On Feasibility of Attribute-Aware Relationship-Based Access Control Policy Mining

Shuvra Chakraborty and Ravi Sandhu

Institute for Cyber Security (ICS) and NSF Center for
Security and Privacy Enhanced Cloud Computing (C-SPECC)
Department of Computer Science
University of Texas at San Antonio, San Antonio, Texas, USA
{shuvra.chakraborty,ravi.sandhu}@utsa.edu

Abstract. This paper studies whether exact conversion to an AReBAC (Attribute-aware Relationship-Based Access Control) system is possible from an Enumerated Authorization System (EAS), given supporting attribute and relationship data. The Attribute-aware ReBAC Ruleset Existence Problem (ARREP) is defined formally and solved algorithmically, along with complexity analysis. Approaches to resolve infeasibility using exact solutions are discussed.

Keywords: Access Control Policy Mining · Attribute-aware Relationship-Based Access Control · Attribute-Based Access Control · Relationship-Based Access Control.

1 Introduction

Relationship-Based Access Control (ReBAC) [7] emerged from the access control requirements of Online Social Networks. ReBAC expresses authorization policy in terms of various relationship parameters such as type and depth, whereas Attribute-Based Access Control (ABAC) [8] has been motivated by its generalized structure and versatility in access control policy specification through attributes of users, resources and environment. Although ReBAC expresses authorization through direct and indirect relationships, there are cases where using relationships only is insufficient. Consider a social network policy where only adults (18 or higher age) can send a friend request to anyone who lives in the same location as themselves. Here, both age and location of each user in the network must be known. Formally, these two required characteristics/attribute can be incorporated with users to make policy generation more expressive and flexible. Integrating attributes with ReBAC components certainly add more expressiveness [6], formally named as Attribute-aware ReBAC (AReBAC).

Generally, access control policy mining facilitates automation of migrating from one access control system to another with certain set of assumptions such as allowing direct use of entity ID in rule generation, strict or approximate equivalency between the source and generated policy, and availability of appropriate

supporting data. Deployment of manual effort to convert from one access control system to another could be tedious, labor-intensive and error-prone.

This paper analyzes the feasibility of AReBAC policy mining from a given Enumerated Authorization System (EAS) under certain assumption, for example, no user ID will be allowed in the generated AReBAC rule. Note that, rule generation is always possible with use of entity IDs. The major contributions made in this paper are as follows.

- The first formal notion of Attribute-aware ReBAC RuleSet Existence Problem (ARREP) is developed. A novel algorithm for AReBAC policy mining feasibility detection is presented along with complexity analysis.
- Infeasibility problem in ARREP is formulated. Furthermore, exact solutions are proposed.
- Rule structure generality and unrepresented path label problem are noted.

2 Related Works

Although both ABAC and ReBAC have their own advantages to express authorization policies (see [1] for a rigorous comparison of their expressive power), integrating ABAC with ReBAC can provide finer-grained controls and improve the expressiveness of standalone ABAC or ReBAC. For example, [6] presents an attribute-aware ReBAC access control model.

Although the policy specification language in this paper is very different from [2, 9], these two works are relevant related work. In [2], an approach to mine ABAC and ReBAC policies has been proposed where access control lists and incomplete information about entities are given. A few significant points about [2] are i) the proposed algorithm prefers the context of ReBAC mining because ReBAC is more general than ABAC, ii) entity ids are allowed to be used (which makes the generated policy less general), and iii) there is a policy quality metric available. Compared to [2], entity ids are strictly prohibited in the attribute-aware context of this paper. On the other hand, [9] presents an attribute-supporting ReBAC model for Neo4j (a popular graph database) that provides finer-grained access control by operating over resources.

While this paper introduces feasibility in the field of AReBAC policy mining for the first time, there are a few similar prior feasibility studies as follows.

- The work in [4] introduces ABAC RuleSet Existence Problem for the first time. Besides, the notion of infeasibility correction has been discussed.
- The work in [5] adapts and extends ABAC RuleSet Existence Problem for RBAC input. Additionally, it proposes infeasibility solution, with and without presence of supporting attribute data.
- In [3], feasibility of ReBAC policy mining has been investigated for the first time, assuming user to user relations are given by a static relationship graph.

3 Attribute-aware ReBAC RuleSet Existence Problem

This section defines the ARREP along with a feasibility detection algorithm, complexity analysis and related issues.

3.1 Preliminaries

A user/subject is an entity who performs operation on a resource/object. The set of users is represented by U. A user requests to perform an operation on another user. An operation is an action performed by a user on another user. The set of operations in the system is represented by OP. Without loss of generality it is assumed that OP is a singleton given by $\{op\}$, since each operation has its specific policy or rules. An access request is a tuple $\langle u, v\rangle$ where user u is asking permission to perform operation op on user v where $u, v \in U, op \in OP, u \neq v$. An access request is either granted or denied, based on the access control policy. In any access control system, a logical construct is required to decide the outcome of an access request. The logical construct is formally defined as, $checkAccess: U \times U \rightarrow \{True, False\}$, where the result $True$ grants access while $False$ denies it.

We define a simple authorization system, EAS as follows:

Definition 1. ***Enumerated Authorization System (EAS)***
An EAS is a tuple $\langle U, AUTH, checkAccess_{EAS}\rangle$ where, U is the finite sets of users and $AUTH \subseteq U \times U$, is a specified authorization relation where

$$checkAccess_{EAS}(u, v) \equiv (u, v) \in AUTH$$

For example, given U = $\{Alice, Bob\}$ and OP=$\{readData\}$, Bob can read Alice's data iff (Bob, Alice) belongs to AUTH.

In order to define an Attibute-aware ReBAC system, the key component is Attribute-aware Relationship Graph (ARG), which is defined as follows.

Definition 2. ***Attribute-aware Relationship Graph (ARG)***
The Attribute-aware Relationship Graph
ARG = (V, VA, VA-RangeSet, UATTValue, EA, EA-RangeSet, E)
is a directed labeled graph where,

a. *V is the set of vertices in ARG, representing the set of users in the system.*
b. *VA is the finite set of atomic user attribute function names $\{va_1, va_2, ..., va_m\}$.*
c. *For each $va_i \in VA$, Range(va_i) specifies a finite set of atomic values for user attribute va_i. VA-RangeSet = $\{(va_i, value)|va_i \in VA \wedge value \in Range(va_i)\}$.*
d. *UATTValue denotes the user attribute value assignments. UATTValue = $\{UATTValue_{va_i}|va_i \in VA\}$ where $UATTValue_{va_i}: V \rightarrow Range(va_i)$. For convenience, we understand $va_i(a)$ to denote $UATTValue_{va_i}(a)$, that is the attribute value assignment of an actual user a for attribute va_i.*
e. *EA is the finite set of edge attribute function names, $\{ea_1, ea_2, ..., ea_n\}$.*

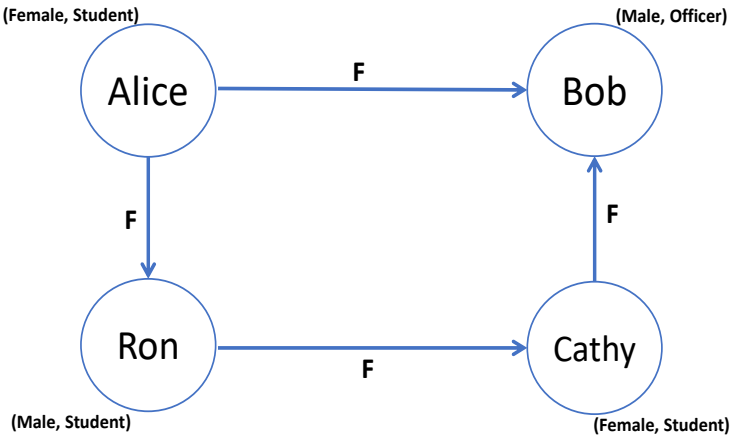

Fig. 1. Example ARG

f. For each $ea_i \in EA$, Range(ea_i) specifies a finite set of atomic values for edge attribute ea_i. EA-RangeSet = $\{(ea_i, value) | ea_i \in EA \wedge value \in Range(ea_i)\}$.

g. $E \subseteq V \times V \times Range(ea_1) \times Range(ea_2) \times ... \times Range(ea_n)$ is a finite set of directed edges where, an edge $(u, v, \sigma_1, \sigma_2, ..., \sigma_n) \in E$, $u \neq v$, represents the relations $\sigma_1, \sigma_2, ..., \sigma_n$ from user $u \in V$ to $v \in V$ in ARG where $\sigma_1 \in Range(ea_1), \sigma_2 \in Range(ea_2), ..., \sigma_n \in Range(ea_n)$.
Note: For a directed edge e from vertex a to vertex b in ARG, $ea_i(e)$ specifies the associated edge attribute value assignment for $ea_i \in EA$.

Fig. 1 presents an ARG where the set of users V = {Alice, Bob, Cathy, Ron}, the set of user attribute function names, VA = $\{Gender, Profession\}$, the set of edge attribute function names, EA = $\{Relation - type\}$, and the set of edges E = { (Alice, Ron, F), (Alice, Bob, F), (Ron, Cathy, F), (Cathy, Bob, F) }. The user and edge attribute value assignments are shown in Fig. 1. The notion of a path in an ARG is defined as follows:

Definition 3. ***Path in ARG***
Given ARG as in Def. 2 and a vertex pair $(u, v) \in V \times V$ where $u \neq v$, a path from u to v is a sequence of edges where the terminating (i.e., second) vertex of each edge is same as the starting (i.e., first) vertex of the next edge given by $\langle (u, v_i, \sigma_{w1}, \sigma_{w2}, ..., \sigma_{wn}), (v_i, v_j, \sigma_{x1}, \sigma_{x2}, ..., \sigma_{xn}), ..., (v_k, v_l, \sigma_{y1}, \sigma_{y2}, ..., \sigma_{yn}), (v_l, v, \sigma_{z1}, \sigma_{z2}, ..., \sigma_{zn}) \rangle$, where

a. $u, v_i, v_j, ..., v_k, v_l, v \in V$

b. $\sigma_{w1}, \sigma_{x1}, ..., \sigma_{y1}, \sigma_{z1} \in Range(ea_1)$, $\sigma_{w2}, \sigma_{x2}, ..., \sigma_{y2}, \sigma_{z2} \in Range(ea_2)$, ..., $\sigma_{wn}, \sigma_{xn}, ..., \sigma_{yn}, \sigma_{zn} \in Range(ea_n)$.

A path p from u to v is said to be simple iff $u, v_i, v_j, ..., v_k, v_l, v \in V$ *are distinct. The length of p, denoted by* $|p|$*, is the number of edges in the path. The attribute aware path label of the path p from u to v, denoted by* $pathLabel_{att}(p)$*, is* $(va_1(u), va_2(u), ..., va_m(u)).(\sigma_{w1}, \sigma_{w2}, ..., \sigma_{wn}).(va_1(v_i), va_2(v_i), ..., va_m(v_i)).(\sigma_{x1}, \sigma_{x2}, ..., \sigma_{xn}).(va_1(v_j), va_2(v_j), ..., va_m(v_j))..... (va_1(v_k), va_2(v_k), ..., va_m(v_k)).(\sigma_{y1}, \sigma_{y2}, ..., \sigma_{yn}).(va_1(v_l), va_2(v_l), ..., va_m(v_l)).(\sigma_{z1}, \sigma_{z2}, ..., \sigma_{zn}).(va_1(v), va_2(v), ..., va_m(v))$.

Clearly, $pathLabel_{att}(p)$ is a string, consisting of concatenated tuples of vertex and edge attribute value assignments, traversed in the same order as the vertices and edges appear in path p. Note that, the vertex and edge attribute values follow specific orders, given by $\langle va_1, va_2, ..., va_m \rangle$ and $\langle ea_1, ea_2, ..., ea_n \rangle$, respectively. For sth edge in path p where $1 \leq s \leq |p|$, starting vertex, edge, and terminating vertex attribute value assignments are represented by $(2 \times s - 1)$th, $(2 \times s)$th, and $(2 \times s + 1)$th tuples in $pathLabel_{att}(p)$, respectively.

Given ARG in Fig. 1, the only path p from Cathy to Bob is $\langle (Cathy, Bob, F) \rangle$ with $pathLabel_{att}(p) = (Female, Student).(F).(Male, Officer)$. Henceforth, we understand path to mean simple path.

Definition 4. ***Attribute aware ReBAC policy***
An Attribute aware ReBAC policy, POL_{AAR} *is a tuple, given by* $\langle$ *OP, VA, EA, RuleSet*$\rangle$ *where,*

a. *OP, VA, and EA are as defined in Def. 2.*
b. *RuleSet is a set of rules where, for each operation* $op \in OP$*, RuleSet contains a rule* $Rule_{op}$*. Each* $Rule_{op}$ *is specified using the grammar below.*
$Rule_{op} ::= Rule_{op} \vee Rule_{op} \mid pathRuleExpr \mid Attexp$
$pathRuleExpr ::= pathRuleExpr \wedge pathRuleExpr \mid (pathLabelExpr)$
$pathLabelExpr ::= pathLabelExpr.pathLabelExpr \mid edgeExp$
$Attexp ::= Attexp \wedge Attexp \mid uexp = value \mid vexp = value$
$edgeExp ::= edgeExp \wedge edgeExp \mid edgeuexp = value \mid edgevexp = value \mid edgeattexp = value$
where, value is a atomic constant.
$uexp \in \{va(u) | va \in VA\}$*, u is a formal parameter.*
$vexp \in \{va(v) | va \in VA\}$*, v is a formal parameter.*
$edgeuexp \in \{va(e.u) | va \in VA\}$*, e.u is a formal parameter.*
$edgevexp \in \{va(e.v) | va \in VA\}$*, e.v is a formal parameter.*
$edgeattexp \in \{ea(e) | ea \in EA\}$*, e is a formal parameter.*

Here "." is the concatenation operator. The length of a pathLabelExpr is given by the number of concatenation operators plus 1. A pathLabelExpr can be split at the point of each . operator into edgeExp, and numbered sequentially, starting from 1 to the length of the pathLabelExpr.

Based on the stated POL_{AAR}, the following defines an access control system:

Definition 5. ***Attribute aware ReBAC system***
An Attribute aware ReBAC system is a tuple, $\langle ARG, POL_{AAR}, checkAccess_{AAR}\rangle$ *where ARG and* POL_{AAR} *are as in Def. 2 and 4, respectively. For an access request (a, b),* $checkAccess_{AAR}$*(a:V, b:V)* $\equiv$ $Rule_{op}$*(a:V, b:V) where* $Rule_{op}$ *is evaluated as follows:*

Step 1:

a. *for each Attexp in* $Rule_{op}$*, substitute the values va(a) for va(u) and va(b) for va(v), where* $va \in VA$*.*
b. *For a pathLabelExpr in* $Rule_{op}$*, substitute True iff i) there exists a simple path p from a to b in ARG such that* $|p|$ *= length of pathLabelExpr, and ii) each* s*th edgeExpr of the pathLabelExpr where* $1 \leq s \leq$ *length of pathLabelExpr, evaluates to True. To evaluate* s*th edgeExpr, substitute va(e.u), ea(e), and va(e.v) by the corresponding* $va \in VA$*,* $ea \in EA$*, and* $va \in VA$ *attribute value assignments from* $(2 \times s - 1)th$*,* $(2 \times s)th$*, and* $(2 \times s + 1)th$ *tuples in* $pathLabel_{att}(p)$*, respectively.*

Step 2:
Evaluate the resulting boolean expression.

User a is permitted to do operation op on object b if and only if $Rule_{op}(a, b)$ *evaluates to True.*

For example, given ARG in Fig. 1, and $Rule_{op}$ = (Gender(e.u)=Female $\wedge$ Profession(e.u)=Student $\wedge$ Relation-type(e) = F $\wedge$ Gender(e.v)=Male $\wedge$ Profession(e.v)=Student), $Rule_{op}(Alice, Ron)$ evaluates to True.

Although both ReBAC and ABAC are powerful, flexible and comparable [1] in expressing authorization policies, relying solely on one is often insufficient. An example case will be used in order to compare ABAC policy presented in [4] and ReBAC policy in [3] with the proposed AReBAC policy in Def. 4. Consider the ARG in Fig. 1 and Table 1. Each row of table 1 represents a case and an associated authorization state example.

1. Row 1 indicates that both AReBAC and ReBAC policies can express the authorization state (Alice, Bob) whereas only ABAC rules cannot. ABAC rule fails because Alice and Cathy have the same attribute value combination. The generated ReBAC and AReBAC rules are "F.F.F" and "(Relation-type(e)=F.Relation-type(e)=F.Relation-type(e)=F)", respectively.
2. Row 2 indicates that both ABAC and AReBAC policies can express the authorization state (Ron, Bob) whereas only ReBAC rules cannot. ReBAC rule fails because there is only one path labeled "F.F" from Ron to Bob which is satisfied by unauthorized pair, such as, (Alice, Cathy). The generated ABAC and AReBAC rule is the same, "Gender(u)=Male $\wedge$ Profession(u)=Student $\wedge$ Gender(v)=Male $\wedge$ Profession(v)=Officer".
3. The 3rd row, authorization state (Alice, Ron), cannot be expressed by both ABAC and ReBAC. ABAC rule fails because Alice and Cathy have the same attribute value combination. ReBAC rule fails because the only path label

Table 1. Example data

ReBAC	ABAC	AReBAC	AUTH
Yes	No	Yes	{(Alice, Bob)}
No	Yes	Yes	{(Ron, Bob)}
No	No	Yes	{(Alice, Ron)}
No	No	No	{(Bob, Alice)}

"F" is satisfied by other unauthorized pairs, such as (Alice, Bob). The AReBAC rule is "(Gender(e.u)=Female ∧ Profession(e.u)=Student ∧ Relationtype(e) = F ∧ Gender(e.v)=Male ∧ Profession(e.v)=Student)".

4. The 4th row, Auth = {(Bob, Alice)} is not expressible by only ABAC (Since (Bob, Cathy) will be allowed), only ReBAC (since no path exists from Bob to Alice), and AReBAC ((Bob, Cathy) will be allowed and no path exists).

According to the used policy specification language, AReBAC is more expressive than ABAC [4] and ReBAC [3]. Additionally, it can be clearly observed that, if entity ids are allowed, AReBAC policy will never fail (such as [2]). However, imposing this condition conflicts with core principles of ABAC and ReBAC. Therefore, the AReBAC policy specification in this paper checks whether the target access control system could be generated avoiding explicit use of unique entity id. Based on this motivation, ARREP problem is defined as follows:

Definition 6. ***Attribute aware ReBAC RuleSet Existence Problem (ARREP)*** *Given an EAS and an ARG as in Def. 1 and 2, respectively, where V=U, does there exist a RuleSet as in Def. 4 so that the resulting Attribute aware ReBAC system satisfies:*

$$(\forall u, v \in U)[checkAccess_{AAR}(u, v) \Leftrightarrow checkAccess_{EAS}(u, v)]$$

Such a RuleSet, if it exists, is said to be a suitable RuleSet, otherwise the problem is said to be infeasible.

The following subsection develops a ARREP solution algorithm.

3.2 ARREP Solution Algorithm

Algorithm 1 resolves the ARREP problem. Given an ARREP instance, it returns either feasible status and $Rule_{op}$, or infeasible status, incomplete $Rule_{op}$ and failed authorizations. Given any graph, the task finding all possible simple paths from a source vertex to a target vertex is well known, hence, details of function FindAllSimplePath() in Algorithm 1 are not provided (it can be adapted from [3]). The overall complexity of computing all possible paths from a vertex to another in ARG is $O(|E|!)$ as it considers only simple paths.

Theorem 1. *The overall complexity of ARREP feasibility detection Algorithm 1 is* $O(|V|^4 \times (|E|!))$.

Algorithm 1 ARREP Solution Algorithm

Input: An EAS and an ARG where V=U.

Output: Feasible/infeasible status and $Rule_{op}$. If infeasible, failedAuthPairs.

1: $Rule_{op}$:= NULL
2: failedAuthPairs := $\emptyset$
3: tempAUTH := AUTH
4: **for** each $(a, b) \in tempAUTH$ **do**
5: **if** ABAC-Expr(EAS, VA, UATTValue, a, b) == SUCCESS **then**
6: **if** $Rule_{op}$ is NULL **then** $Rule_{op} := \bigwedge_{va \in VA} va(u) = va(a) \wedge \bigwedge_{va \in VA} va(v) = va(b)$
else $Rule_{op} := Rule_{op} \vee \bigwedge_{va \in VA} va(u) = va(a) \wedge \bigwedge_{va \in VA} va(v) = va(b)$
7: $tempAUTH \setminus := \{(a, b)\}$
8: **while** $\exists (a, b) \in tempAUTH$ **do**
9: $SP(a, b)$:= FindAllSimplePath(a,b, ARG)
10: **if** $SP(a, b) = \emptyset$ **then**
11: $failedAuthPairs := failedAuthPairs \cup \{(a, b)\}$ //Not Feasible for (a,b) tuple
12: $tempAUTH \setminus := \{(a, b)\}$ and **Continue**
13: $PATHLABEL_{att}(a.b) := \{pathLabel_{att}(p) | p \in SP(a, b)\}$
14: **for** each $pl \in PATHLABEL_{att}(a.b)$ **do**
15: $SAT_{ab}(pl) = \{(c, d) \in V \times V |$ there exists a simple path s from c to d in ARG, c≠d, (c,d)∉AUTH, pl=$pathLabel_{att}(s)\}$
16: $Q_{ab} := \bigcap_{pl \in PATHLABEL_{att}(a.b)} SAT_{ab}(pl)$
17: **if** $Q_{ab} \neq \emptyset$ **then**
18: $failedAuthPairs := failedAuthPairs \cup \{(a, b)\}$ //Not Feasible for (a,b) tuple
19: $tempAUTH \setminus := \{(a, b)\}$ and **Continue**
20: **if** $Rule_{op}$ is NULL **then** $Rule_{op} := \bigwedge_{pl \in PATHLABEL_{att}(a.b)} (generateRule(pl))$
else $Rule_{op} := Rule_{op} \vee \bigwedge_{pl \in PATHLABEL_{att}(a.b)} (generateRule(pl))$
21: $tempAUTH \setminus := \{(a, b)\}$
22: **if** failedAuthPairs is $\emptyset$ **then**
23: **return** "feasible" and $Rule_{op}$
24: **else**
25: **return** "infeasible" and failedAuthPairs and $Rule_{op}$

Algorithm 2 ABAC-Expr

Input: EAS, VA, UATTValue, vertex a, vertex b.
Output: SUCCESS or FAILURE
1: $R1 = \{u1 | \forall va \in VA.va(a) = va(u1)\}$
2: **if** $\exists u1, u2 \in R1.(u1, u3) \in Auth \wedge (u2, u3) \in \overline{Auth}$ where $u3 \in V$ **then**
3: **return** FAILURE
4: $R2 = \{u4 | (\forall va \in VA.va(b) = va(u4)\}$
5: **if** $\exists u4, u5 \in R2.(u4, u6) \in Auth \wedge (u5, u6) \in \overline{Auth}$ where $u6 \in V$ **then**
6: **return** FAILURE
7: **return** SUCCESS

Algorithm 3 generateRule

Input: String pathlabel
Output: String rule
1: rule := NULL
2: SubStr := splitStr(pathlabel,".") // The splitStr function splits pathlabel using . into an ordered list of substrings, and return the saved substrings into an array.
3: numEdges := (number of elements in SubStr-1)÷ 2
4: //rm function returns the given string after removal of leading "(" and trailing ")"
5: **for** i = 1 to numEdges **do**
6: tempu := splitStr(rm(SubStr[2*i-1]), ",")
7: tempv := splitStr(rm(SubStr[2*i+1]), ",")
8: tempe := splitStr(rm(SubStr[2*i]), ",")
9: **if** rule is NULL **then** rule := $\bigwedge_{1 \le j \le m} va_j(e.u) = tempu[j] \wedge va_j(e.v) = tempv[j] \wedge \bigwedge_{1 \le k \le n} ea_k(e) = tempe[k]$ **else** rule := rule . $\bigwedge_{1 \le j \le m} va_j(e.u) = tempu[j] \wedge va_j(e.v) = tempv[j] \wedge \bigwedge_{1 \le k \le n} ea_k(e) = tempe[k]$ //. means the concatenation
10: **return** rule

Proof. In Algorithm 2, overall complexity of Lines 1, 4, 2-3 and 5-6 are O($|U|$), O($|U|$), O($|AUTH|$), and O($|AUTH|$), respectively. Therefore, overall complexity of Algorithm 2 is O($|AUTH|$). The overall complexity of Algorithm 3 is O($|V|$) since the maximum number of edges allowed in a simple path of ARG is $|V|$-1. Combining all these, the computational complexity of Algorithm 1 as follows: Lines 4-7 of Algorithm 1 give $O(|AUTH|^2)$ complexity. According to the complexity of FindAllSimplePath() noted before, Lines 9 and 13, both give $O(|E|!)$ complexity. The overall complexity of Lines 14-15 is $O(|V|^2 \times (|E|!))$, and the set intersection in Line 16 takes $O(|E|!)$. Lines 17-21 can be ignored compared to others, therefore, the loop from Lines 8-21 takes overall $O(|V|^4 \times (|E|!))$ complexity as the loop may iterate $|AUTH| \le |V|^2$ times. Hence, the worst case complexity of Algorithm 1 is $O(|V|^4 \times (|E|!))$.

The correctness proof of Algorithm 1 is similar to the feasibility detection algorithm in [3], and is therefore omitted. Although overall complexity of feasibility

detection algorithm in [3] and Algorithm 1 are same, however, the latter may have more or less computation time. If Algorithm 2 succeeds $\forall(a, b) \in AUTH$, only O($|AUTH|^2$) will be the real computational complexity, which is linear compared to the computed worst case complexity. The computational complexity significantly reduces even if Algorithm 2 succeeds for some $(a, b) \in AUTH$ since avoiding all possible path generation from a source vertex to target vertex in ARG (FindAllSimplePath() in Line 9) to any extent helps. Otherwise, taking both user and edge attribute value combination into consideration certainly adds overhead to the computation time of Algorithm 1, compared to feasibility detection algorithm in [3].

Let us consider the ARG in Fig. 1 where Range(Relation-type) is changed from $\{Friendship\}$ to $\{Friendship, Parent\}$. Since the "Parent" relation is not present anywhere as edge attribute in the ARG, the effect of introducing a new user with "Parent" relation in ARG remains undetermined. This might happen to any ARG with a particular rule structure as change in relationships or adding a new user may effect the validity of the current rule set. We call this "unrepresented path labels" problem in ARG. The rule structure in this paper compares direct values, the $Rule_{op}$ generated by Algorithm 1 does consider all user and edge attributes, and ARG is static by nature. Thereby, unrepresented path labels does not impact the $Rule_{op}$.

In order to show a comparison with our AReBAC policy language in user to user relationship context, the model presented in [6] is compared as follows:

- By construction, the policy language in this paper does not support inverse relationship and count attribute as in [6].
- The policy language in Def. 4 is unable to count the number of existing paths between access initiator and target users. Another example is, the policy language in Def. 4 is unable to compare attribute value assignments of any two particular users along the path from initiator to target in ARG.
- The policy language in [6] supports the common regular expression feature, wildcard (* means to 0 to any number), optional (? means 0 or 1) notation, and negative path expression, while this paper completely ignores them.

Clearly the AReBAC rule structure presented in this paper is not the most general one. More expressiveness can be added such as in [6] and current feasibility problem statement could be correspondingly reformulated.

4 ARREP Infeasibility Solution

Given an infeasible ARREP instance as in Def. 6, an infeasibility solution basically generates a RuleSet which completes the AReBAC system. Formally, given an infeasible ARREP instance as in Def. 6, an infeasibility solution is said to be exact iff: $(\forall u, v \in U)[checkAccess_{AAR}(u, v) \Leftrightarrow checkAccess_{EAS}(u, v)]$.

In this section, an exact solution to infeasibility in ARREP will be discussed with computational complexity as well as shortcomings. It is accomplished by adding edges to the given ARG as follows:

Definition 7. ***Add relationship edge***
Given an ARREP infeasible instance, Algorithm 1 returns a set of failed authorization pairs, failedAuthPairs. Subsequently, the following steps are used:

1. *It is assumed that,* $\forall ea \in EA.op \notin Range(ea)$.
2. $\forall ea \in EA, Range(ea) \cup := op$, *where* $op \in OP$.
3. *For each* $(a, b) \in failedAuthPairs$, $E := E \cup \{(a,b,op,op,...,op)\}$.
   Note: for each newly added edge, say e,* $\forall ea \in EA.ea(e) = op$.
4. $Rule_{op} := Rule_{op} \vee (\bigwedge_{ea \in EA} ea(e) = op)$, $Rule_{op}$ *is returned by Algorithm 1.*

For example, given the previous infeasible example where Auth = $\{(Bob, Alice)\}$ and ARG as in Fig. 1, an additional relationship edge from Bob to Alice, labeled by the operation op$\in OP$ where op is added to Range(Relation-type), solves the problem. The following theorem proves the correctness of the stated infeasibility correction approach in Def. 7.

Theorem 2. *Def. 7 provides an exact solution to infeasibility in ARREP.*

Proof. As stated, for all $(a, b) \in failedAuthPairs$, adding an edge from vertex a to b in ARG creates a path of length 1. By the checkAccess evaluation presented in Def. 5, all $(a, b) \in failedAuthPairs$ satisfy $(\bigwedge_{ea \in EA} ea(e) = op)$, and therefore, adding a term is sufficient for a operation $op \in OP$. Since it is assumed that, $\forall ea \in EA.op \notin Range(ea)$, therefore, no other $U \times U \setminus failedAuthPairs$ satisfies $(\bigwedge_{ea \in EA} ea(e) = op)$. Hence, the claim is correct.

As stated in Def. 7, the solution adds $|AUTH|$ edges to the ARG at most. Hence, the worst case complexity is linear to $|AUTH|$. However, this solution approach has limitations. For example, less number of additional edges could be used to resolve the infeasibility [3]. Furthermore, there might be cases where it is undesirable to alter the given ARG and Range of attributes at all. We leave considering such cases as future work.

5 Conclusion

This paper provides an insightful discussion regarding attribute-aware ReBAC policy mining. It introduces the ARREP problem and formalizes infeasibility issues in ARREP. A few simple rule optimization technique may reduce the generated rule size. For instance, rule minimization is limited to finding minimal number of path labels in conjunctive terms only. As per Algorithm 1, for a tuple (a,b) in AUTH, the conjunctive term is formed by AND'ing all possible path labels from a to b iff i) Algorithm 2 fails, and ii) the conjunctive term evaluates false for all unauthorized tuples. Instead of using all possible path labels in the conjunctive term of such (a,b), the smallest possible subset (except empty set) of those is used to form the conjunctive term, ensuring that

the minimal size conjunctive evaluates false for all unauthorized tuples. For instance, given ARG in Fig. 1 and AUTH = $\{(Alice, Bob)\}$, i) Algorithm 2 returns FAILURE for (Alice, Bob), ii) there exist two paths, say p1 and p2, from Alice to Bob in ARG where $pathLabel_{att}(p1)$ and $pathLabel_{att}(p2)$ are (Female, Student).(F).(Male,Officer) and (Female, Student).(F).(Male,Student).(F).(Female, Student).(F).(Male,Officer). Without any rule minimization, $Rule_{op}$ generated by Algo. 1 is given by the conjunction of generateRule($pathLabel_{att}(p1)$) and generateRule($pathLabel_{att}(p2)$): (Gender(e.u) = Female ∧ Profession(e.u) = Student ∧ Relation-type(e) = F ∧ Gender(e.v) = Male ∧ Profession(e.v) = Officer) ∧ (Gender(e.u) = Female ∧ Profession(e.u) = Student ∧ Relation-type(e) = F ∧ Gender(e.v) = Male ∧ Profession(e.v) = Student . Gender(e.u) = Male ∧ Profession(e.u) = Student ∧ Relation-type(e) = F ∧ Gender(e.v) = Female ∧ Profession(e.v) = Student . Gender(e.u) = Female ∧ Profession(e.u) = Student ∧ Relation-type(e) = F ∧ Gender(e.v) = Male ∧ Profession(e.v) = Officer). The possible subset of path labels in this case is: either one or both. It is evident that, i) only $pathLabel_{att}(p1)$ is not possible because it is satisfied by unauthorized pair (Cathy, Bob) ii) only $pathLabel_{att}(p2)$ is possible since it is not satisfied by unauthorized pairs. Thereby, $Rule_{op}$ reduces to (Gender(e.u) = Female ∧ Profession(e.u) = Student ∧ Relation-type(e) = F ∧ Gender(e.v) = Male ∧ Profession(e.v) = Student . Gender(e.u) = Male ∧ Profession(e.u) = Student ∧ Relation-type(e) = F ∧ Gender(e.v) = Female ∧ Profession(e.v) = Student . Gender(e.u) = Female ∧ Profession(e.u) = Student ∧ Relation-type(e) = F ∧ Gender(e.v) = Male ∧ Profession(e.v) = Officer).

Acknowledgement. This work is partially supported by NSF CREST Grant HRD-1736209.

References

1. Ahmed, T., Sandhu, R., Park, J.: Classifying and comparing attribute-based and relationship-based access control. In: 7th ACM CODASPY 2017. pp. 59–70
2. Bui, T., Stoller, S.D.: Learning attribute-based and relationship-based access control policies with unknown values. In: Info. Systems Security. pp. 23–44. Springer (2020)
3. Chakraborty, S., Sandhu, R.: Formal analysis of rebac policy mining feasibility. In: Proc. of the 11th ACM CODASPY. pp. 197–207 (2021)
4. Chakraborty, S., Sandhu, R., Krishnan, R.: On the feasibility of attribute-based access control policy mining. In: IRI. IEEE (2019)
5. Chakraborty, S., Sandhu, R., Krishnan, R.: On the feasibility of rbac to abac policy mining: A formal analysis. In: SKM. pp. 147–163. Springer (2019)
6. Cheng, Y., Park, J., Sandhu, R.: Attribute-aware relationship-based access control for online social networks. In: DBSec. pp. 292–306. Springer (2014)
7. Fong, P.W., Siahaan, I.: Relationship-based access control policies and their policy languages. In: Proc. of the 16th ACM SACMAT. p. 51–60. ACM (2011)
8. Hu, V., et al.: Guide to Attribute Based Access Control (ABAC) definition and considerations. NIST Special Publication pp. 162–800 (2014)
9. Rizvi, S.Z.R., Fong, P.W.L.: Efficient authorization of graph-database queries in an attribute-supporting rebac model. ACM Trans. Priv. Secur. **23**(4) (Jul 2020)