedures.
- Introducing the problem of mining graph transition policies and extending our solution to address the problem. To the best of our knowledge, this is a first of its kind approach of mining authorization rules in an evolving system.
- Demonstrating the feasibility and effectiveness of the proposed algorithms through experiments, in addition to comparing performance with a previous work [4].

At a high level, our mining algorithms for authorization policies and graph transition policies work as follows. By adopting ideas from rule mining [5] and frequent graph-based pattern mining [22], the former conducts a systematic search on a combination of a system graph and an authorization log in order to mine concise path conditions in the graph that model authorization decisions in the log. In the latter case, we can consider that each record in the authorization log is accompanied with a snapshot of the system graph used for making a grant or deny decision. Alternatively, we can model the same behavior using a system graph with timestamped edges. A pleasant consequence of this alternative model is that it enables our mining process to work on a single system graph instead of a history of graph snapshots, which in turn allows us to follow a strategy similar to our first algorithm while additionally account for consistency of mined patterns with regards to timestamps.

2 POLICY MODEL & MINING CHALLENGES

In this section, based on existing ReBAC models [11, 8, 7], we present a reference model for ReBAC that captures the necessary features in the context of this paper.

The authorization information in a system is captured as a directed graph $G(V, E)$, called *system graph* [8], where V is the set of system entities and $E \subseteq V \times V \times R$ represents relationships among them. Each edge in the graph is labelled by a relationship in R.

2.1 Authorization Policies

A ReBAC authorization policy consists of a set of authorization rules. An authorization rule is of the form $\langle \phi, acts \subseteq ACT \rangle$ where ϕ is a condition composed itself of a set of *relationship patterns*. A *relationship pattern* is defined as a sequence of relationship types that can be used to characterize patterns of paths in a system graph. Access request applicable to these policies is of the form $\langle s, o, act \rangle$ where $s \in V$ requests performing action $act \in ACT$ on $o \in V$. Here, ACT is a set of all abstract actions depending on the application domain. An access request matches an authorization rule if it satisfies all relationship patterns in the rule condition. Condition ϕ in authorization rule is formally defined using the following grammar:

$$\phi ::= relpattern\ [;\ relpattern]$$
$$relpattern ::= [-]r\ [,\ relpattern]$$

Here, $-r$ represents an edge of type $r \in R$ traversed in inverse direction. For example, edge $\langle Bob, Alice, \textit{has-primary-doctor} \rangle$ in a health-care system (Alice is Bob's primary doctor) can also be represented as $\langle Alice, Bob, -\textit{has-primary-doctor} \rangle$. Consider a sample authorization rule that allows primary doctors to read their patients' medical records: $\langle$"$-$ *has-primary-doctor*, $-$ *has-owner*", $\{read\}\rangle$. The following relationship instance match the condition in the rule, and therefore, allows Alice to read Bob's record:

$$Alice \xrightarrow{-has\text{-}primary\text{-}doctor} Bob \xrightarrow{-has\text{-}owner} Bob\_Rec$$

An example of a rule with multiple conditions in the context of an OSN could be $\langle$*"friend, friend, owns; city, -city, owns"*, $\{read\}\rangle$ that allows access by only friends of friends living in the same city as the owner.

2.2 Graph Transition Policies

Graph transition policies govern modifications to the system graph. Therefore, the system graph is used as authorization information as well as target for action. We assume that a subject modifying the system graph is also an entity in the graph itself. A graph transition request is of the form $\langle s, x, y, r, op \rangle$ where subject $s \in V$ requests performing operation $op \in \{create, delete\}$ on edge $\langle x, y, r \rangle$. We do not consider operations on system graph vertices as they can be captured using operations on adjacent edges. A vertex is created when an edge with that vertex as one of its end-point is created; a vertex is deleted when all its adjacent edges are deleted.

A graph transition rule needs to specify permissions on an affected edge. It is formally represented by a tuple $\langle \phi_s, \phi_t, r \in R, ops \subseteq \{create, delete\} \rangle$. Here, ϕ_s and ϕ_t represent the conditions that need to be satisfied between the subject and the source (beginning vertex) and the target (ending vertex) of the affected edge, respectively. They follow the same format defined in § 2.1. r and ops indicate the relationship type of affected edge and the permitted operations, respectively.

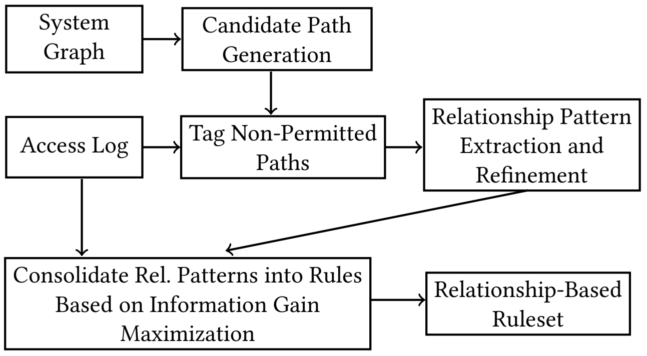
System Graph
Candidate Path Generation
Access Log
Tag Non-Permitted Paths
Relationship Pattern Extraction and Refinement
Consolidate Rel. Patterns into Rules Based on Information Gain Maximization
Relationship-Based Ruleset

Figure 1: High level flow of ReBAC mining algorithms for authorization and graph transition policies

2.3 Challenges

Given an access log and a system graph, the procedure of mining ReBAC policies is a non-trivial task. Suppose a certain user is authorized to perform an action on a certain object as per the access log. There can exist multiple paths between the user and the object in the system graph that could have resulted in that authorization. The challenge is to find the most concise combination of path conditions that holds for all similar accesses in the log.

In the case of actions causing system graph transition, the search for candidate paths needs to be performed across multiple temporal snapshots of the system graph. Furthermore, along with multiple paths between the subject and end-points of the modified edge, we also need to take the relative timestamps of edges into consideration. For instance, for mining a rule that states *users can tag friends in their posts* in an OSN, the timestamp of *tag* relationship should be greater than that of the *owner* and *friend* relationships for all matching instances.

3 ALGORITHM

Figure 1 shows an overview of the steps taken by our authorization and graph transition policy mining algorithms. In this section, we describe the details of each of the steps.

3.1 Mining Authorization Policies

3.1.1 Inputs. There are two inputs to our ReBAC mining algorithm, namely system graph data and access control log. If $node1$ and $node2$ are end-points of an edge of type *edge-label*, then an entry in the input graph data can be represented as $\langle$*node1*, *node2*, *edge-label*$\rangle$.

Algorithm 1: mineReBACPolicy

```
Input  : L (access log), G (system graph)
Output : ReBAC ruleset
1  foreach node ∈ G.V do
2      paths ← getCandidatePaths(node, L, G)
3  $\mathcal{R}$ ← group paths based on relationship pattern
4  Remove patterns containing only deny paths from $\mathcal{R}$
5  while PERMIT ∈ L do
6      sublog ← log; rule.$\phi$ ← empty
7      while ¬(sublog contains only PERMIT) do
8          pr, siz, cov ← calculate conditional probability, size
             and coverage for each rel_pattern ∈ $\mathcal{R}$
9          best ← relationship pattern with lexicographically
             maximal value of the triple ⟨pr,siz,cov⟩
10         Append best to rule.$\phi$
11         sublog ← sublog subset containing only best instances
12     Append rule to ruleset; L ← L \ sublog
13 return ruleset
```

Algorithm 2: getCandidatePaths

```
Input  : source (source node), L (access log), G (system graph),
         T [optional] (timestamp), EL [optional] (edge label)
Output : set of paths from source
1  permit_log ← Get all permit records of log
2  if EL not defined then
3      paths ← Get all paths in G from source
4  else
5      paths ← Get those paths in G from source s.t. every edge
         on the path has timestamp value less than T
6  foreach path ∈ paths do
7      if EL not defined then
8          if path end-points not in permit_log then
9              Mark path as deny
10     else
11         if ⟨path end-points, EL⟩ not in permit_log then
12             Mark path as deny
13 return paths
```

The access control log enumerates all possible access requests in the graph. Each entry in our input log file can be represented as a tuple of the form ⟨*requesting-user*, *requested-resource*, *decision*⟩.

3.1.2 Consolidate Patterns into Rules. Algorithm 1 works in an iterative manner as follows. For each candidate relationship pattern, the conditional probability for PERMIT given that candidate pattern is calculated, followed by measuring the candidate's coverage in given log and the size of the candidate (Line 8). Then the *best* candidate pattern is chosen according to the lexicographical ordering ⟨*conditional-probability*, *size*, *coverage*⟩ (Line 9). Subsequently, we append the best pattern computed above to *rule*'s condition set, and filter the log based on log records satisfied by this candidate relationship pattern. The above steps are repeated until the filtered log contains records with PERMIT decisions only (Line 7). At this point, we obtained a positive authorization rule which would be a conjunction of best relationship patterns found so far, and thus this grant rule is added to ReBAC *ruleset*. Next, the filtered log is removed from the original log. In other words, all records satisfied by the rule are removed from the log. The outer while loop (Line 5) repeats this entire procedure until there are no more records in the log that have PERMIT decision i.e. all the positive authorization rules have been mined.

One of the prime functionalities in the *mineReBACPolicy* algorithm is to estimate the best candidate pattern based on the triple ⟨*conditional-probability*, *size*, *coverage*⟩. Calculation of conditional probability (probability for getting PERMIT given relationship pattern) utilizes the candidate patterns from Line 4 and iteratively estimates the information yield of each candidate about the PERMIT class. Size calculation estimates the size of each candidate pattern which equals the number of edges in that relationship pattern. Finally, coverage calculation estimates the candidate's coverage in the access log, where, the coverage of a relationship pattern in a log is the total number of log records satisfied by that pattern. A relationship pattern *satisfies* a log record if the end-points of an instance of that pattern is same as the log record. An *instance* covered by a path's relationship pattern is simply the sequence of nodes in that path.

3.1.3 Candidate Path Generation. Initially, Algorithm 2 explores paths of length 1 from the source node (Line 3), in which each "non-permitted" path is tagged with a *deny* label (Line 9). Non-permitted paths do not have their end-points in the *permit_log* (we refer to the log records having PERMIT decision as *permit_log*), or in other words the underlying system policy does not grant access to request where the requesting and requested parties are connected by the non-permitted path. Then paths of length 2 from the source node are explored, and this procedure is repeated until all possible paths in the system graph have been explored from the source node.

To be able to deal with large real-world networks containing several nodes with high degree, the proposed mining algorithm is provisioned with a user-specified constraint indicating the maximal pattern length in rule conditions. This is because large graphs and high degrees increases the computational complexity of the mining process, or more particularly, the possible path combinations that need to be evaluated magnifies substantially.

3.1.4 Relationship Pattern Extraction and Refinement. After generating candidate paths, the next step is to group the set of all paths according to relationship patterns (Algorithm 1 Line 3). In the following example, node sequences ⟨*Alice*, *Bob*, *Bob_Rec*⟩ and ⟨*Tom*, *Carol*, *Carol_Rec*⟩ follow the same pattern, so are grouped into ⟨*−has-primary-doctor*, *−has-owner*⟩:

$$Alice \xrightarrow{-has\text{-}primary\text{-}doctor} Bob \xrightarrow{-has\text{-}owner} Bob\_Rec$$

$$Tom \xrightarrow{-has\text{-}primary\text{-}doctor} Carol \xrightarrow{-has\text{-}owner} Carol\_Rec$$

Finally, we prune the grouped paths so that it contains only those patterns that were permitted at least once according to the access log (Algorithm 1 Line 4). Particularly, for each relationship pattern in grouped paths we check if the end-points of any instance following that pattern is in the *permit_log*, with the help of *deny* tag. If none of the instances, satisfying the pattern, has end-points

```
Algorithm 3: mineGraphTransitionRules
Input  : $L$ (timed access log), $G$ (initial system graph)
Output : ReBAC ruleset
1  foreach newly inserted edge new_E in $L$ do
2      Set timestamp of new_E to incremented time
3      $G \leftarrow G \cup new\_E$
4  foreach unique edge_label in newly inserted edges do
5      edges ← Get edges with label edge_label from $G$
6      foreach $edge \in edges$ with end-points $node1$ and $node2$ do
7          if $node1.timestamp < edge.timestamp$ then
8              $paths \leftarrow getCandidatePaths(node1, L, G,$
                 $edge.timestamp, edge\_label)$
9          if $node2.timestamp < edge.timestamp$ then
10             $paths \leftarrow getCandidatePaths(node2, L, G,$
                 $edge.timestamp, edge\_label)$
11     Produce key-value pairs of
         ⟨relationship pattern, instances covered⟩
12     Generate rules and append ⟨rule, edge_label⟩ to ruleset
13 return ruleset
```

in the *permit_log* i.e. all the candidate paths associated with that pattern are marked *deny*, then it simply implies that all access requests between nodes connected by that pattern have been denied, thus removing the relationship pattern from the candidates. The resultant patterns are stored as key-value pairs that basically reflect the coverage of each relationship pattern in the given access log.

3.2 Mining Graph Transition Policies

3.2.1 Overview. For the purpose of simplicity of presentation, we assume that edges are always inserted into the graph. Therefore, if $G_1, G_2, ..., G_T$ represent graph snapshots at consecutive time instances, then $G_1 \subseteq G_2 \subseteq G_3 \subseteq .... \subseteq G_T$. Extending the proposed algorithm for mining rules that authorize deletion of edges is part of future work.

Each edge is annotated with a creation time to keep track of edges present in the system at any time. Because a node is always associated with an edge, the timestamp of a node will implicitly be the timestamp of that edge of the node that appeared first in the system graph.

3.2.2 Inputs. There are two inputs to the ReBAC graph transition policy mining algorithm, access log and initial system graph. A log record is ⟨*modifier*, *node1*, *node2*, *edge_label*, *action*, *decision*⟩ where, *modifier* is the user requesting to perform (*action*) on edge, with label *edge_label*, connecting the nodes *node*1 and *node*2.

Each record in the input initial graph specifies an edge, whose format can be denoted as ⟨*node*1, *node*2, *edge_label*⟩ where the nodes *node*1 and *node*2 are related by the edge having label *edge_label*.

3.2.3 Create Timestamped Graph. In the beginning, Algorithm 3 (Line 1) sets the timestamp of all nodes and edges in the given preliminary graph to zero assuming the graph denotes inception state of the system under observation. Then each successful attempt to insert an edge in the system, indicated by PERMIT decision for the corresponding access log record, happens in one time instance. So each new authorized edge is supplemented to the current graph structure, assigning incremented time to the edge and its associated nodes, if necessary (that is if any of the edge's nodes is newly added). An agreeable outcome of this strategy is that our mining approach needs to process just a single graph instead of a chronological sequence of graph snapshots, thereby significantly optimizing the space usage.

3.2.4 Candidate Path Generation and Mining. For each distinct modified edge label, we check if the nodes connected by that edge already existed in the system (referred as *existing nodes*) or were they added as a result of edge insert operation (new nodes). This is because the user responsible for inserting an edge must be "related" to the existing node(s) of that modified edge. Based on this interesting observation, our algorithm generates candidate paths from the existing nodes associated with the edge (over all modified edge labels) (Line 6).

While exploring an edge, we need to account for edge timestamps (Algorithm 2 Line 5). Particularly, during the graph traversal only those edges already existing in the system should be considered. Furthermore, while pruning the set of candidate relationship patterns, along with the end-points of node sequences, we also need to check if the label of modified edge is in *permit_log*, since the interest in graph transitions is to understand the authorizations enforced on network edges.

The candidate paths are grouped according to relationship patterns and those having only *deny*-marked paths are eliminated from the candidate key-value pairs. Subsequently, rules are produced by following the process described in Algorithm 1 (Line 5), appending the label of modified edges, along with rules, into the ReBAC ruleset.

4 EVALUATION

The general flow for all of our experiments is as follows. We begin with an original ReBAC policy (ground-truth), create a sample system graph based on the policy, generate access log by employing the system graph (or initial system graph for graph transition policies) and the original policy, mine ReBAC rules, and finally compare the measurements. We executed all our experiments on a 64-bit Windows 10 machine with Intel Core i7-6700HQ processor and 12 GB of RAM. All experiments were performed 10 times each and we report the average time measurement in each experiment.

4.1 Setup and Configurations

We assess the performance of our algorithms on following policies that are inspired by real-world applications. EHR deals with authorizing medical staff to electronic health records [10]. eWorkForce adopts ReBAC policies in an electronic workforce management system that regulates access to work-orders, appointments and other services [9]. We also consider graph transition rules for inserting various edges such as work-orders, appointments, tasks, etc. Project controls access of organization employees such as project leaders and contractors to resources such as task, schedule and budget [4]. We slightly modified the specifications of this dataset to confirm with the policy model of this paper.

Based on the rules given in datasets, we create the (initial) system graph pseudo-randomly using information about entities and edge

Table 1: Performance Evaluation of Proposed ReBAC Mining Algorithms

Mining Alg.	Policy	$\|V\|$	$\|E\|$	$\|R\|$	$\|L\|$	$\|\rho_{orig}\|$	$\|\rho_{mined}\|$	WSC_{orig}	WSC_{mined}	*SemSim*	Time (s)
Authorization	`EHR`	36	52	8	180	8	8	22	22	1	2
[4]						8	8	22	22	1	28
Authorization	`eWorkForce`	126	197	29	10660	22	22	57	57	1	86
[4]						22	23	57	50	0.9	240
Authorization	`Project`	55	82	7	638	11	11	48	48	1	13
[4]						11	11	48	48	1	32
Graph Transition	`eWorkForce`	64	80	22	824	6	6	15	15	1	17

types in the system. In Table 1, $|V|$, $|E|$ and $|R|$ are as defined in § 2, and $|L|$ is size of access log. For access log generation, in the case of authorization policies, we create all possible access requests from the set of users (U) and resources (R) in the system, and determine if the decision is permit or deny associated with each request, based on system graph and access control policy. For graph transition policy, we create requests by randomly selecting users and resources in the system and when an access request is successful, the requested edge is inserted into the system graph. Since the authorization graph is persistently evolving, we consult the most recent snapshot for making authorization decisions.

Evaluation Metrics. To assess the correctness of proposed algorithms, we use the notion of *Semantic Similarity* (*SemSim*) which is the fraction of granted permissions shared between original and mined rulesets. Further, we evaluate the quality of mined policy based on *Weighted Structural Complexity (WSC)* [17, 4], that measures conciseness of the policy. Conciseness of a ReBAC rule can be modeled as a weighted sum of the lengths of its conditions and authorized actions:

$$WSC(rule) = w_1 \sum_{relpatt \in rule.\phi} |relpatt| + w_2|acts|$$

where $|s|$ denotes the length of set s and w_is are user-specified weights. In our experiments, all w_is are assumed to be 1. WSC of a ReBAC policy is the sum of WSC of each of the contained rules in the policy.

4.2 Results

Table 1 summarizes important observations from the experiments. *Mining Alg.* indicates whether it is the proposed algorithm (authorization polices), or previous work's algorithm [4]. $|\rho_{orig}|$, $|\rho_{mined}|$, WSC_{orig} and WSC_{mined} identify the number of rules and WSC measure of original and mined rulesets. *Time* gives the running time in seconds for each demonstration.

As shown in the table, for all the datasets and the proposed algorithms, the size and WSC of mined ReBAC ruleset is the same as that of the original ruleset, inferring that the mined policy is at least as concise as the original policy. We manually compared the mined and original ruleset and noticed that they are syntactically as well as semantically equivalent. Consequently, no rule is overfitted to just few instances, and hence our algorithm tries to find general patterns of authorizations, reducing the overall complexity of the mined policy. Furthermore, the running time is of the order of a few seconds, which is reasonable for the application domains considered in the experiments.

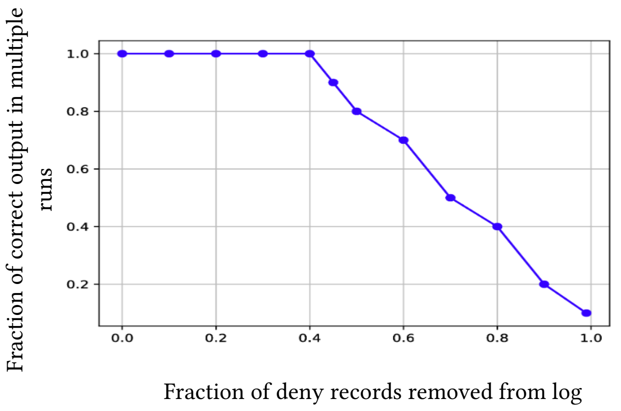

Figure 2: Robustness of mining graph transition rules versus different sampling rates of `DENY` records from the log

4.2.1 *Authorization Policies.*

Performance comparison between our mining algorithm and [4] yields the following observations. For smaller datasets like `EHR` and `Project`, both the algorithms perform similar when concerned with the quality of mined rules, that is, both yield mined policies that have the same ruleset size and WSC as the original policies. However, for larger datasets like `eWorkForce`, previous work does not output correct ReBAC ruleset, as manifested by a semantic similarity lower than 1. Moreover, there is a significant difference in the running time of both algorithms for all the datasets. As depicted in the table, our algorithm takes less than half the time taken by [4] to extract ReBAC policies.

4.2.2 *Graph Transition Policies and Case of Missing Log Records.*

In real-world systems, complete access logs might not be available, for example, if certain access requests have not been exercised yet. To this end, we run multiple experiments where we obtain a random sample of `DENY` records from the access log, while not altering the `PERMIT` records since they are used to determine the authorization state of system. We estimate how correct the mined rules are (i.e. how intact the mined policy is) as the log is randomly sampled.

Figure 2 depicts that the mined policy is intact until 40% of `DENY` records are removed from the access log. As expected, after 40% the graph gradually decreases, implying the influence of `DENY` records in log on the mining process. Thus, this demonstration gives an illustration of the lower bound on the number of records to be present in the access log for our mining approach to work 100% correctly. Nevertheless, depending on the needs of an organization, if minor errors in the mining phase are tolerable, then the correctness of our algorithm's output ruleset is reasonable even when around 60% DENY records are removed from the log. This demonstrate the robustness and scope of our mining approach for real-world generic applications.

5 RELATED WORK

There is a vast literature on role mining which deals with obtaining optimal set of roles from user-permission relation [16, 17]. Subsequently, researchers have looked into transforming policies modeled as access control lists (ACLs) or role-based policies into attribute-based access control (ABAC) policies, by mining authorization rules based on attributes of users and resources [12, 20, 15]. We borrow some ideas from PRISM [5] rule mining algorithm, which was also adopted in ABAC mining [12]. However, compared to these previous policy mining problems, in this paper, we deal with extracting relational policies rather than unary predicates like attributes or roles.

Bui et al. presents a greedy solution for the ReBAC policy mining problem [4] that can be summarized as follows: iterate over the tuples of subject-permission relation (ACL); generate candidate ruleset that together cover the entire subject-permission relation and select highest-quality rules based on a rule-quality metric; finally, optimize the candidate ruleset by merging and simplifying the rules. The authors have also extended this work recently to a grammar-based evolutionary algorithm [3]. Compared to [4], we render a systematic solution that generates rules by optimizing information gain about the PERMIT authorizations. In contrast, [4] performs pairwise rule combining, while other combinations (triples, quadruples, etc.) could be more optimal. Moreover, while generating candidate rules, their approach generates all possible paths between the user and resource for every selected subject-permission tuple that increases their search space tremendously. On the contrary, we traverse the graph from all nodes in the system only once and store the candidate patterns in key-value pairs for future retrieval, thus improving performance significantly.

Our work is related to frequent subgraph mining [6, 13, 21, 22] that extracts maximal, frequently occurring patterns from a given graph database or a single graph. However, we deal with an additional constraint about matching patterns with access log, which makes mining ReBAC policies a challenging task.

Our proposed algorithm for mining graph transition rules is related to approaches to discover local structural patterns that occur frequently as the graph evolves [1, 2]. However, we account for the additional authorization data based on the access log along with considering frequently occurring patterns and relative edge timestamps. To the best of our knowledge, we are the first to propose an algorithm for mining authorization rules in an evolving system.

6 CONCLUSION

In this paper, we presented approaches for mining two categories of ReBAC policies: authorization policies and graph transition policies. Experimental evaluations show that both proposed algorithms are efficient in terms of running time and mining capability, and outperforms the previous work [4]. In future, we will extend our ReBAC mining framework for extracting relationship-based conditions that consider topological patterns beyond paths.

REFERENCES

[1] M. Berlingerio, F. Bonchi, B. Bringmann, and A. Gionis. Mining graph evolution rules. In *joint European conference on machine learning and knowledge discovery in databases*, pages 115–130. Springer, 2009.

[2] B. Bringmann, M. Berlingerio, F. Bonchi, and A. Gionis. Learning and Predicting the Evolution of Social Networks. *IEEE Intelligent Systems*, 25(4):26–35, July 2010. ISSN: 1541-1672.

[3] T. Bui, S. D. Stoller, and J. Li. Greedy and evolutionary algorithms for mining relationship-based access control policies. *Computers & Security*, 80:317–333, 2019.

[4] T. Bui, S. D. Stoller, and J. Li. Mining Relationship-Based Access Control Policies. In *Proceedings of the 22Nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '17 Abstracts, pages 239–246. ACM, 2017.

[5] J. Cendrowska. PRISM: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27(4):349–370, Oct. 1, 1987. ISSN: 0020-7373.

[6] Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):190–202, Feb. 2005. ISSN: 1041-4347.

[7] J. Crampton and J. Sellwood. ARPPM: Administration in the RPPM Model. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 219–230. ACM, 2016.

[8] J. Crampton and J. Sellwood. Path Conditions and Principal Matching: A New Approach to Access Control. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*, SACMAT '14, pages 187–198. ACM, 2014.

[9] M. Decat, J. Bogaerts, B. Lagaisse, and W. Joosen. The workforce management case study: functional analysis and access control requirements. *CW Reports, volume CW655*, 40, 2014.

[10] P. W. Fong. Relationship-based access control: protection model and policy language. In *Proc. CODASPY '11*, pages 191–202. ACM, 2011.

[11] P. W. Fong and I. Siahaan. Relationship-based access control policies and their policy languages. In *Proc. 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 51–60. ACM, 2011.

[12] P. Iyer and A. Masoumzadeh. Mining Positive and Negative Attribute-Based Access Control Policy Rules. In *Proceedings of the 23Nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '18, pages 161–172. ACM, 2018.

[13] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 345–356. SIAM, 2004.

[14] A. Masoumzadeh. Inferring unknown privacy control policies in a social networking system. In *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society*, pages 21–25. ACM, 2015.

[15] E. Medvet, A. Bartoli, B. Carminati, and E. Ferrari. Evolutionary inference of attribute-based access control policies. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 351–365. Springer, 2015.

[16] B. Mitra, S. Sural, J. Vaidya, and V. Atluri. A Survey of Role Mining. *ACM Comput. Surv.*, 48(4):50:1–50:37, Feb. 2016. ISSN: 0360-0300.

[17] I. Molloy, N. Li, Y. A. Qi, J. Lobo, and L. Dickens. Mining roles with noisy data. In *Proceedings of the 15th ACM symposium on Access control models and technologies*, pages 45–54. ACM, 2010.

[18] S. Z. R. Rizvi, P. W. Fong, J. Crampton, and J. Sellwood. Relationship-Based Access Control for an Open-Source Medical Records System. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, pages 113–124. ACM, 2015.

[19] S. D. Stoller. An Administrative Model for Relationship-Based Access Control. In *SpringerLink*. IFIP Annual Conference on Data and Applications Security and Privacy, pages 53–68. Springer, Cham, July 13, 2015.

[20] Z. Xu and S. D. Stoller. Mining Attribute-Based Access Control Policies. *IEEE Transactions on Dependable and Secure Computing*, 12(5):533–545, Sept. 2015. ISSN: 1545-5971.

[21] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining Significant Graph Patterns by Leap Search. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 433–444. ACM, 2008.

[22] X. Yan and J. Han. Gspan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* Pages 721–724. IEEE, 2002.