Mining Attribute-based Access Control Policies

Gunjan Batra,
PhD, IT
Rutgers Business School,
Newark, NJ

Abstract –*As opposed to Role Based Access Control (RBAC), Attribute-based access control (ABAC) provides a high level of flexibility that promotes security and information sharing. ABAC policy mining algorithms have potential to significantly reduce the cost of migration to ABAC, by partially automating the development of an ABAC policy from information about the existing access-control policy*
and attribute data. This presents a survey of different ABAC policy mining approaches being applied while mining policies from Access Control Lists, Role Based Access Control (RBAC) Policies and Operation Logs with accompanying attribute data. To the best of my knowledge, the approaches presented in the research paper were first ABAC policy mining algorithm for each kind.

1. Introduction

Securing organizational resources is one of the fundamental challenges in today's corporate environment. Organizations use access control mechanisms to mitigate the risk of unauthorized access to their data, resources and systems. Replacing the traditional access control mechanisms such as Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role based Access Control (RBAC), Attribute Based Access Control is gaining a lot of attention from research groups and organizations because of its efficiency, dynamic nature, flexibility and scalability[4]. ABAC policy mining essentially requires identification of the "significant attributes" acceptable to the organization that a user must possess to get access to a particular resource.

ABAC has even solved the problem of 'Role Explosion' [4][5], associated with RBAC [7]. Gartner has predicted that "by 2020, 70% of enterprises will use attribute-based access control…as the dominant mechanism to protect critical assets, up from less than 5% today." ABAC provides a high level of flexibility that promotes information sharing [1]. In an ABAC based system, there is a well-defined set of attributes for users, objects and environment conditions. An attribute defines a particular characteristic of an entity. It also includes a list of possible operations that can be performed in the system. To make authorization decisions, a set of authorization rules (policies) is maintained, where each rule consists of predicates over a subset of user attributes, object attributes and the environmental attributes. A user is allowed to access an object and perform an operation on it in an environment condition, if and only if, the attribute set of the user, the object and environment have a configuration satisfying the rule associated with the requested access.

Whenever a new user with a set of attributes joins the organization, how an ABAC rule (or policy) can be automatically assigned to that user based on the existing rules in the system. This boils down to the problem ABAC policy mining Manual development of ABAC policies can be difficult [6] and expensive [4]. ABAC policy mining algorithms have potential to reduce the cost of ABAC policy development. [1]

2. Approaches Proposed

We surveyed 3 approaches for policy mining of Attribute based access controls (ABAC)

1. Mining Attribute-Based Access Control Policies, 2013
2. Mining Attribute-Based Access Control Policies from Role-Based Policies, 2013
3. Mining Attribute-Based Access Control Policies from Logs, 2014

These approaches presented by Zhongyuan Xu and Scott Stoller present ways for mining ABAC policies from different inputs original policy mining approach defined by Scott Stoller in [1] . The first algorithm aims at mining an ABAC policy from ACLs and attribute data. The second one aims at

mining ABAC policies from RBAC policies and attribute data. The third algorithm uses operation logs and attribute data to mine the ABAC policies.

2.1. Mining Attribute-Based Access Control Policies, 2013 [1]

Algorithm summary – The algorithm iterates over tuples in the given user permission relation. It uses selected tuples as seeds for constructing candidate rules. Then it attempts to generalize each candidate rule to cover additional tuples in the user-permission relation by replacing conjuncts in attribute expressions with constraints. After constructing candidate rules that together cover the entire user permission relation, it attempts to improve the policy by merging and simplifying candidate rules. Finally, it selects the highest-quality candidate rules for inclusion in the generated policy. Also, developed an extension of the algorithm to identify suspected noise in the input

Policy Language

- Given a set U of users and a set A_u of user attributes, user attribute data is represented by a function d_u such that $d_u(u; a)$ is the value of attribute a for user u.
- The set A_u of user attributes can be partitioned into a set $A_{u;\ 1}$ of single valued user attributes which have atomic values, and a set $A_{u;\ m}$ of multi-valued user attributes whose values are sets of atomic values.
- Given a set R of resources and a set A_r of resource attributes, resource attribute data is represented by a function d_r such that $d_r(r; a)$ is the value of attribute a for resource r
- The set A_r of resource attributes can be partitioned into a set $A_{r;\ 1}$ of single-valued resource attributes and a set of $A_{r;\ m}$ of multivalued resource attributes.
- We assume Vals includes a distinguished value ⌊ used to indicate that an attribute's value is unknown. The set of possible values of multivalued attributes is Valm = Set (ValS \ {⌊}) U ⌊, where Set(S) is the powerset of set S.
- A user attribute expression (UAE) is a function e such that, for each user attribute a, e (a) is either the special value T, indicating that e imposes no constraint on the value of attribute a, or a set (interpreted as a disjunction) of possible values of a excluding ⌊ (in other words, a subset of Vals or Valm, depending on whether a is single-valued or multi-valued).
- Example - Suppose $A_{u;\ 1}$ = {dept; position} and $A_{u;\ m}$ = {courses}. The function e_1 with e_1(dept) = {CS} and e_1(position) = {grad; ugrad} and e_1(courses) = {{CS101; CS102}} is a user-attribute expression satisfied by users in the CS department who are either graduate or undergraduate students and whose courses include CS101 and CS102 (and possibly other courses).
- The expression e1 may be written as "dept = CS ∧ position ∈ {ugrad; grad} ∧ courses ⊇ {CS101; CS102}"
- Similarly Resource Attribute Expression RAE is defined
- A user-permission tuple is a tuple ‹u; r; o› containing a user, a resource, and an operation. This tuple means that user u has permission to perform operation o on resource r. A user-permission relation is a set of such tuples.
- A rule is a tuple ‹ e_u;e_r;O;c › where e_u is a user-attribute expression, er is a resource-attribute expression, O is a set of operations, and c is a constraint. For a rule ρ = ‹ e_u;e_r;O;c ›, let uae(ρ) = e_u, rae(ρ) = e_r, ops(ρ) = O, and con(ρ) = c. For example, the rule ‹true, type=task ∧ proprietary=false, {read, request}, projects ∋ project ∧ expertise expertise› used in our project management case study can be interpreted as
- "A user working on a project can read and request to work on a non-proprietary task whose required areas of expertise are among his/her areas of expertise." User u, resource r, and operation o satisfy a rule, denoted ‹u; r; o› ⊨ ρ, if u ⊨ uae(ρ) ∧ r ⊨ rae(ρ) ∧ o ops(ρ) ∧ ‹u; r› ⊨ con(ρ).
- An ABAC policy is a tuple ‹U;R; Op;Au;Ar; du; dr;Rules ›, where U, R, Au, Ar, du, and dr are as described above, Op is a set of operations, and Rules is a set of rules.
- The user-permission relation induced by a rule ρ is [[ρ]] = {‹ u; r; o › ϵ U X R X Op | ‹ u; r; o › ρ. Note that U, R, du, and dr are implicit arguments to [[ρ]].
- The user-permission relation induced by a policy with the above form is [[π]] = ∪ ρ∈Rules [[ρ]].

Problem Definition

- **An ABAC policy** is a tuple ‹U;R; Op;A_u;A_r; d_u; d_r;Rules ›, where U, R, A_u, A_r, d_u, and d_r are as described above, Op is a set of operations, and Rules is a set of rules**.**
- **An access control list (ACL)** policy is a tuple ⟨U; R; Op; UP_0 ⟩, where U is a set of users, R is a set of resources, Op is a set of operations, and $UP_0 \subseteq$ U X R X Op is a user-permission relation, obtained from the union of the access control lists.
- An **ABAC policy π is consistent with an ACL policy** ⟨U; P; Op; UP_0 ⟩if they have the same sets of users, resource, and operations and **[[π]] = UP_0.**
- An ABAC policy can be consistent with ACL policy by creating a separate rule corresponding to each user-permission tuple in the ACL policy.
- The **ABAC policy mining problem** is:
  Given an ACL policy π_0 = ⟨U;R; Op; UP_0 ⟩, user attributes A_u, resource attributes A_r, user attribute data d_u, resource attribute data d_r, and a policy quality metric Q_{pol}, find a set Rules of rules such that the ABAC policy π = ‹U;R; Op;A_u;A_r; d_u; d_r;Rules› that
  (1) is consistent with π_0
  (2) Uses uid only when necessary,
  (3) Uses rid only when necessary, and
  (4) Maximum policy quality metric – Q_{pol}

Policy Quality Metric -

- The policy quality metric that our algorithm aims to optimize is weighted structural complexity (WSC) [8] shown in Fig. 1, a generalization of policy size. This is consistent with usability studies of access control rules, which conclude that more concise policies are more manageable [6]. Informally, the WSC of an ABAC policy is **a weighted sum of the number of elements in the policy**. Formally, the WSC of an ABAC policy with rules *Rules* is WSC(π) = WSC(Rules), defined by Figure 1[1]

$$\begin{aligned}
\mathrm{WSC}(e) &= \sum_{a \in \mathrm{attr}_1(e)} |e(a)| + \sum_{a \in \mathrm{attr}_m(e), s \in e(a)} |s| \\
\mathrm{WSC}(\langle e_u, e_r, O, c \rangle) &= w_1 \mathrm{WSC}(e_u) + w_2 \mathrm{WSC}(e_r) \\
&\quad + w_3 |O| + w_4 |c| \\
\mathrm{WSC}(Rules) &= \sum_{\rho \in Rules} \mathrm{WSC}(\rho),
\end{aligned}$$

where $|s|$ is the cardinality of set s, and the w_i are user-specified weights.

Figure 1: WSC [1]

The third equation is the sum of all the WSC of each rule ρ which is equal to e_u, e_r , O , c

This third equation is calculated with help of second equation which is in turn calculated with help of first.

Based on second eq. we can say that WSC is weighted sum of the number of elements in a policy.

Policy Mining Algorithm Pseudocode(Figure 1)

```
    // Rules is the set of candidate rules
1:  Rules = ∅
    // uncovUP contains user-permission tuples in UP_0
    // that are not covered by Rules
2:  uncovUP = UP_0.copy()
3:  while ¬uncovUP.isEmpty()
        // Select an uncovered user-permission tuple.
4:      ⟨u, r, o⟩ = some tuple in uncovUP
5:      cc = candidateConstraint(r, u)
        // s_u contains users with permission ⟨r, o⟩ and
        // that have the same candidate constraint for r as u
6:      s_u = {u' ∈ U | ⟨u', r, o⟩ ∈ UP_0
7:                    ∧ candidateConstraint(r, u') = cc}
8:      addCandidateRule(s_u, {r}, {o}, cc, uncovUP, Rules)
        // s_o is set of operations that u can apply to r
9:      s_o = {o' ∈ Op | ⟨u, r, o'⟩ ∈ UP_0}
10:     addCandidateRule({u}, {r}, s_o, cc, uncovUP, Rules)
11: end while
    // Repeatedly merge and simplify rules, until
    // this has no effect
12: mergeRules(Rules)
13: while simplifyRules(Rules) && mergeRules(Rules)
14:     skip
15: end while
    // Select high quality rules into final result Rules'.
16: Rules' = ∅
17: Repeatedly select the highest quality rules from
    Rules to Rules' until ∑_{ρ∈Rules'} [[ρ]] = UP_0,
    using UP_0 \ [[Rules']] as second argument to Q_rul
18: return Rules'
```

Figure 2: Pseudocode for algorithm [1]

```
    function addCandidateRule(s_u, s_r, s_o, cc, uncovUP, Rules)
    // Construct a rule ρ that covers user-permission
    // tuples {⟨u, r, o⟩ | u ∈ s_u ∧ r ∈ s_r ∧ o ∈ s_o}.
1:  e_u = computeUAE(s_u, U)
2:  e_r = computeRAE(s_r, R)
3:  ρ = ⟨e_u, e_r, s_o, ∅⟩
4:  ρ' = generalizeRule(ρ, cc, uncovUP, Rules)
5:  Rules.add(ρ')
6:  uncovUP.removeAll([[ρ']])
```

Figure 3 : Function AddCandidateRule [1]

```
    function generalizeRule(ρ, cc, uncovUP, Rules)
    // ρbest is highest-quality generalization of ρ
1:  ρbest ← ρ
    // cc' contains formulas from cc that lead to valid
    // generalizations of ρ.
2:  cc' ← new Vector()
3:  // gen[i] is a generalization of ρ using cc'[i]
4:  gen ← new Vector()
    // find formulas in cc that lead to valid
    // generalizations of ρ.
5:  for f in cc
        // try to generalize ρ by adding f and elimi-
        // nating conjuncts for both attributes used in f.
6:      ρ' ← ⟨uae(ρ)[uAttr(f) ↦ ⊤], rae(ρ)[rAttr(f) ↦ ⊤],
7:            ops(ρ), con(ρ) ∪ {f}⟩
        // check if ⟦ρ'⟧ is a valid rule
8:      if ⟦ρ'⟧ ⊆ UP0
9:          cc'.add(f)
10:         gen.add(ρ')
11:     else
            // try to generalize ρ by adding f and elimi
            // nating conjunct for one user attribute used in f
12:         ρ' = ⟨uae(ρ)[uAttr(f) ↦ ⊤], rae(ρ),
13:               ops(ρ), con(ρ) ∪ {f}⟩
14:         if ⟦ρ'⟧ ⊆ UP0
15:             cc'.add(f)
16:             gen.add(ρ')
17:         else
        // try to generalize ρ by adding f and elimi-
        // nating conjunct for one resource attribute used in f.
18:             ρ' = ⟨uae(ρ), rae(ρ)[rAttr(f) ↦ ⊤],
19:                   ops(ρ), con(ρ) ∪ {f}⟩
20:             if ⟦ρ'⟧ ⊆ UP0
21:                 cc'.add(f)
22:                 gen.add(ρ')
23:             end if
24:         end if
25:     end if
26: end for
27: for i ← 1 to cc'.length
28:     // try to further generalize gen[i]
29:     ρ'' ← generalizeRule(gen[i], cc'[i+1..], uncovUP',
30:                          Rules)
31:     if Qrul(ρ'', uncovUP) > Qrul(ρbest, uncovUP)
32:         ρbest = ρ''
33:     end if
34: end for
35: return ρbest
```

Fig. 3. Generalize rule ρ by adding some formulas from cc to its constraint and eliminating conjuncts for attributes used in those formulas. $f[x \mapsto y]$ denotes a copy of function f modified so that $f(x) = y$. $a[i..]$ denotes the suffix of array a starting at index i.

Figure 4: Function generalize rule [1]

```
    function mergeRules(Rules)
1:  // Remove redundant rules
2:  rdtRules = {ρ ∈ Rules | ∃ ρ' ∈ Rules \ {ρ}. ⟦ρ⟧ ⊆ ⟦ρ'⟧}
3:  Rules.removeAll(rdtRules)
4:  // Merge rules
5:  workSet = {(ρ1, ρ2) | ρ1 ∈ Rules ∧ ρ2 ∈ Rules
                ∧ ρ1 ≠ ρ2 ∧ con(ρ1) = con(ρ2)}
6:  while not(workSet.empty())
        // Remove an arbitrary element of the workset
7:      (ρ1, ρ2) = workSet.remove()
8:      ρmerge = ⟨uae(ρ1) ∪ uae(ρ2), rae(ρ1) ∪ rae(ρ2),
                  ops(ρ1) ∪ ops(ρ2), con(ρ1)⟩
9:      if ⟦ρmerge⟧ ⊆ UP0
            // The merged rule is valid. Add it to Rules,
            // and remove rules that became redundant.
10:         rdtRules = {ρ ∈ Rules | ⟦ρ⟧ ⊂ ⟦ρmerge⟧}
11:         Rules.removeAll(rdtRules)
12:         workSet.removeAll({(ρ1, ρ2) ∈ workSet |
                        ρ1 ∈ rdtRules ∨ ρ2 ∈ rdtRules})
13:         workSet.addAll({(ρmerge, ρ) | ρ ∈ Rules
                                ∧ con(ρ) = con(ρmerge)})
14:         Rules.add(ρmerge)
15:     end if
16: end while
17: return true if any rules were merged
```

Fig. 4. Merge pairs of rules in $Rules$, when possible, to reduce the WSC of $Rules$. (a, b) denotes an unordered pair with components a and b. The union $e = e_1 \cup e_2$ of attribute expressions e_1 and e_2 over the same set A of attributes is defined by: for all attributes a in A, if $e_1(a) = \top$ or $e_2(a) = \top$ then $e(a) = \top$ otherwise $e(a) = e_1(a) \cup e_2(a)$.

Figure 5: Function mergeRules [1]

As shown in Fig. 2The function of the policy mining algorithm works by covering all the uncovered tuples in the user permission relation, (uncovUP =the tuples in UP_0) one by one, until they are covered by Rules

After selecting a user permission tuple in <u,r,o>, the function candidate constraint(r,u) returns a set containing all the atomic constraints between resource r and user u. s_u contains users with permission <r; o> and that have the same candidate constraint for r as u and s_o is set of operations that u can apply to Function addCandidateRule (Fig3) constructs the rule ρ that covers the user-permission tuples where user u, resource r and operation o are such that user belongs to S_u and resource belongs to S_r (only one resource r in example) and operations belong to S_o (only one operation o in example) It calls 3 functions: A) The function computeUAE(s;U) computes a userattribute expression eu that characterizes the set s of users. B) computeRAE(s;R) computes a resource-attribute expression that characterizes the set s of resources. C) generalize rule (ρ;cc;uncovUP;Rules) in Fig 4 attempts to generalize rule ρ by adding some of the atomic constraints f in cc to ρ and eliminating the conjuncts of the user attribute expression and the resource attribute expression corresponding to the attributes used in f. The function mergeRules(Rules) in Figure 5 tries to reduce the WSC of Rules by removing redundant rules and merging pairs of rules. Informally, rules 1 and 2 are merged by taking, for each attribute, the union of the conjuncts in 1 and 2 for that attribute. If the resulting rule merge is valid, merge is added to Rules, and 1 and 2 and any other rules that are now redundant are removed from Rules.

The function simplifyRules(Rules) attempts to simplify all of the rules in Rules.

This algorithm incorporates heuristics and does not guarantee to generate policy with minimal WSC.

Noise detection

Noise = Over assignments + Underassignments

An over-assignment is when a permission is inappropriately granted to a user.

An under-assignment is when a user lacks a permission that he or she should be granted.

- To detect over-assignments, we introduce a rule quality threshold T
    - This rule quality metric is the first component of the metric used in the loop in

that constructs Rules'; thus, T is a threshold on the value of Qrul(ρ; uncovUP)
  - The rules with quality less than or equal to T form a suffix of the sequence of rules added to Rules'
  - The extended algorithm reports as suspected over-assignments the user-permission tuples covered in Rules' only by rules with quality less than or equal to T , and then it removes rules with quality less than or equal to from Rules'
  - Adjustment of T is guided by the user
- To detect under-assignments, we look for rules that are almost valid, i.e., rules that would be valid if a relatively small number of tuples were added to UP0
  - A rule is α almost valid if the fraction of invalid user-permission tuples in [[ρ]] is at most α, i.e., $|[[\rho]] \setminus UP0| \div |[[\rho]]| \leq \alpha$
  - The extended algorithm reports $U_{\rho \in Rules}$'[[ρ]]\UP0 as the set of suspected under-assignments, and (as usual) it returns Rules' as the generated policy.
  - Adjustment of α is guided by the user

Evaluation

The method used for evaluation of ABAC policies is using an ABAC policy (including attribute data), generate an equivalent ACL policy and attribute data. Then, run the algorithm on the resulting ACL policies and attribute data, and compare the mined ABAC policy with the original ABAC policy.

They developed sample policies (University Sample policy, Health Care Sample policy, Project Management Sample policy) that are similar to policies that might be found in the real-world case studies. They consisted of rules and manually written attribute dataset.

The evaluation was done on Sample polices and Synthetic Policies. For the Sample Policies, the attribute data chosen was both manual and synthetic.

Example

Data – The following example presents a sample policy of University data:

- The statement rule(uae; pae;ops; con) defines a rule; the four components of this statement correspond directly to the four components of a rule
  - Uae=User attribute expression
  - Rae=resource attribute expression
  - Ops=Operations
  - Con=Constraints
- User attributes include :
  1. Position (applicant, student, faculty, or staff),
  2. department (the user's department),
  3. crsTaken (set of courses taken by a student),
  4. crsTaught (set of courses for which the user is the instructor (if the user is a faculty) or the TA (if the user is a student), and
  5. isChair (true if the user is the chair of his/her department).
- Resource attributes include :
  1. type (application, gradebook, roster, or transcript),
  2. crs (the course a gradebook or roster is for, for those resource types)
  3. student (the student whose transcript or application this is, for type=transcript or type=application)
  4. department (the department the course is in, for type 2 fgradebook; rosterg; the student's major department, for type=transcript).

The rules for the sample policy are written manually with some knowledge. The dataset for this sample policy is written manually and generated synthetically.

- Manual - few instances of each type of user and resource, two academic departments, a few faculty, a few gradebooks,, several students, a few staff in each of 2 administrative departments(admissions office and registrar)
- Synthetic – A series of synthetic attribute datasets was generated - parameterized by number of academic department.

The policy rules and illustrative UserAttrib and ResourceAttrib are shown in Figures 6 and 7 resp.

```
// Rules for Gradebooks
// A user can read his/her own scores in gradebooks
// for courses he/she has taken.
rule(; type=gradebook; readMyScores; crsTaken ] crs)
// A user (the instructor or TA) can add scores and
// read scores in the gradebook for courses he/she
// is teaching.
rule(; type=gradebook; {addScore, readScore};
      crsTaught ] crs;)
// The instructor for a course (i.e., a faculty teaching
// the course) can change scores and assign grades in
// the gradebook for that course.
rule(position=faculty; type=gradebook;
      {changeScore, assignGrade}; crsTaught ] crs)

// Rules for Rosters
// A user in registrar's office can read and modify all
// rosters.
rule(department=registrar; type=roster; {read, write}; )
// The instructor for a course (i.e., a faculty teaching
// the course) can read the course roster.
rule(position=faculty; type=roster; {read};
      crsTaught ] crs)
```

Figure 6 : Policy Rules [1]

```
// Rules for Transcripts
// A user can read his/her own transcript.
rule(; type=transcript; {read}; uid=student)
// The chair of a department can read the transcripts
// of all students in that department.
rule(isChair=true; type=transcript; {read};
     department=department)
// A user in the registrar's office can read every
// student's transcript.
rule(department=registrar; type=transcript; {read}; )

// Rules for Applications for Admission
// A user can check the status of his/her own application.
rule(; type=application; {checkStatus}; uid=student)

// A user in the admissions office can read, and
// update the status of, every application.
rule(department=admissions; type=application;
     {read, setStatus}; )

// An illustrative user attribute statement.
userAttrib(csFac2, position=faculty, department=cs,
           crsTaught={cs601})
// An illustrative resource attribute statement.
resourceAttrib(cs601gradebook, department=cs,
               crs=cs601, type=gradebook)
```

Figure 7 : Policy Rules [1]

Processing of Algorithm

Figure 8 illustrates the processing of the user permission tuple t = <csFac2; addScore; cs601gradebook> selected as a seed (i.e., selected in line 4 of Figure 8), in a smaller version of the university sample policy containing only one rule, second rule. Attribute data for user csFac2 and resource cs601gradebook have been mentioned before. The edge from t to cc labeled "candidateConstraint" represents the call to candidateConstraint, which returns the set of atomic constraints that hold between csFac2 and cs601gradebook; these constraints are shown in the box labeled cc. The two boxes labeled "addCandidateRule" represent the two calls to addCandidateRule. Internal details are shown for the first call but elided for the second call. The edges from t to eu and from t to er represent the calls in addCandidateRule to computeUAE and computeRAE, respectively. The call to computeUAE returns a user-attribute expression eu that characterizes the set su containing users u0 with permission addScore; cs601gradebooki and such that candidateConstraint(cs601gradebook; u0) = cc. The call to computeRAE returns a resource-attribute expression that characterizes fcs601gradebookg. The set of operations considered in this call to addCandidateRule is simply so = addScore. The call to generalizeRule generates a candidate rule 1 by assigning eu, er and so to the first three components of 1, and adding the two atomic constraints in cc to 1 and eliminating the conjuncts in eu and er corresponding to the attributes mentioned in cc. Similarly, the second call to addCandidateRule generates another candidate rule 2. The call to mergeRules merges 1 and 2 to form 3, which is simplified by the call to simplifyRules to produce a simplified rule 4, which is added to candidate rule set Rules0.

In summary, the algorithm was very effective for all three sample policies. There were only small differences between the original and mined policies if no attributes are declared unremovable, and the original and mined policies were identical if the resource-type attribute is declared unremovable.

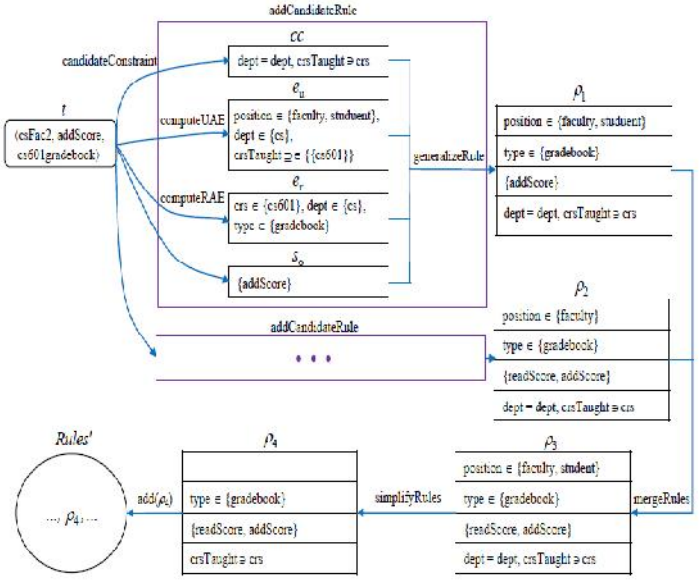

Figure 8: Processing of Algorithm [1]

2.2. Mining Attribute based Access Control policies from Logs

This paper presents an algorithm for mining ABAC policies from logs and attribute data. To the best of our knowledge, it is the first algorithm for this problem. It is based on the algorithm for mining ABAC policies from ACLs [1].

The main challenge with logs is that they give incomplete information about granted permissions. They provide only a lower bound on the entitlements. Therefore, the generated policy needs

to include over-assignments, i.e., entitlements not reflected in the logs.

Similar to [1], this algorithm iterates over tuples in the user-permission relation extracted from the log, uses selected tuples as seeds for constructing candidate rules, and attempts to generalize each candidate rule to cover additional tuples in the user-permission relation by replacing conjuncts in attribute expressions with constraints.

After constructing candidate rules that together cover the entire user-permission relation, it attempts to improve the policy by merging and simplifying candidate rules.

Finally, it selects the highest quality candidate rules for inclusion in the generated policy.

Several changes were done to algorithm for mining ABAC policies from ACLs to adapt it to mining from logs. When the algorithm generalizes, merges, or simplifies rules, it discards candidate rules that are invalid, i.e., that produce over-assignments. Those parts of the algorithm were modified to consider those candidate rules, because over-assignments must be permitted. For those candidate rules, they introduced generalized notions of rule quality and policy quality that quantify a trade-off between the number of over assignments and other aspects of quality.

We consider a metric that includes the normalized number of over-assignments in a weighted sum, a frequency-sensitive variant that assigns higher quality to rules that cover more frequently used entitlements, along the lines of [6], and a metric based on a theory quality metric in inductive logic programming [8, 9]

ABAC policy Language

The ABAC policy language used in this research paper is adapted from [1]

Problem Definition

An operation log is a sequence of logs. The log entry e is defined as a tuple <u, r, o, t> where u U is a user, r R is a resource, o Op is an operation, and t is a timestamp.

The user-permission relation induced by an operation log L is

UP(L) = {u, r, o | t.< u, r, o, t> L}.

The **input to the ABAC-from-logs policy mining problem** is a tuple I = <U,R, Op ,Au, Ar, du, dr, L>, where U is a set of users, R is a set of resources, Op is a set of operations, Au is a set of user attributes, Ar is a set of resource attributes, du is user attribute data, dr is resource attribute data, and L is an operation log, such that the users, resources, and operations that appear in L are subsets of U, R, and Op, respectively.

The problem is defined assuming that attribute data does not change during the time covered by the log.

The problem is to find a set of rules *Rules* such that the ABAC policy π = <U, R, Op, A_u, A_r, d_u, d_r, Rules> maximizes a suitable policy quality metric.

The policy quality metric reflecting **the size** and **meaning** of the policy is defined as a weighted sum of the two aspects:

$Qpol(\pi, L) = WSC(\pi) + wo \cdot |[[\pi]] \setminus UP(L)| / |U|$

where WSC is weighted structural complexity and the policy over-assignment weight wo is a user-specified weight for over assignments

- WSC reflects the **size**. The smaller policies are considered to have higher quality. More concise policies are more manageable. WSC of an ABAC policy is a weighted sum of the number of elements in the policy.
- The WSC of a rule is a weighted sum of the WSCs of its components, namely, $WSC(<e_u, e_r, O, c>) = w_1 WSC(e_u) + w_2 WSC(e_r) + w_3 WSC(O) + w_4 WSC(c)$, where the w_i are user-specified weights. the WSC of an attribute expression is the number of atomic values that appear in it, the WSC of an operation set is the number of operations in it, the WSC of a constraint is the number of atomic constraints in it,
- The **meaning** $[[\pi]]$ of the ABAC policy is taken into account by considering the differences from UP(L), which consist of over-assignments and under assignments.
- The over-assignments are $[[\pi]] \setminus UP(L)$. The under-assignments are $UP(L) \setminus [[\pi]]$.
- Since logs provide only a lower-bound on the actual user-permission relation (a.k.a entitlements), it is necessary to allow some over-assignments, but not too many. Allowing

under-assignments is beneficial if the logs might contain noise, in the form of log entries representing uses of permissions that should not be granted, because it reduces the amount of such noise that gets propagated into the policy.

Algorithm

The algorithm steps:

The algorithm is based on the algorithm for mining ABAC policies from ACLs and attribute data in [1]

Top-level pseudocode is in Figure 9.

candidateConstraint(r, u) - returns a set containing all of the atomic constraints that hold between resource r and user u.

The function addCandRule(s_u, s_r, s_o, cc, uncovUP, Rules) in Fig 11 -

1. computeUAE(s,U) to compute a user-attribute expression e_u that characterizes set s_u of users. Preference is given to attribute expressions that do not use uid, since attribute-based policies are generally preferable to identitybased policies, even when they have higher WSC, because attribute-based generalize better
2. computeRAE(s,R) to compute a resource-attribute expression e_r that characterizes s_r of resources

generalizeRule(ρ, cc, uncovUP, Rules) – generalize rule ρ = <e_u, e_r, s_o, ∅> to ρ' and adds ρ' to candidate rule set Rules. It generalizes rule ρ by adding some of the atomic constraints in cc to ρ and eliminates the conjuncts of the user attribute expression and/or the resource attribute expression corresponding to the attributes used in those constraints, i.e., mapping those attributes to T. We call a rule obtained in this way a generalization of ρ. generalizeRule(ρ, cc, uncovUP, Rules) returns the generalization ρ' of ρ with the best quality according to a given rule quality metric. Note that ρ' may cover tuples that are already covered (i.e., are in UP i.e. rules whose meanings overlap.

A rule quality metric is a function Qrul(ρ,UP) that maps a rule ρ to a totally ordered set, with the ordering chosen so that larger values indicate high quality. The second argument UP is a set of user-permission tuples. Our rule quality metric assigns higher quality to rules that cover more currently uncovered userpermission tuples and have smaller size, with an additional term that imposes a penalty for over-assignments, measured as a fraction of the number of userpermission tuples covered by the rule, and with a weight specified by a parameter w'o, called the rule over-assignment weight.

Qrul(ρ,UP) = | [[ρ]] ∩ UP||ρ| × (1 – w'o× |overAssign(ρ)|| [[ρ]] |).

In generalizeRule, uncovUP is the second argument to Qrul, so [[ρ]] ∩ UP is the set of user-permission tuples in UP0 that are covered by ρ and not covered by rules already in the policy. The loop over i near the end of the pseudocode for generalizeRule considers all possibilities for the first atomic constraint in cc that gets added to the constraint of ρ. The function calls itself recursively to determine the subsequent atomic constraints in c that get added to the constraint.

The function mergeRules(Rules) in Figure 10 attempts to improve the quality of Rules by removing redundant rules and merging pairs of rules. A rule ρ in Rules is redundant if Rules contains another rule ρ such that every userpermission tuple in UP0 that is in [[ρ]] is also in [[ρ]]. Informally, rules ρ1 and ρ2 are merged by taking, for each attribute, the union of the conjuncts in ρ1 and ρ2 for that attribute. If adding the resulting rule ρmrg to the policy and removing rules (including ρ1 and ρ2) that become redundant improves policy quality and does not introduce over-assignments where none existed before, then ρmrg is added to Rules, and the redundant rules are removed from Rules. As optimizations (in the implementation, not reflected in the pseudocode), meanings of rules are cached, and policy quality is computed incrementally. mergeRules(Rules) updates its argument Rules in place, and it returns a Boolean indicating whether any rules were merged.

The function simplifyRules(Rules) attempts to simplify all of the rules in Rules.

```
// Rules is the set of candidate rules
Rules = ∅
// uncovUP contains user-permission tuples
// in UP_0 that are not covered by Rules
uncovUP = UP_0.copy()
while ¬uncovUP.isEmpty()
  // Select an uncovered tuple as a "seed".
  ⟨u, r, o⟩ = some tuple in uncovUP
  cc = candidateConstraint(r, u)
  // s_u contains users with permission ⟨r, o⟩
  // and that have the same candidate
  // constraint for r as u
  s_u = {u' ∈ U | ⟨u', r, o⟩ ∈ UP_0
            ∧ candidateConstraint(r, u') = cc}
  addCandRule(s_u, {r}, {o}, cc, uncovUP, Rules)
  // s_o is set of operations that u can apply to r
  s_o = {o' ∈ Op | ⟨u, r, o'⟩ ∈ UP_0}
  addCandRule({u}, {r}, s_o, cc, uncovUP, Rules)
end while
```

Figure 9 : Pseudocode [2]

```
// Repeatedly merge and simplify
// rules, until this has no effect
mergeRules(Rules)
while simplifyRules(Rules)
        && mergeRules(Rules)
  skip
end while
// Select high quality rules into Rules'.
Rules' = ∅
Repeatedly move highest-quality rule
from Rules to Rules' until
∑_{ρ∈Rules'} [[ρ]] ⊇ UP_0, using
UP_0 \ [[Rules']] as second argument to
Q_rul, and discarding a rule if it does
not cover any tuples in UP_0 currently
uncovered by Rules'.
return Rules'
```

Figure 10 : Function mergeRules [2]

```
function addCandRule(s_u, s_r, s_o, cc, uncovUP, Rules)
// Construct a rule ρ that covers user-perm. tuples {⟨u, r, o⟩ | u ∈ s_u ∧ r ∈ s_r ∧ o ∈ s_o}.
e_u = computeUAE(s_u, U); e_r = computeRAE(s_r, R); ρ = ⟨e_u, e_r, s_o, ∅⟩
ρ' = generalizeRule(ρ, cc, uncovUP, Rules); Rules.add(ρ'); uncovUP.removeAll([ρ'])
```

Figure 11 : Function addCandRule [2]

Example

The university case study example is discussed below:

Using a single rule ρ0 =< true, type {gradebook}, {addScore, readScore}, crsTaught ɜ crs > and all of the attribute data from the full case study, except attribute data for gradebooks for courses other than cs601.

Considering an operation log L containing three entries: {csFac2, cs601gradebook, addScore, t1
csFac2, cs601gradebook, readScore, t2
csStu3, cs601gradebook, addScore, t3}.

User csFac2 is a faculty in the computer science department who is teaching cs601; User attributes are position = faculty, dept = cs, and crsTaught ={cs601}.

User csStu3 is a CS student who is a TA of cs601; User attributes are position =student, dept = cs, and crsTaught = {cs601}.

Resource cs601gradebook is a resource with resource attributes type = gradebook, dept = cs, and crs = cs601.

The algorithm selects user-permission tuple : csFac2, cs601gradebook, addScore as the first seed

It calls function:

candidateConstraint to compute the set of atomic constraints that hold between csFac2 and cs601gradebook; the result is cc = {dept = dept, crsTaught crs}.

addCandRule is called twice to compute candidate rules.

- The first call to addCandRule calls computeUAE to compute a UAE eu that characterizes the set su containing users with permission addScore, cs601gradebook and with the same candidate constraint as csFac2 for cs601gradebook; the result is
  eu = (position {faculty, student} dept {cs} crsTaught {{cs601}}).
- addCandRule also calls computeRAE to compute a resource-attribute expression that characterizes cs601gradebook}; the result is
  er = (crs {cs601} dept {cs} type {gradebook}).

- The set of operations considered in this call to addCandRule is simply so = {addScore}. addCandRule then calls generalizeRule, which generates a candidate rule ρ1 which initially has eu, er and so in the first three components, and then atomic constraints in cc are added to ρ1, and conjuncts in eu and er for attributes used in cc are eliminated; the result is
  ρ1 =<position {faculty, student}, type {gradebook}, {addScore}, dept = dept crsTaught ∋ crs>, which also covers the third log entry.
- The second time addCandRule is called it generates a candidate rule
  ρ2 = <position {faculty}, type {gradebook}, {addScore, readScore}, dept = dept crsTaught ∋ crs>, which also covers the second log entry.
- All of UP(L) is covered, so our algorithm calls mergeRules, which attempts to merge ρ1 and ρ2 into rule
  ρ3=< position {faculty, student}, type {gradebook}, {addScore, readScore}, dept = dept crsTaught ∋ crs>.
  ρ3 is discarded because it introduces an over-assignment while ρ1 and ρ2 do not.
- simplifyRules is called, which first simplifies ρ1 and ρ2 to ρ1' and ρ2', respectively, and then eliminates ρ1' because it covers a subset of the tuples covered by ρ2'. The final result is ρ2', which is identical to the rule ρ0 in the original policy.

Evaluation

They evaluated the policy mining algorithm on synthetic operation logs generated from ABAC policies (some handwritten and some synthetic) and probability distributions characterizing the frequency of actions. Hence, they could evaluate the effectiveness of our algorithm by comparing the mined policies with the original ABAC policies.

Results show algorithm's effectiveness even while the log reflects a fraction of the entitlements. Although the original (desired) ABAC policy is not reconstructed perfectly from the log, the mined policy is sufficiently similar to it that the mined policy would be very useful as a starting point for policy administrators tasked with developing that ABAC policy.

2.3. Mining Attribute-Based Access Control Policies from RBAC Policies, 2013

Challenge with 2.1 - In [1], it is mentioned that the algorithm can be used to mine an ABAC policy from an RBAC policy and attribute data, by expanding the RBAC policy into ACLs. And adding a "role" attribute to the attribute data (to avoid information loss), and then applying the algorithm. However, the approach has many limitations. Firstly, the ABAC policy generated by this method might not have the most desired structure, because the structure of the RBAC policy is not used to guide the structure of the ABAC policy. Secondly, we notice that often the available attribute data is insufficient and using this algorithm [1] we cannot substitute role membership information for unavailable attribute information. This will lead to lower-level policies that use user identity instead of role membership information.

While developing the methodology, the authors realized that they did want to get a 1-to-1 correspondence between roles and rules; i.e., for each role r, the mined policy has a rule that covers the same user-permission tuples. However, becomes a strict requirement as some roles cannot be expressed as a single rule and it is often desirable to express multiple related roles by a single rule. To relax this requirement, they first, split the given roles, so that each role's set of assigned permissions is the Cartesian product of a set of resources and a set of operations, and we require a correspondence between the resulting split roles and the mined rules. Then, they allow multiple roles to correspond to a single rule

As per our knowledge, the algorithm discussed in the paper [3] for mining ABAC from RBAC policies and attribute data is the first problem definition and algorithm of its kind.

RBAC Policy Language -

An *RBAC policy* is a tuple ⟨*U*, *Res*, *Op*, *Roles*, *UA*, *PA*, *RH*⟩, where *U* is a set of users, *Res* is a set of resources, *Op* is a set of operations, *Roles* is a set of roles, $UA \subseteq U \times Roles$ is the user-role assignment, $PA \subseteq Roles \times Perm$ is the permissionrole assignment, and the role hierarchy *RH* is an acyclic transitive binary relation on roles. A *permission* is a pair containing a resource and an operation, and $Perm = Res \times Op$. A tuple ⟨*r*, *r'*⟩ in *RH* means that *r* is junior to *r'* (or, equivalently, *r'* is senior to *r*). This means that *r* inherits members from *r'*, and *r'*

inherits permissions from *r*. The *authorized users* of a role include the role's directly assigned users and its inherited users. The *authorized permissions* of a role are defined similarly. These ideas are expressed in the equations below.

asgndU(*r*)={*u* *U* | ⟨*u*, *r*⟩ *UA*}

asgndP(*r*)={*p* *Perm* | ⟨*r*, *p*⟩ *PA*}

ancestors(*r*)={*r'* *Roles* | ⟨ *r*, *r'*⟩ *RH*}

descendants(*r*)={*r'* *Roles* | ⟨ *r'*, *r*⟩ *RH*}

authU(*r*)= asgndU(*r*) $_{r' \ ancestors(r)}$ asgndU(*r'*)

authP(*r*)= asgndP(*r*) $_{r' \ descendants(r)}$ asgndP(*r'*)

The user-permission assignment induced by a role *r* and an RBAC policy π with the above form are defined by authUP(*r*) = authU(*r*) × authP(*r*) and M(π) = $_{r \ Roles}$ authUP(*r*), respectively.

ABAC Policy Language

- This has been adapted mostly from ABAC Policy Language of [1]
- An ABAC policy is a tuple ⟨U, Res, Op, Au, Ar, du, dr, Rules⟩ where U, Res, Au, Ar, du and dr are as described in the policy, Op is a set of operations, and Rules is a set of rules.
- The user-permission relation induced by a rule ρ is M(ρ) = {⟨u, ⟨r, o⟩⟩∈U×Res×Op | ⟨u, ⟨r, o⟩⟩ |= ρ}. Note that U, Res, du and dr are implicit arguments to M(ρ).
- The user-permission relation induced by a policy π with the above form is M(π) =$U_{ρ∈Rules}$ M(ρ) .

Problem Definition

We need to mine an ABAC policy that is semantically consistent with a given RBAC policy and preserves the
structure of the RBAC policy.

RBAC policy is semantically consistent with
an ABAC policy π if M($π_{RBAC}$) = M(π)..

Given a set P of permissions, P is expressed as a sum (union) of Cartesian products. Let ops(P) be the set of operations that appear in P. Let resources(o, P) be the set of resources associated with o in P, i.e., {r∈Res | ⟨r, o⟩ ∈P}.

Two operations are equivalent if they are associated with the same resources in P, i.e., o ≡ $_P$o' iff resources(o, P) = resources(o', P). Let S be a partition of ops(P) containing the equivalence classes of O with respect to ≡ $_P$. Define SOP(P) =$U_{O∈S}$ {⟨resources(O), O⟩ }, where resources(O) is the set of resources associated with any operation in O (by definition, all operations in O are associated with the same resources). Note that P=$U_{< R,O>∈SOP(P)}$ R×O .

Given an RBAC policy $π_{RBAC}$ = ⟨U, Res, Op, Roles, UA, PA, RH⟩, the sum-of-products policy SOP ($π_{RBAC}$) is ⟨U, Res, Op, Roles', UA', PA', RH'⟩, where Roles' ,UA' , PA', RH' are defined basis the sum of products policy.

Given an RBAC policy $π_{RBAC}$ = ⟨U, Res, Op, Roles, UA, PA, RH⟩ and an ABAC policy π = ⟨U, Res, Op, Au, Ar, du, dr, Rules⟩, a structural correspondence between $π_{RBAC}$ and π is an onto function κ from the roles in SOP($π_{RBAC}$) whose authUP is non-empty to the rules in π such that, for each rule ρ, M(ρ) = Ur∈$κ^{-1}$ (ρ)authUP(r), where $κ^{-1}$ is the inverse of κ, i.e., $κ^{-1}$(ρ) is the set of roles that map to rule ρ.

An ABAC policy is structurally consistent with an RBAC policy if there exists a structural correspondence between them.

The ABAC-from-RBAC policy mining problem is: given an RBAC policy $π_{RBAC}$ = ⟨U, Res, Op, Roles, UA, PA, RH⟩, attribute data ⟨A_u, A_r, d_u, d_r ⟩, and a policy quality metric Qpol, find a set Rules of rules such that the ABAC policy π = ⟨U, Res, Op, A_u∪{roles}, Ar, d'$_u$, d_r, Rules⟩

(1) is semantically and structurally consistent with RBAC π , (2) does not use uid,

(3)uses roles and rid only when necessary, and

(4) has the best quality, according to Q_{pol}, among policies that satisfy conditions (1) through (3).

For the policy quality metric, we use weighted structural
complexity as defined in research paper [1]

Policy Mining Algorithm

The algorithm has following steps:

1. Splits the roles in RBAC policy so that each role's assigned permissions are Cartesian product of set of resources and set of operations. This is important so that each role can be translated into a single rule
2. It constructs an ABAC policy rule corresponding to each role
3. Improve the algorithm by merging and simplifying rules

The toplevel Pseudocode is given in Fig 12 and the functions are described as follows:

ComputeUAE(s,U) – computes user attribute expression e_u that characterises the set s of users
elimRedundantSets(e) – Attempts to lower the WSC of e by examining the conjuct for each multi-valued user attribute, and removing each set that is superset of another set in the same conjunct; this leaves the meaning of the rule unchanged, because ⊇ is used in the condition for multivalued attributes in the semantics of user attribute expressions.
ComputeRAE – same at computeUAE except it uses resource attributes
candConst(u,r) – mnemonic for "candidate constraint", returns a set containing all atomic constraints that hold between user u and resource r.
mergeRules(Rules,ƙ) – attempts to reduce the WSC of Rules while preserving semantic and structural consistency, by removing redundant rules and merging pairs of rules. A rule ρ is subsumed by a role ρ' if M(ρ) ⊆ M(ρ'). A rule ρ in Rules is redundant if it is subsumed by another rule in Rules. Informally, rules ρ1 and ρ2 are merged by taking, for each attribute, the union of the conjuncts in ρ1 and ρ2 for that attribute. If adding the resulting rule (including ρ1 and ρ2) preserves structural consistency, then these changes are made to Rules, and the structural correspondence κ is updated accordingly. mergeRules(Rules, κ) updates Rules and κ in place, and it returns a Boolean indicating whether any rules were merged.
simplifyRules(Rules, ƙ) - attempts to simplify the rules in Rules. It updates its arguments Rules and κ in place, replacing rules in Rules with simplified versions when simplification succeeds. It returns a Boolean indicating whether any rules were simplified. It attempts to simplify each rule in several ways, which are embodied in the following
elimConjuncts(ρ, Rules, κ, UP) - attempts to increase the quality of rule ρ by eliminating some conjuncts. Based on our primary goal of minimizing the generated policy's WSC, the quality of rule ρ is |M(ρ)| / WSC(ρ).
elimConstraints(ρ, Rules, κ, UP) - attempts to improve the quality of ρ by removing unnecessary atomic constraints from ρ's constraint ρmerge and An atomic constraint is unnecessary in a rule ρ if removing it from ρ's constraint leaves ρ valid.
elimElements(ρ, Rules, κ, UP) - attempts to decrease the WSC of rule ρ by removing elements from sets in conjuncts for multi-valued user attributes, if removal of those elements produces a rule ρ' that can replace the rules it subsumes; note that, because ⊇ is used in the semantics of user attribute expressions, the set of user-permission pairs that satisfy a rule is unchanged or increased (never decreased) by such removals.
useRoleAttribute(Rules, κ) - replaces uses of "uid" with uses of the user attribute "roles"
removing rules subsumed by ρmerge

```
// Rules is the set of rules
Rules = Ø
// κ is the structural correspondence
κ = Ø
for r in Roles'
    if authUP(r).isEmpty
        continue
    end if
    // create a rule corresponding to r
    e_u = computeUAE(authU(r))
    e_r = computeRAE(asgndRes(r))
    O = asgndOp(r)
    cc = ⋂_{u∈authU(r), s∈asgndRes(r)} candConstr(u, s)
    ρ = ⟨e_u, e_r, O, cc⟩
    Rules.add(ρ)
    κ.add(⟨r, ρ⟩)
end for
// Rules is semantically and structurally consistent with
// π_RBAC. Try to improve its quality, by repeatedly merging
// and simplifying rules, until this has no effect.
mergeRules(Rules, κ)
while simplifyRules(Rules, κ)
  if not mergeRules(Rules, κ)
    break
  end if
end while
useRoleAttribute(Rules, κ)
return ⟨Rules, κ⟩
```

Figure 12 : Pseudocode [3]

Evaluation

Evaluation was performed on manually written case studies used in [1] with slight modification
They manually wrote semantically consistent case study policies in RBAC and ABAC, applied our algorithm to the RBAC policy and accompanying attribute data, and compared the generated ABAC policy with the manually written one.
These experiments with Full Attribute Data demonstrate that, when all relevant attribute data is available, the algorithm successfully produces an

intuitive high-level ABAC policy from an RBAC policy.

Experiments with Incomplete Attribute Data demonstrate that, when some relevant attribute information is unavailable, our algorithm successfully produces an intuitive high-level ABAC policy that uses the available attribute data and uses role membership information as a substitute for missing attribute data

3. Conclusion

We have surveyed the approaches presented by Xu and Stoller for ABAC policy mining given different inputs such as ACLs, RBAC policy and logs using manually written sample policies for evaluation. There is further scope in this study for ABAC policy mining algorithms based on data mining techniques and machine learning. More language features and complex arithmetic constructs can be added to the work. We also need to figure out which attribute to use for constructing the rules from a vast number of available attributes.

4. References

[1] Zhongyuan Xu and Scott D. Stoller. Mining Attribute-Based Access Control Policies. IEEE Transactions on Dependable and Secure Computing 12(5):533-545, September-October 2015.

[2] Zhongyuan Xu and Scott D. Stoller. Mining Attribute-Based Access Control Policies from Logs. In Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec 2014)

[3] Zhongyuan Xu and Scott D. Stoller. Mining Attribute-Based Access Control Policies from Role-Based Policies. In Proceedings of the 10th International Conference & Expo on Emerging Technologies for a Smarter World (CEWIT 2013). © IEEE Press, 2013

[4] Hu et al, A. Guide to Attribute Based Access Control (ABAC) Definition and Consideration (Draft), NIST Special Publication 800-162.

[5] NextLabs, Inc, Managing role explosion with attribute based access control, Jul 2013.

[6] M. Beckerle and L. A. Martucci, “Formal definitions for usableaccess control rule sets—From goals to metrics,” in Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS).ACM, 2013, pp. 2:1–2:11.

[7] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, “Role-based access control models,” IEEE Computer, vol. 29, no. 2, pp.38–47,Feb. 1996.

[8] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. B. Calo, and J. Lobo, “Mining roles with multiple objectives,” ACM Trans.Inf. Syst. Secur., vol. 13, no. 4, 2010.

[9] H. Lu, J. Vaidya, and V. Atluri, “Optimal Boolean matrix decomposition:Application to role engineering,” in Proc. 24th International Conference on Data Engineering (ICDE). IEEE, 2008, pp. 297– 306.