Efficient Conformance Testing Approach for Relationship-Based Access Control

Benjamin Michaliszyn

Department of Mathematics & Computer Science
Drury University, Springfield, MO 65802
Email: bmichaliszyn@drury.edu
(Advisor: Padmavathi Iyer, riyer@drury.edu)

March 2025

Abstract

Relationship-based access control (ReBAC) is a type of access control model that uses relationships between entities in a system to determine who can access what resources. To this end, the data is organized as a graph structure with entities represented as nodes and their relationships as edges. Then, a ReBAC policy will check to see if proper sequences of relationships (relationship patterns) exist in the graph to determine if an entity has access to a resource. There have been several different policy mining techniques implemented to convert low-level access control (LLA) into ReBAC. To ensure that a mined policy has captured the correct rules, we have developed an efficient testing method to ensure that each relationship pattern in the policy can be verified as a significant rule.

1 Introduction

Access control is an important service of information systems that specifies if a user has access to a protected resource. As organizations grow and their information systems become more complex, efficiently managing the access control requirements has become increasingly complex. This has led to the development of numerous access control models, including the *relationship-based access control* (ReBAC) model [Fong and Siahaan(2011)]. In ReBAC, a policy is comprised of a set of rules, where each rule specifies a sequence of relationships (*relationship patterns*) that should exist between entities to have access permissions (e.g., a teacher can access the grades of a student if the student is enrolled in their class). ReBAC is a more modern access control paradigm and more expressive than some of the older, prominent models such as role-based [Sandhu et al.(1996)] (e.g., teachers can access student grades), because of its ability to regulate access control based on relational predicates.

However, such fine-grained control over access permissions hinders the adoption of ReBAC. Suppose an organization wants to migrate to ReBAC. They have access to the *system graph* and the *low-level authorizations* (LLA) (e.g., Alice can access Grade A). Based on these two inputs, they want to produce a policy consisting of a set of high-level rules that are based on the relationships between the underlying entities. Manually generating such a policy can be tedious and error-prone; there is usually a multitude of different relationship patterns that can exist between a pair of entities in the system graph. To cope with these challenges, access controls in large systems have attempted to convert from LLAs to ReBAC policies through an automated process. This process is famously known in the access control literature as *policy mining* and their goal is to convert from low-level to high-level policies while preserving all the existing permissions.

Policy miners [Bui et al.(2019), Lu et al.(2008), Xu and Stoller(2014)] have attained a fair bit of progress with regard to successfully mining the correct rules that preserve all the low-level permissions. Mining algorithms are given a graph and an LLA to determine which paths in the graph constitute rules that are part of a correct policy. However, many policies depend on large, complicated graphs and LLAs and may incorrectly identify a particular relationship pattern as a rule, especially because there can be multiple, different paths (some rules and some not) between a single pair of entities.

Checking whether a policy is correct requires either a clear and accurate policy specification to compare with or checking the complete LLA to ensure we have verified all possible access permissions. Nevertheless, in the real world, a clear policy specification is rarely available and it is inefficient, even impossible, to explore the complete LLA of a large system with large numbers of users and resources. Therefore, we need an efficient way to verify whether a mined policy conforms with the existing LLA (*conformance testing*). To this end, at a high level, our algorithm uses the system graph and the LLA and finds evidence for or against different rules that have been identified by the miner by mining the ReBAC policy itself and recording instances where the rules are applicable. This will generate a much smaller LLA that can efficiently verify the mined policy and this approach also does not need access to a ground-truth policy.

2 System Graph and Low-Level Authorizations

As mentioned in the introduction section, both policy miners and our conformance testing approach utilize system graphs and LLA for their functionalities, and so we clearly formulate both of these concepts. LLAs are composed of two-dimensional lists that simply state: entity '1' has or does not have access to entity '2'. Between a pair of entities, an *access request* is stored in an LLA that captures both the requesting entity and the requested entity. Therefore, each LLA record comprises an access request and corresponding access decision according to the access control policy. The LLA stores N(N-1) access requests where N is the number of entities in the system. ReBAC models are composed of a graph, $G(V, E, L)$ that has labeled relationships (i.e., edges E labeled with

a relationship type from L) between entities (i.e., nodes V). Access is granted between entities in an access request on the basis of a set of *rules*, where each rule is a sequence of relationship types $[l_1, l_2, \ldots, l_n]$. For example, entity 'x' has a path consisting of edges (a, c, b, a) connecting to entity 'y'. A rule states that nodes that have a path of (a, c, b, a) are granted access. In this case, 'x' would be granted access to 'y'. In the following, we provide a visualization of a small graph and corresponding LLA if the policy consists of the rules: $\{BR, BB\}$. Here, BR (similarly for BB) stands for a black edge followed by a red edge.

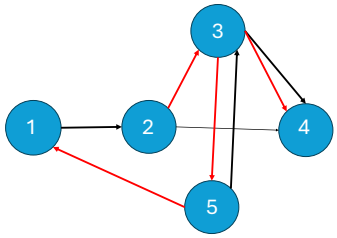

Figure 1: Sample Graph

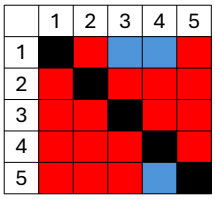

Figure 2: LLA

3 Problem Statement

The issue that arises from policy miners is the inability to verify if a mined policy contains rules that do not grant access and missing ones that do. Our solution to the problem is to mine the policy ourselves and verify that any given relationship pattern is accounted for as a grant or a non-grant. By recording the relevant nodes, sections of the LLA pertaining to those nodes, and the paths that are present, we can provide evidence for or against the rules in a given mined policy. For instance, if a rule states that a relationship pattern of ABB grants access but does not, our conformance tester would reject the rule and present nodes 1,3,2,11 along with the LLA showing nodes 1 to 11 is false.

Our miner is given two inputs – an LLA and a graph – and will output a reduced LLA, policy, and set of nodes proving each grant or non-grant. Mining the policy heavily relies on a variation of depth-first search, but by pre-processing the nodes for connectivity within a certain distance we can begin our mining on more promising nodes than if we were to arbitrarily select the order. Our conformance tester is built on this output. The tester will then be able to receive a policy and return the rules that are false as well as the rules that are missing.

A major significance of our approach is that it tremendously reduces the number of LLA test cases that our conformance tester needs to completely verify a policy. To check whether a policy has extra rules or is missing any rules, previous works have mostly resorted to using the complete LLA or sampling a random subset of the LLA, when a clear, original policy is unavailable [Bui et al.(2019), Xu and Stoller(2014), Iyer and Masoumzadeh(2020)]. However, such exhaus-

tive and random methods require a much larger number of test cases compared to our approach (as we will show in our experiments). Besides, the randomized approach might not even identify all errors in a policy because of missing some important test cases. Therefore, our approach provides an efficient as well as a correct solution for conformance testing.

4 Mining the Policy

In order to setup our problem we first must create a graph and LLA to use for mining. Using Erdős–Rényi graph generation, we create a graph with N nodes and M edges. We then create an LLA for all nodes and by default the access is "False". We then randomly create a set of rules that must be applied. Once the rules are established, we apply the rules to each node and set the access to "True" if the rule applies to the given path. Note that each node can be connected once for each relationship type. Two methods will be discussed.

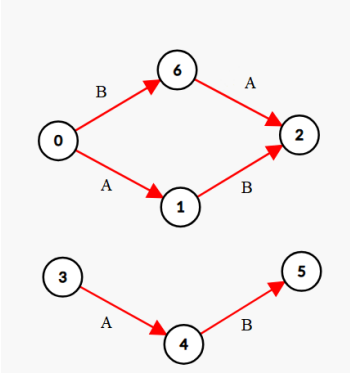

Figure 3: Sample Graph

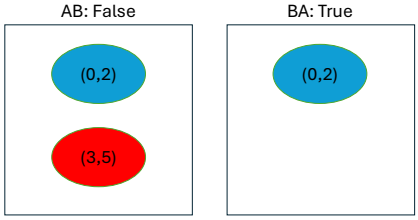

Figure 4: LLA

4.1 Proving a Rule as a Path

Proving that a path does not grant access is simpler than proving a path does grant access. Any path between two nodes where access is not granted must hold for any other nodes where the same path is given. Finding non-grant rules is simple. There may be instances where two pairs have the same path and access may appear to be granted for one pair and not the other, but if one of the pairs is not granted access the path is not a grant rule.

In Figure 3 the path between nodes 0,1,2 may be the same as 3,4,5. 0 has access to 2, but 3 does not have access to 5. Path AB is confirmed to not grant access. This, however, does not mean that all nodes that are connected via AB are all denies. This simply states that AB does not grant access. Since we know that AB does not grant access, BA must grant access since there are no other

paths from 0 to 2. Both methods will use this logic to deduce whether or not a relationship pattern grants access.

4.2 Brute Force Buckets

A brute force method iterates over each node and performs a DFS-esque examination. Using this method, we place each node pair originating from a parent node inside a "bucket" that contains node pairs that match a particular pattern. After iterating over each node, we will go through each bucket and look for any non-grant pairs utilizing the LLA. A singular non-grant confirms that the pattern does not grant access. If a bucket only contains grants, and the presence of a node pair exists only within that bucket, we can confirm that the pattern grants access.

4.2.1 Creating Buckets

Let M be the max length of a potential rule and R be the number of relationship types. The number of buckets B will be: $\sum_{n=1}^{M} R^n = B$. For example, if there are 3 relationship types and M is 3, there would be 39 different buckets representing the different relationship types (a, b, c, aa, ab, etc.). Each bucket will contain node pairs that have a relationship pattern.

4.2.2 Confirming Patterns

Each bucket will contain node-pairs that have a path of a single relationship pattern. A single instance of a non-grant confirms that a pattern does not grant access. We can eliminate the entire bucket and the nodes within. Figure 4 shows a representation of buckets AB and BA using the example in Section 4.1. Since node pair (0,2) does not appear in any other non-eliminated bucket, we can confirm that this pattern grants access.

Note that all buckets are not represented. Since node (0,2) will not appear in any other bucket, we conclude that access is granted via pattern BA as bucket AB contains non-grants. For cases where a node appears as a grant in several buckets, we wait until only one of the buckets is confirmed true. If the node pair(3,5) did not exist, we will not confirm that pattern AB or BA grants access.

4.3 Algorithm for Brute Force Method

An issue that may arise is the instance of a bucket that only contains grants, but the node-pairs within are also present in another grant only bucket. Due to the way we generate our graph, we will always have at least one instance of a non-grant represented in order to avoid this edge case. The *exploreNode* function essentially performs depth first search within M nodes and returns the target node and relationship pattern(s) to connect. This is the most costly part of the algorithm as we must inspect each node's neighbors up to M distance. The second part of the algorithm runs in O(N) time where N is the number of node pairs inside the buckets.

```
1: Procedure BucketMine(R, M, G : Graph, LLA)
2: B = CreateBuckets(R, M)
3: for node in G do
4:    nodePair, bucket = exploreNode(node) returns (partent,target), Relationship-
      Pattern
5:    Add nodePair to B[bucket]
6: end for
7: policy = []
8: i = 0
9: while B do
10:   Cur = B[i]
11:   for j in Cur do
12:      np = Cur[j]]
13:      if LLA[np] = False then
            Remove B and continue to next B
14:      end if
15:      if np not in other B then
            add B to rules
16:      end if
17:   end for
18: end while
19: return policy
20: End Procedure =0
```

4.4 Heuristic Mining

The second method we developed utilizes the logic of confirming grants and non-grants in Section 4.2.2. Instead of examining all nodes, we first explore the nodes that are the *least connected* to other nodes. To do this we iterate through each node and used a reduced depth first search to see how many nodes are connected within M nodes. Alternatively, since this can be computationally expensive, we can arbitrarily replace M with a smaller number. The heuristic we utilize is based on the assumption that nodes that have lower amount of connections will likely have less grants that have multiple paths.

After sorting by least connected to most, we iterate through the nodes in order to mine the policy. Each time a non-grant is discovered, we record the relationship pattern as such. If a grant is discovered and is the only viable path, we record the pattern as a rule. When all relationship patterns with length M and less are accounted for, the algorithm outputs a policy and a reduced LLA consisting of the relevant node-pairs as proof.

4.5 Why the Heuristic Matters

The main reason we seek to explore nodes that are less connected is that the conditions necessary to prove that a relationship pattern is a rule is more likely

to be contained within said nodes. Consider the figures below.

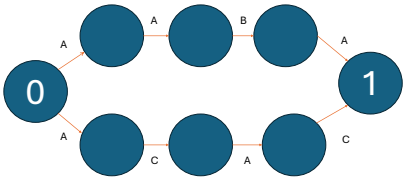

Figure 5: Graph 1

Figure 6: Graph 2

In Figure 5, node 0 may have access to node 1 by two different paths. In Figure 6, node 0 can have access only via one path. If a parent node is granted access to a target node but has multiple paths to said node, the node-pair does not have any value to the miner unless we can disprove all but one path. It possible that a parent node has access to a target node via two different paths.

In reference to Graph 1 in Figure 5, in very rare instances, we could be faced with a situation where one of the following is true: 1. "AABA" is granting access. 2. "ACAC" is granting access. 3. Both patterns are granting access.

In situations like this, we must wait to confirm that one of the two patterns is a non-grant. While very unlikely, a graph may contain a situation where both patterns are grants, but the only instances of the patterns are represented on the same node pair. In our case, we want and can avoid these sorts of node-pairs altogether by using the heuristic. In graph 1, the parent node is connected to 7 nodes while graph 2 is connected to 4 nodes. Due to preprocessing, when both subgraphs are a part of a larger main graph, we will always explore graph 2 first, circumventing the ambiguity contained within graph 1.

4.6 Algorithm for Heuristic Method

In this algorithm we will keep track of all patterns and set them to 'None' by default. Once every pattern is found to be grant or non-grant, the algorithm will end. In the examine function, a node will traverse up to M nodes away and check the LLA to update the policy. If a pattern between nodes appears to grant access is found and only has one existing path after filtering out the paths by the confirmed non-grants, we add the pattern to the set of confirmed grants. Otherwise we save the node, target node, and paths to a dictionary for re-examination once more non-grants are confirmed. During re-examination, if a previously ambiguous grant only has a single viable path, we can confirm that this path grants access as the other options will be eliminated as more non-grants are confirmed. Along the way, we will save the node-pairs that are being used to prove the status of relationship patterns.

```
1: Procedure HeuristicMine(R, M, G : Graph, LLA)
2: policy = CreatePolicy(R, M)
3: evidence = dict(): contains (pattern, node-pair)
4: grant, nGrant = set() set()
5: aGrant = dict(): contains(node-pair, paths)
6: nodes = G.nodes
7: for node in G do
8:    nodePair, connectiviy = findConnectivity(node)
9:    Add nodePair to nodes
10: end for
11: sortedNodes = nodes.sort() sorting nodes by connectivity
12: while policy not complete do
13:    for sNode in sortedNodes do
14:       grants, nonGrants, ambiguousGrants(tNode, paths)=examine(sNode)
15:       aGrant.reExamine()
16:       aGrant.update(sNode, tNode, paths)
17:    end for
18: end while
19: return (policy, evidence)
20: End Procedure =0
```

5 Experiments and Evaluation

We implemented a prototype of our algorithms in Python using *networkx*[1]. In the following sections, we will be looking at the heuristic algorithm along with variations to highlight the efficacy of the heuristic. They differ by: *not* sorting the nodes by connectivity and by *reversing* the heuristic by exploring highly connected nodes first. It should be noted that the number of nodes explored in each test set were in the single to low double-digit range (1-19). The brute force method would attempt to explore every single node and thus, for the sake of brevity, we will not compare the brute force method to the heuristic.

5.1 Experimental Setup and Parameters Involved

When creating the graph, we must consider how many nodes ($\mid V \mid$) and edges ($\mid E \mid$) we want to create. In addition, since each edge represents a relationship, we have to consider how many types of relationships to represent ($\mid L \mid$). Finally, when creating rules for a policy we must choose how many rules we want to create as well as a maximum length of a rule (*Max.Len.*).

For testing, we generated a random graph using Erdős–Rényi. The maximum rule length, number of nodes, and number of edges will vary. Each iteration contains 50 runs showing a low to high, unsorted, and high to low list of nodes. We chose to use 30 randomly selected rules of length 4 or more. Runtime does not include pre-processing (described in Section 4.4 when finding connectivity).

[1]The GitHub link for our code is `https://github.com/bmichaliszyn/Iyer-Michaliszyn`.

In these runs, the algorithm fully finds the connectivity up to the maximum length of the rule. Table 1 summarizes the results from our experiments.

$\mid V \mid$	$\mid E \mid$	$\mid L \mid$	**Avg Runtime (s)**			*Max. Len.*	**Avg Reduced LLA**		
			L	U	H		L	U	H
500	3000	3	0.07	0.19	0.23	5	1146	984	912
1000	8000	3	0.28	1.99	2.53	5	2138	2102	2102
1000	10000	5	0.12	0.20	0.24	4	1480	1334	1306
1000	8000	4	0.49	0.85	0.89	5	2711	1880	1885

Table 1: Experimental Data and Results Demonstrating Performance of our Heuristic Algorithm 4.6

5.2 Observations and Results

Our algorithm demonstrates that sorting by connectivity does indeed run significantly faster than a non-sorted list of nodes. The average reduced LLA size does not seem to change whether you use the heuristic or not. Rows 1 and 3 may indicate this, but row 2 defies this assumption. It is not until row 4 where this is noticeably significant. In the grand scheme of things, this is not a trade-off worth considering, as the original LLA includes $n(n-1)$ access requests.

In row 2, the time complexity is almost 5 times as fast with the cost of an LLA that is 136 access requests larger. This test run is the least egregious LLA size increase. The original LLA contains 999,000 access requests. Row 4 is almost twice as fast, and while the reduced LLA is 40 percent larger for the faster execution, the reduced LLA is still significantly smaller than the original LLA. In real-world situations there will not be as many high density nodes as there are in our generated graph. For example, it would typically *not* be the case that a doctor has twice the patients as another doctor or that a doctor is connected to every other patient in the system.

We also performed experiments to see if our reduced LLA is able to detect errors in a policy. Specifically, after mining a policy, we take any other (potentially incorrect) policy and compare the rules found by each. If there is a disagreement about a rule, our program will provide evidence in the form of a node pair alongside the reduced LLA that pertains to the rule in contention. For instance, a policy was input with a rule (A, A, C), and our mined policy found this to be incorrect. Our program declared this as an error and provided a node pair (0,233) and also showed within the LLA the access request where this pattern is untrue (i.e., the access request was denied instead of being granted).

6 Conclusion and Future Work

The algorithm is significantly faster than the brute-force method and provides a practical way to test existing mined policies for small ReBAC systems. This makes our approach especially suitable for large, enterprise systems with regular policy updates and so need fast policy verification cycles where traditional approaches become infeasible. While large systems may require considerable preprocessing time, potential workarounds include testing a subset of nodes for connectivity, using a fraction of the largest rule as a basis. Another promising approach is leveraging a Gaussian distribution, selecting only nodes that meet a threshold on the lower end of the curve.

A key limitation of our work is that ReBAC models are derived from real-world systems such as hospitals, social media platforms, and universities. However, our testing relies on Erdős–Rényi graphs. While these offer valuable insights, real-world systems do not perfectly align with these graph structures.

Despite these concerns, our heuristic approach of examining less-connected nodes appears to hold true. Further testing with adjusted parameters could provide deeper insights. Additional metadata that may be of interest includes the total number of nodes visited, the ratio of instantiated edges to total possible edges, and the significance of inverse connectivity.

References

[Bui et al.(2019)] Thang Bui, Scott D Stoller, and Hieu Le. 2019. Efficient and Extensible Policy Mining for Relationship-Based Access Control. In *Proc. ACM Symposium on Access Control Models and Technologies*. 161–172.

[Fong and Siahaan(2011)] Philip WL Fong and Ida Siahaan. 2011. Relationship-based access control policies and their policy languages. In *Proc. ACM SACMAT*. 51–60.

[Iyer and Masoumzadeh(2020)] Padmavathi Iyer and Amirreza Masoumzadeh. 2020. Active Learning of Relationship-Based Access Control Policies. In *Proc. ACM Symposium on Access Control Models & Technologies*. 155–166.

[Lu et al.(2008)] Haibing Lu, Jaideep Vaidya, and Vijayalakshmi Atluri. 2008. Optimal boolean matrix decomposition: Application to role engineering. In *IEEE 24th International Conference on Data Engineering*. IEEE, 297–306.

[Sandhu et al.(1996)] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. 1996. Role-based access control models. *Computer* 29, 2 (1996), 38–47.

[Xu and Stoller(2014)] Zhongyuan Xu and Scott D Stoller. 2014. Mining attribute-based access control policies. *IEEE Trans. Dependable Secure Comput.* 12, 5 (2014), 533–545.