Published in final edited form as:

IEEE Conf Collab Internet Comput. 2017 October ; 2017: 339–348. doi:10.1109/CIC.2017.00051.

Efficient bottom-up Mining of Attribute Based Access Control Policies

Tanay Talukdar*, Gunjan Batra*, Jaideep Vaidya*, Vijayalakshmi Atluri*, and Shamik Sural†

*MSIS Department, Rutgers Business School, USA

†Dept. of Computer Science and Engineering, IIT Kharagpur, India

Abstract

Attribute Based Access Control (ABAC) is fast replacing traditional access control models due to its dynamic nature, flexibility and scalability. ABAC is often used in collaborative environments. However, a major hurdle to deploying ABAC is to precisely configure the ABAC policy. In this paper, we present an *ABAC mining* approach that can automatically discover the appropriate ABAC policy rules. We first show that the ABAC mining problem is equivalent to identifying a set of *functional dependencies* in relational databases that cover all of the records in a table. We also propose a more efficient algorithm, called ABAC-SRM which discovers the most general policy rules from a set of candidate rules. We experimentally show that ABAC-SRM is accurate and significantly more efficient than the existing state of the art.

1. Introduction

Securing organizational resources is one of the fundamental challenges in today's corporate environment, especially given the collaborative nature of business today. Organizations use access control mechanisms to mitigate the risk of unauthorized access to their data, resources and systems, whether such access is local to the organization or external to the organization. Depending on the information required for authorization and the process of making decisions, different access control models have been developed. For traditional information systems, where a system provider needs to deal only with a predictable set of users, access control models such as Discretionary Access Control (DAC) [1], Mandatory Access Control (MAC) [2] and Role based Access Control (RBAC) [3] have been proposed.

Among these, RBAC has emerged as the most adopted by organizations as a means to deploying their access control. However, the primary limitation of RBAC is its significant dependence on user identity for mapping it to a set of roles. As a result, many organizations are now moving to Attribute Based Access Control (ABAC), which is a non-identity based model and is highly dynamic and flexible. Indeed, in collaborative environments, ABAC is significantly more useful and Gartner has predicted that by 2020, 70% of enterprises would use attribute-based access control as the dominant mechanism to protect critical assets, up from less than 5% today. ABAC can comprehensively handle various factors affecting access control decisions like location, time, server load, etc., as well as to facilitate inter-domain accesses. The advantage of not being specific in the relation between a subject and an object helps to provide access to a large number of subject-object combinations. Since ABAC

allows for the creation of access policies based on the existing attributes of the users and objects, rather than the manual assignment of roles, ownership or security labels, it minimizes the need for manual intervention in configuring and deploying access control. Moreover, such use of user and object attributes for defining access control makes it more portable across organizational domains. Indeed the flexibility, portability and *identity-less* access control features make ABAC very attractive to be employed in many application domains, including cloud computing, web services, collaborative and coalition based systems, as it is feasible to make access control decisions without any prior knowledge of the subject.

While an ABAC system is highly flexible in creation and execution of rules to assign access to any user with almost any syntax desired, a significant challenge is, when an organization wants to migrate to ABAC from the traditional DAC based access control, how can an ABAC rule (or policy) be automatically assigned to that user based on the existing rules in the system? Additionally, when a new user with a set of attributes joins the organization, how can one determine the appropriate ABAC rule(s) to that user? This boils down to the problem of *ABAC policy mining*. Such rule discovery is also needed to ensure dynamic sharing of resources with the users who are not defined in the system. It is also useful when resources are shared across organizations in a collaborative manner. ABAC policy mining essentially requires identification of the *significant attributes* acceptable to the organization that a user must possess to gain access to a particular resource.

A solution to the problem of mining ABAC policies was proposed by Xu et al. [4] (we refer to this as Xu-Stoller in the rest of the paper), which presents an ABAC policy mining algorithm discovered from Access Control Lists (ACLs) and attribute data. Although there exist other efforts, this is the only significant solution to date. In this paper, we present a novel ABAC Policy mining approach which is more efficient than Xu-Stoller and also finds better policy rules. We first recognize that discovering ABAC policy rules is equivalent to discovering the combination of attribute conditions that are present in all cases where the permission is present while not resulting in any case where the permission is not present. We show that this is actually equivalent to discovering *functional dependencies* from a database relation (table). Based on this, we propose an ABAC policy mining algorithm, which we refer to as ABAC-FDM. Since this approach exhaustively enumerates all possible user-object pairs, it is not scalable. Therefore, we present a more efficient bottom-up approach, called ABAC-SRM, which first discovers a candidate set of ABAC rules, and then finds the most general rules from this candidate set of rules. We show that this approach is both accurate (generates the correct set of ABAC rules) and efficient (much faster than the state of the art).

2. Preliminaries

In this section, we briefly present the attribute based access control model (ABAC) [5], [6], upon which all of the following work is based. In ABAC the authorization to perform an operation (e.g., read/write/modify) is granted based on the attributes of the requesting user, requested object, and the environment in which a request is made. The basic components of ABAC are as follows:

Users ($\mathcal{U}$): Represents a set of authorized users/subjects. u_i , for $1 \leq i \leq |\mathcal{U}|$ denotes each member of this set.

Objects ($\mathcal{O}$): Represents a set of resources to be protected. Each member of this set is denoted as o_i , for $1 \leq i \leq |\mathcal{O}|$.

Environment ($\mathcal{E}$): Represents a set of environment conditions, independent of users and objects. Each member of this set is denoted as e_i , for $1 \leq i \leq |\mathcal{E}|$.

$\mathcal{U}_{\mathcal{A}}$: Represents a set of user attribute names. Members of these sets are represented as ua_i , for $1 \leq i \leq |\mathcal{U}_{\mathcal{A}}|$. Each ua_i is associated with a set of possible values it can acquire. For instance, if a user attribute *Role* is associated with the values $\{Doctor, Nurse, Patient\}$, then for every $u \in \mathcal{U}$, value of the attribute *Role* can be either *Doctor*, *Nurse* or *Patient*.

$\mathcal{O}_{\mathcal{A}}$: Represents a set of object attribute names. Members of these sets are represented as oa_i , for $1 \leq i \leq |\mathcal{O}_{\mathcal{A}}|$. Each oa_i is associated with a set of possible values it can acquire. For instance, if an object folder with records of medical patients has object attribute *Department* associated with a set of values $\{Cardiology, Dermatology, Gynecology\}$, then for every $o \in \mathcal{O}$, value of the attribute *Department* can be either *Cardiology*, *Dermatology* or *Gynecology*.

$\mathcal{E}_{\mathcal{A}}$: Represents a set of environment attribute names. Members of these sets are represented as ea_i , for $1 \leq i \leq |\mathcal{E}_{\mathcal{A}}|$. Each ea_i is associated with a set of possible values it can acquire. For instance, if an environment attribute *working shift* is associated with a set of values $\{DayShift, EveningShift, NightShift\}$, then for every $e \in \mathcal{E}$, value of the attribute *workingshift* can be either *DayShift*, *EveningShift* or *NightShift*.

$\mathcal{P}$: A set consisting of all possible permissions/operations on objects allowed in a system. For example, if *read* and *write* are the only two possible operations on objects, then $\mathcal{P} = \{read, write\}$. Each member of $\mathcal{P}$ is represented as p_i , for $1 \leq i \leq |\mathcal{P}|$.

$\mathcal{U}_{\mathcal{C}}$: Represents a set of all possible user attribute conditions denoted as uc_j , for $1 \leq j \leq |\mathcal{U}_{\mathcal{C}}|$. Members of this set are represented as equalities of the form $n = c$, where n is a user attribute name and c is either a constant or *any*. For instance if user attribute *Role* has possible values $\{Doctor, Nurse, Patient\}$ and user attribute *Specialty* has possible values as $\{Cardiology, Medicine, Pediatrics\}$, then $\mathcal{U}_{\mathcal{C}}$ will be a set comprising of $\{Role=Doctor, Role=Patient, Role=Nurse, Role=any, Specialty=Cardiology, Specialty=Medicine, Specialty=Pediatrics, Specialty=any\}$. Note here, that the condition $n = any$ does not have to be explicitly chosen. It is set only if at least one other condition for n is present.

$\mathcal{O}_{\mathcal{C}}$: Represents a set of all possible object attribute conditions denoted as oc_k , for $1 \leq k \leq |\mathcal{O}_{\mathcal{C}}|$. Members of this set are represented as equalities of the form $n = c$, where n is an object attribute name and c is either a constant or *any*. For instance if object attribute *Department* has possible values $\{Cardiology, Dermatology, Gynecology\}$ and object attribute *RecordOf* has possible values $\{Doctor, Nurse, Patients, Staff\}$, then $\mathcal{O}_{\mathcal{C}}$ will be a set comprising of $\{Department=Cardiology, Department=Dermatology, Department=Gynecology, Department=any, RecordOf=Doctor, RecordOf=Patient, RecordOf=Nurse, RecordOf=Staff, RecordOf=any\}$. For an attribute name n , if the value of c is *any*, then the attribute n is

not relevant for making the corresponding access decision. Therefore, as above, the condition $n = any$ does not have to be explicitly chosen. It is set only if at least one other condition for n is present. Π : Represents a set of access rules called the rule base of the ABAC system. Each member of this set is denoted as r_i for $1 \leq i \leq |\Pi|$. A rule r in ABAC is a quadruple of the form $\langle uc, oc, ec, p \rangle$. If a user makes a request to access an object, the rule base is searched for any rule through which the user can gain access. If such a rule exists, then the access is granted, otherwise it is denied.

For the sake of simplicity, in this paper, we assume the environment condition to be *any*. In the $\mathcal{U}\mathcal{C}$ and $\mathcal{O}\mathcal{C}$ we have represented the attribute conditions as equalities, however, our approach is flexible to include the complex attribute condition constructs (inequalities, negation, subset, etc.) by converting them to their corresponding list of attributes conditions. In the following, we define the mapping between users and user attribute conditions as well as objects and object attribute conditions.

$\mathcal{U}\mathcal{A}$: User attribute relation $\mathcal{U}\mathcal{A} \subseteq \mathcal{U} \times \mathcal{U}\mathcal{C}$ is a many-to-many mapping of users and user attribute conditions. We use a $m \times n$ binary matrix to represent $\mathcal{U}\mathcal{A}$, where $\mathcal{U}\mathcal{A}[i,j]=1$, if user u_i satisfies an attribute condition uc_j . An example is shown in table 1 where user u_1 is a Nurse whose specialty is *Cardiology*.

$\mathcal{O}\mathcal{A}$: Object attribute relation, $\mathcal{O}\mathcal{A} \subseteq \mathcal{O} \times \mathcal{O}\mathcal{C}$ is a many-to-many mapping of objects and the set of all attributes conditions, where we again use a $m \times n$ binary matrix to represent $\mathcal{O}\mathcal{A}$. $\mathcal{O}\mathcal{A}[i,j]=1$ if an object o_i satisfies an object attribute condition oc_j . Table 2 shows an example where object o_1 is the *record of Patient* in *Cardiology Department*.

3. ABAC Policy Mining Problem

In this section, we formally define the ABAC mining problem. Intuitively, our goal is to discover ABAC policies from traditional authorizations in such a way that the set of ABAC policies is minimum and at the same time the permissions are equivalent to those under the traditional authorization policies. In the following, we develop our definition of the ABAC mining problem, and the formalism needed to precisely state this definition.

$\mathcal{A}$: We assume an authorization a is of the form of $\langle u, o, p \rangle$ represent user $u \in \mathcal{U}$, object $o \in \mathcal{O}$, and permission $p \in \mathcal{P}$, respectively, where a denotes a user u is allowed to perform an operation p on an object o . We use $p.a$ as the permission associated with a . We denote the set of all authorizations as $\mathcal{A}$. For each permissions $p_i \in \mathcal{P}$, we define $\mathcal{A}_{p_i} \subseteq \mathcal{A}$ such that for every $a \in \mathcal{A}_{p_i}$ $p.a = p_i$. For example, if $\mathcal{P} = \{\text{read}, \text{write}\}$, we have $\mathcal{A}_{\text{read}}$ and $\mathcal{A}_{\text{write}}$ such that $\mathcal{A}_{\text{read}} \cup \mathcal{A}_{\text{write}} = \mathcal{A}$.

In ABAC, a policy is comprised of the attribute conditions, which include user attribute conditions and object attribute conditions that determine the grant of a permission to the user $\langle uc, oc, p \rangle$. (Recall from section 2 that we ignore the environment conditions in the ABAC policy specification.) Since our goal is to discover the policies of the nature above, we first construct a *user object permission* ($\mathcal{U}\mathcal{O}\mathcal{P}$) relation as follows. We consider all the user attribute conditions and the object attributes associated with users and objects in $\mathcal{A}$. Given $\mathcal{A}$, we construct a table $\mathcal{U}\mathcal{O}\mathcal{P}_p$ for each permission type p (read, write etc.). The columns of

this matrix are all possible user attribute conditions and object attribute conditions of users and objects in $\mathcal{A}$, respectively, and a column for p . There is a row in $\mathcal{UOP}_p$ for each user object pair. For each row, if the user attribute condition (object attribute condition) is true for a user (object), the corresponding cell is filled with 1, otherwise with 0. If there exists a $a = \langle u, o, p \rangle$, we insert a 1 in the p column of that $u - a$. For the remaining rows, the p column is 0. Given $\mathcal{P} = \{p_1, p_2 \dots p_n\}$, $\mathcal{UOP} = \cup_{i=1}^n \mathcal{UOP}_{p_i}$.

$\mathcal{UOP}_p$: User Object Permission Matrix $\mathcal{UOP}_p$ is a $M \times N$ matrix, where $M = |\mathcal{U}| \times |\mathcal{O}|$ comprising of a row for each user-object pair, and $N = |\mathcal{U}\mathcal{A}| + |\mathcal{O}\mathcal{A}| + 1$, comprising of a column for each object attribute condition, a column for each user attribute condition, and a column for the permission p . For the $\mathcal{U}\mathcal{A}$ in table 1, $\mathcal{O}\mathcal{A}$ in table 2 and $\mathcal{A}$ in table 3a, table 3b shows the $\mathcal{UOP}_{read}$ constructed.

Now, the ABAC Mining Problem is defined as follows:

Definition 1 (ABAC Mining Problem)

Given the set of authorizations $\mathcal{A}$, the set of user attribute conditions ($\mathcal{U}\mathcal{A}$), the set of object attribute conditions ($\mathcal{O}\mathcal{A}$), the ABAC Mining Problem is to discover minimum set of access rules Π such that there exists a rule $r \in \Pi$ where u is allowed to perform p on o iff $a = \langle u, o, p \rangle \in \mathcal{A}$.

Thus the ABAC policy mining problem aims to simply discover the combinations of attribute conditions that are present in at least one row having the permission, while not existing in any row that does not have the permission. We realize that this problem is equivalent to that of discovering *functional dependencies* (FD) from a relation (table). Essentially, if there exists a table with two columns (a_1 and a_2), we say that there exists a FD $a_1 \rightarrow a_2$ if for every pair of rows t_1 and t_2 , if $t_1(a_1) = t_2(a_1)$, then $t_1(a_2) = t_2(a_2)$.

We map the problem of discovering ABAC rules to that of discovering functional dependencies in $\mathcal{UOP}_p$. Essentially, to discover a rule in ABAC, we decompose the attributes of $\mathcal{UOP}_p$ as *determinant columns* (LHS) and *dependent columns* (RHS), where LHS = $\mathcal{U}\mathcal{A} \cup \mathcal{O}\mathcal{A}$ and RHS = $\{p\}$. we need to find the set of attribute conditions $\{ac_1, \dots ac_n\} \subseteq \text{LHS}$ such that if $ac_1 = 1, \dots ac_n = 1$ and $p = 1$, then there does not exist another row in $\mathcal{UOP}_p$ such that $ac_1 = 1, \dots ac_n = 1$ and $p = 0$.

Then we say that LHS $\rightarrow$ RHS, i.e., $\{ac_1, \dots ac_n\} \rightarrow p$. We identify such combinations of attribute conditions as rules in ABAC. And our goal is to find a minimum set of such ABAC rules. For example, we observe in the $\mathcal{UOP}_{read}$ matrix (Table 3) that whenever the value of attribute conditions *Specialty=Medicine* and *Role=Doctor* and *RecordOf=Patient* is 1, the value of read permission is always 1. Thus the ABAC rule is $\{Specialty=Medicine, Role=Doctor, RecordOf=Patient\}$.

Theorem 1—The ABAC mining problem is NP-hard.

Proof 1: We now prove that the ABAC mining problem is NP-hard. Since the ABAC mining Problem is an optimization problem, we first frame the decision version of the problems.

Definition 2. (Decision ABAC)

Given a set of users $\mathcal{U}$, a set of objects $\mathcal{O}$, a set of user-attribute condition $\mathcal{UC}$, a set of object-attributes $\mathcal{OC}$, a user-object-permission assignment $\mathcal{UOP}_p$ which encodes the authorizations A , and $k \geq 0$, are there a set of rules Π such that there exists a rule $r \in \Pi$ where u is allowed to perform p on o iff $a = \langle u, o, p \rangle \in A$ and $|\Pi| \leq k$?

We can now prove that decision ABAC is NP-complete. The Decision ABAC problem is clearly in NP since given a rule set, we can verify in polynomial time if the rule set enables precisely the set of all existing authorizations. We now show how the Vertex Cover Problem[7] can be reduced to Decision-ABAC.

Definition 3. (Vertex Cover)

Given a graph $G = (V, E)$, and a positive integer $k \leq |V|$, is there a vertex cover of size k or less for G (i.e., a subset $V' \subseteq V$ with $|V'| \leq k$, such that for each edge $(u, v) \in E$, at least one of u and v belongs to V')?

Given an instance of the Vertex Cover problem ($G = (V, E)$ and k), we show how it can be reduced to a corresponding instance of the ABAC policy mining problem. First, each vertex v is assigned to either $\mathcal{UC}$ or $\mathcal{OC}$. Thus, $\mathcal{UC} \cup \mathcal{OC} = V$. Next, we construct the matrix $\mathcal{UOP}_p$ as follows: for each edge $e \in E$, we construct a row of $\mathcal{UOP}_p$ which has a 1 for the vertices u and v which comprise it, and 0 otherwise, and the permission 1. Thus, $\mathcal{UOP}_p$ has $|E|$ rows, one for each edge. There are no zero permission rows. k in Decision ABAC is set to k from the vertex cover instance.

Now, we show how a solution to the Decision ABAC problem can be transformed back into a solution for the vertex cover problem. First, a solution to the Decision ABAC problem gives a set of rules Π where, every permission 1 row of $\mathcal{UOP}_p$ is covered by some rule, and the number of rules is no more than k . First, observe that since every permission 1 row in $\mathcal{UOP}_p$ has no more than two attributes (one for each vertex forming the edge), it is impossible for a rule to have three or more attributes. Therefore, a rule must have either 1 or 2 attributes. In the first case, the attribute included corresponds to the vertex in the vertex cover problem that must be included as part of the solution. In the second case, the vertex corresponding to either attribute can be included in the solution. In any case, the final solution to the vertex cover problem contains no more than k vertices (each corresponding to one of the rules forming the solution for the decision ABAC), and covers all of the edges (since each edge corresponds to a row of $\mathcal{UOP}_p$ which is covered by some rule in Π). The transformation is clearly polynomial both ways. This concludes the proof.

4. ABAC Mining Algorithms

Our approach to ABAC mining is to discover the smallest set of access rules (Π) which covers the permission 1 rows and does not cover the permission 0 rows of the $\mathcal{UOP}_p$ matrix. If the total number of user attributes ($\mathcal{UC}$) and object attributes ($\mathcal{OC}$) is $(|\mathcal{UC}| + |\mathcal{OC}|)$ in the $\mathcal{UOP}_p$ matrix, the number of all possible combinations of user-object attributes is $2^{(|\mathcal{UC}| + |\mathcal{OC}|)}$. However, the meaningful and existing user-object combinations are a subset of these. Clearly, it is infeasible to enumerate all user-object combinations. A data-driven technique is

necessary to efficiently discover the user-object combinations. We now look at a guiding principle that is relevant to automated mining of ABAC rules.

Guiding Principle—Identify those user-object attribute combinations which don't have a 0 permission. The basic assumption related to the presence of a user-object combination is as follows: If one of the user-object attributes combination has a 0 permission then we will discard all the combinations where this user-object attribute is present. We will discard them though they might be present for certain user-object combination because they have at least one 0 permission for a user-object combination.

For example, if we have user-object attribute combination $\{uc_1, uc_2, uc_3, oc_1, oc_2, oc_3|p = 1\}$, $\{uc_1, uc_3, oc_1, oc_2|p = 1\}$ and $\{uc_1, oc_1|p = 0\}$ we cannot use $\{uc_1, uc_3, oc_1, oc_2|p = 1\}$ as a rule because the permission for the user-object combination $\{uc_1, oc_1|p = 0\}$ is 0.

Ideally, to generate the rules we must identify potential combinations of user-object attribute which covers all the permissions with 1 and doesn't cover any permissions with 0. We provide two approaches ABAC-FDM and ABAC-SRM which take into consideration the above guiding principle. The first algorithm, called ABAC-FDM, is able to identify all the potential rules. However, the complexity of this algorithm becomes exponential when the count of User Attribute Conditions and Object Attribute Conditions increases. The second algorithm ABAC-SRM is faster (the worst case complexity is $|\mathcal{U}\mathcal{O}\mathcal{P}_p|^3$ but in practice it is much faster). We will discuss further in section 4.2.1). ABAC-SRM also discovers the potential specific rules and gives the guarantee that no other user-object attribute combination is valid except those specific rules.

4.1. ABAC-FDM Algorithm

The ABAC-FDM algorithm (Algorithm 1) takes as input the $\mathcal{U}\mathcal{O}\mathcal{P}_p$ matrix, user Attribute conditions ($\mathcal{U}\mathcal{C}$) and object attribute conditions ($\mathcal{O}\mathcal{C}$). It calculates the *PermCount*, which is the number of rows in $\mathcal{U}\mathcal{O}\mathcal{P}_p$, which have not been covered by any FD yet and have value as 1.

Algorithm 1

ABAC-FDM Algorithm

Require: Dataset $D = \mathcal{U}\mathcal{O}\mathcal{P}_p, \mathcal{U}\mathcal{C}, \mathcal{O}\mathcal{C}$

- 1: $i \leftarrow 1$
- 2: $FDs \leftarrow \{\}$
- 3: $PermCount \leftarrow \# \text{ Rows in } \mathcal{U}\mathcal{O}\mathcal{P}_p \text{ where (Permission= 1)}$
- 4: **while** ($PermCount > 0$) and ($i < |\mathcal{U}\mathcal{C}| + |\mathcal{O}\mathcal{C}| + 1$) **do**
- 5: $CFDs \leftarrow FDGen(i)$ {FDGen generates Candidate set of FDs at level i }
- 6: **if** $\exists fd \in CFD$, such that fd covers at least one uncovered row in $\mathcal{U}\mathcal{O}\mathcal{P}_p$ **then**
- 7: $bfd \leftarrow$ pick fd from CFD with the highest uncovered count
- 8: $CoveredCount \leftarrow \# \text{ rows in } \mathcal{U}\mathcal{O}\mathcal{P}_p \text{ that are newly covered by } bfd$
- 9: $PermCount \leftarrow PermCount - CoveredCount$

```

10:   For each row  $r$  in  $\mathcal{U}\mathcal{O}\mathcal{P}_p$  such that  $r$  is covered by  $bfd$ , mark it as covered
11:    $FDs \leftarrow FDs \cup bfd$ 
12:   end if
13:    $i \leftarrow i + 1$ 
14: end while
15: return  $FDs$ 

```

Algorithm 2

FDGen(i)

```

function FDGen( $i$ )
1:  for (Each  $i$  – combination of  $\mathcal{U}\mathcal{C} \cup \mathcal{O}\mathcal{C}$ ) do
2:    if  $\exists$  Row where  $val(\mathcal{U}\mathcal{C}, \mathcal{O}\mathcal{C}) = 1$  and  $(Permission) = 0$  then
3:       $CFDs(i) \leftarrow i$  – combination
4:    end if
5:  end for
6:  return  $CFDs(i)$ 

```

The level of FD is denoted by i , which denotes the i -combinations of $\mathcal{U}\mathcal{C}$ and $\mathcal{O}\mathcal{C}$ that will be checked for having a Functional Dependency with permission. For example, if $i=2$, all $\binom{N}{2}$ ($(uc_1, uc_2), (uc_1, uc_3), (uc_1, oc_1), (uc_1, oc_2)$ and so on) combinations of attribute conditions will be checked for the presence of an FD (where N = total number of attribute conditions). Similarly, for $i=3$, all $\binom{N}{3}$ combinations will be generated and tested for FD. i is initialized as 1 and is increased by 1 after each iteration.

The algorithm does not terminate until all the rows where $p = 1$ in $\mathcal{U}\mathcal{O}\mathcal{P}_p$ are covered, or all possible combinations of $\mathcal{U}\mathcal{C}$ and $\mathcal{O}\mathcal{C}$ get exhausted.

FD Generation—The FDs are checked using the FDGen(i) function in Algorithm 2. The function generates all possible i -combinations of functional dependencies and evaluates each combination by checking $\forall(\mathcal{U}\mathcal{C} \wedge \mathcal{O}\mathcal{C}) \rightarrow P == 1$. It essentially determines that combination as a Candidate FD (CFD) for which there is no Permission=0 in the entire $\mathcal{U}\mathcal{O}\mathcal{P}_p$. FDGen returns all i -level CFDs.

FD Selection—This phase runs from lines 6–12 in Algorithm 1. From the CFDs generated at level i , the algorithm picks bfd , the Candidate FD with the highest Uncovered count. Uncovered count is the number of rows in $\mathcal{U}\mathcal{O}\mathcal{P}_p$ where the FD exists, but have not been covered by an FD before. *CoveredCount* provides precisely this count – the number of rows where the candidate functional dependency exists in the $\mathcal{U}\mathcal{O}\mathcal{P}_p$ and those that have not been covered before. The *PermCount* is updated by reducing it by *CoveredCount*. We also mark the rows in the $\mathcal{U}\mathcal{O}\mathcal{P}_p$ which are covered by the bfd . Now the bfd is added to the FDs.

Incrementing i —Once the CFD list is exhausted, i is incremented by 1. As mentioned before, the entire process is iteratively performed by increasing i every time until all permissions in $\mathcal{UOP}_p$ are covered or all possible FD combinations have been checked.

Example—The entire process is explained using the illustrative example shown in Table 4. Table 4a shows the initial $\mathcal{UOP}_p$. Since there are no candidate FDs generated in the iteration with $i = 1$ (Table 4b), there are no FDs picked (Table 4c), and therefore the covered $\mathcal{UOP}_p$ stays the same. In the second iteration, 2 CFDs are generated by FDGen (table 4e). The FD with higher coverage, $\{uc_3, oc_2\}$, is selected and moved to Final FD table (table 4f). Table 4d shows the covered $\mathcal{UOP}_p$ at this point. Since there are no more CFDs which can cover any more rows of $\mathcal{UOP}_p$ this iteration finishes and i is incremented to 3. In the third iteration again 2 CFDs are generated by FDGen (table 4h) which are processed in the similar way (table 4i). Since $\mathcal{UOP}_p$ is now completely covered (table 4g), the algorithm terminates. Thus, the FDs finally generated are $\{uc_3, oc_3\}$ and $\{uc_1, uc_2, oc_1\}$.

4.2. ABAC-SRM Algorithm

The key problem with the ABAC-FDM algorithm is its computational complexity. Since we compute all possible combinations of user-object attributes, the running time is exponential. This is quite infeasible, except for very small data sets. This brings us to the efficient algorithm for rule mining: ABAC-SRM. This algorithm only generates combinations of user-object attributes that satisfies the guiding principle. Below we discuss the ABAC-SRM algorithm in detail.

The goal of ABAC-SRM is to generate a minimal set of rules where each rule has the minimum weight. To do this, first we create the lists *permission1 Rules* (the rows of $\mathcal{UOP}_p$ where the permission is 1) and *permission0 Rules* (the rows of $\mathcal{UOP}_p$ where the permission is 0) and sort *permission1 Rules* in ascending order based on the size of each row (number of attributes possessed). Since we want to cover all of the cases where the permission is granted, we only need to look at the combinations of attributes present in the *permission1 Rules*. The key idea is to go over each such rule and to see if it can be covered by a pre-existing rule in some way. If not, it is then added to the list of specific rules. Thus, for each rule r in *permission1 Rules*, we see if r can be covered by any rule sr currently existing in *SpecificRules* (line 5). If so, r can be eliminated since it is already covered. If not, we check if the intersection of r and sr (the common attribute set) is present in *permission0 Rules*. If not (i.e., it does not give spurious permissions), we go ahead and include the (smaller) intersection set as a rule in *SpecificRules* while eliminating the current specific rule sr (line 9).

Once we have gone through all of the rules r in *permission1 Rules*, *SpecificRules* contains the final list of ABAC policy rules that generate $\mathcal{UOP}_p$. However, these rules may contain attributes that are unnecessary. Therefore, in the final step, for each rule in the list we remove all attributes from each rule that do not create spurious permissions (lines 16–21). Thus, the *finalSpecificRules* set generated by the ABAC-SRM algorithm not only minimizes the number of rules, but also tries to ensure that each rule has the minimum weight as per

WSC (we have referred to the definition of WSC given in [4]). Algorithm 3 gives the detailed steps.

Example—We again use the illustrative example shown in Table 4a to demonstrate the working of ABAC-SRM. For this $\mathcal{UOP}_p$ table 5a gives the sorted permission 1 rows while table 5b gives the permission 0 rows. As noted earlier, ABAC-SRM starts by going over each of the rules in the sorted permissions 1 list. Initially since there is nothing in the specific rules list, the smallest set $\{uc_1, uc_2, oc_1\}$ in the permission1 list will be inserted into specific list. Next, the algorithm looks at $\{uc_1, uc_3, oc_1, oc_2\}$. Since $\{uc_1, uc_2, oc_1\}$ does not cover it, we take the intersection between $\{uc_1, uc_2, oc_1\}$ and $\{uc_1, uc_3, oc_1, oc_2\} \equiv \{uc_1, oc_1\}$ and check if $\{uc_1, oc_1\}$ is a subset of permission 0 list. Here $\{uc_1, oc_1\}$ is a subset of permission 0 list so we do not remove $\{uc_1, uc_2, oc_1\}$ from the specific list and keep iterating the specific list until it gets exhausted. If no rules are added till the specific list terminates (as in this case) we add the rule $\{uc_1, uc_3, oc_1, oc_2\}$, in this case) to the specific list. Now, the specific list has $\{uc_1, uc_2, oc_1\}$ and $\{uc_1, uc_3, oc_1, oc_2\}$ as the intermediate rules. We then iterate over the rest of the list as described above and see that $\{uc_1, uc_2, oc_1\}$ covers $\{uc_1, uc_2, oc_1, oc_2\}$, so we can skip $\{uc_1, uc_2, oc_1, oc_2\}$. In the final iteration we see that $\{uc_1, uc_3, oc_2, oc_3\}$ is not covered by $\{uc_1, uc_2, oc_1\}$. Taking the intersection between them we get $\{uc_1\}$ which is a subset of permission 0 list, so we discard $\{uc_1\}$ and keep going over the remaining rules in the specific list. $\{uc_1, uc_3, oc_2, oc_3\}$ is also not covered by $\{uc_1, uc_3, oc_1, oc_2\}$. However, their intersection, $(\{uc_1, uc_3, oc_2\})$ is not present in the permission 0 list, therefore we insert $\{uc_1, uc_3, oc_2\}$ into the specific list and remove $\{uc_1, uc_3, oc_1, oc_2\}$ from the specific rules. Finally, *Specificrules* consists of $\{uc_1, uc_2, oc_1\}$ and $\{uc_1, uc_3, oc_2\}$. In the final stage, we check if the rules in the *Specificrules* can be further shortened. First we get the rule $\{uc_1, uc_2, oc_1\}$. On removing each attribute uc_1, uc_2, oc_1 iteratively we see $\{uc_2, oc_1\}$, $\{uc_1, oc_1\}$ and $\{uc_1, uc_2\}$ are subset of permission 0 list, so we cannot reduce the size of $\{uc_1, uc_2, oc_1\}$. Next, we examine the rule $\{uc_1, uc_3, oc_2\}$. On removing attribute uc_1 we see $\{uc_3, oc_2\}$ is not a subset of permission 0 list, thus we replace rule $\{uc_1, uc_3, oc_2\}$ with rule $\{uc_3, oc_2\}$, which cannot be further shortened. Finally we have $\{uc_1, uc_2, oc_1\}$ and $\{uc_3, oc_2\}$ as our final rules. The final output can be seen in Table 5c.

Algorithm 3

ABAC-SRM Algorithm

Require: Dataset $D = \mathcal{UOP}_p$

Require: *permission1 Rules*, the rows of D where the permission is 1

Require: *permission0 Rules*, the rows of D where the permission is 0

- 1: $specificRules, finalSpecificRules \leftarrow \phi$
- 2: Sort *permission1 Rules* in ascending order based on #attributes in each rule
- 3: **for** each rule r in *permission1 Rules* **do**
- 4: **for** each rule sr in *specificRules* **do**
- 5: **if** $sr \subseteq r$ **then**
- 6: continue to next rule in *permission1 Rules* since rule r is already covered
- 7: **else**

```

8:      if  $(sr \cap r) \notin \text{permission0Rules}$  then
9:           $\text{specificRules} \leftarrow (\text{specificRules} \cup sr) \cup (sr \cap r)$ 
10:         continue to next rule in  $\text{permission1Rules}$  since  $r$  is now covered
11:     end if
12: end if
13: end for
14:  $\text{specificRules} \leftarrow \text{specificRules} \cup r$ 
15: end for
16: for each rule  $sr$  in  $\text{specificRules}$  do
17:     for each attribute  $a$  in  $sr$  do
18:         if  $(sr \mid a) \notin \text{permission0Rules}$  then
19:              $sr \leftarrow sr \mid a$ 
20:         end if
21:     end for
22:  $\text{finalSpecificRules} \leftarrow \text{finalSpecificRules} \cup sr$ 
23: end for
24: return  $\text{finalSpecificRules}$ 

```

4.2.1. Asymptotic Running Time—We now analyze the asymptotic running time of the ABAC-SRM algorithm. Let m denote the number of users and n denote the number of objects. Then the size of the dataset $D = mn$. Assume that the number of rows with permission 1 is x and the number of permission 0 rows is y . Clearly, $x + y = mn$. Now let us analyze the main nested loop of algorithm 3. Essentially, we go through all of the permission1Rules . For each permission1Rule we look through the specific rules existing at that point, and when a new candidate rule is generated, the permission0Rules are checked to see if the rule is valid. Initially permission1Rules at line 12 will be of size x and specificRule at line 14 will have a size 0. In the worst case, every time we go through one of the permission1Rules , one rule is added to specificRule . Thus the running time of this part is $\frac{(x(x-1))}{2}$. Now for each iteration in the nested loop we go over all of permission0Rules at line 19 in the worst case. Therefore the total running time is $\frac{(x(x-1)) * y}{2} \equiv \frac{(x(x-1)) * (D-x)}{2}$. Since x and y are both dependent on $|D|$, in the absolute worst case running time of algorithm 3 is cubic in $|D|$, which is similar to the worst case time of Xu-Stoller. However, in practice, the algorithm takes significantly less time, and as can be seen in the experiments significantly outperforms Xu-Stoller.

5. Experimental Evaluation

The purpose of the experimental evaluation is to validate the effectiveness of the proposed approach in a wide variety of settings, and to understand the effect of different factors. Since very difficult to find suitable real data sets, we create synthetic data sets with specific parameter values. We also appropriately convert our data into the format accepted by [4] (which we refer to as Xu-Stoller algorithm) and compare the timing, number of rules generated and total weight of the rules generated (WSC) with ABAC-SRM. Since Xu-Stoller

algorithm also takes the id's into consideration, we made sure to insert id's while creating the dataset to ensure fair comparison. In the following experiments we only create data with a single permission and without putting constraints though our ABAC-SRM can handle multiple permissions and constraints (which we have tested when we run our algorithm on Xu-Stoller dataset).

For each set of parameter values, synthetic data was created with three different random seeds, and Xu-Stoller and ABAC-SRM run 5 different times. The overall results are averaged over all of these runs. For each set of experiments, we compared both the algorithms on three factors. First, size of rules generated. Second, weight of rules generated. Third, time taken to generate the rules. We also report the actual number of rules used to create the dataset so that we can compare the efficiency of both the algorithms with the actual one.

The key parameters are the number of users ($\mathcal{U}$), objects ($\mathcal{O}$), user attributes ($\mathcal{U}_\mathcal{C}$), object attributes ($\mathcal{O}_\mathcal{C}$), number of rules given (R_G) (maximum number of rules that can be used to generate the dataset), and the maximum size of each rule (set by the DivFactor, D_F). In the experiments below, one parameter is varied while keeping the others constant to evaluate the performance with respect to that parameter.

We first look at the effect of varying the number of users, while everything else is kept constant. Table 6 shows the results obtained by both ABAC-SRM and Xu-Stoller in each case. The first six columns give the dataset parameters, while the remaining five columns give the average number of rules (Π) used to generate the data, average number of rules found by ABAC-SRM and Xu-Stoller, and average WSC for ABAC-SRM and Xu-Stoller. We see that both the algorithms discover almost the same number of rules and they are within the limit of the number of input rules. Furthermore, the average WSC of rules for ABAC-SRM is better than that achieved by Xu-Stoller. Figure 1a shows the running time (log scale) of both the algorithms. We also plot the 95% confidence interval for the running time obtained (across all runs). It is clear that Xu-Stoller takes several orders of magnitude more time than ABAC-SRM. We next see the effect of increasing the number of user/object attributes while keeping everything else constant. Table 7 gives the results. Again, we see both the algorithms discover almost the same number of rules and they are within the limit of the number of input rules. Once again, on average, the WSC of rules for ABAC-SRM is better than Xu-Stoller. Figure 1b shows the running time(log scale) of both the algorithms. We can see that ABAC-SRM has a constant running time across all the datasets and that increasing user-object attributes does not have any effect to the run time of ABAC-SRM but Xu-Stoller takes significantly longer time to run than ABAC-SRM and also for Xu-Stoller the time increases exponentially as the user-object attribute count increases. Indeed, for the larger datasets we were not able to complete 5 iterations as Xu-Stoller algorithm was taking very long time (more than a day) to complete each iteration and we had to manually terminate those iterations.

Next we look at the effect of varying the number of rules (increasing the rule count). Table 8 gives the results. Again, both ABAC-SRM and Xu-Stoller discover almost same number of rules and they are less than the number of input rules that are fed to the data generator. In

this setup also we can see average WSC of the rules found out by ABAC-SRM is better than that of Xu-Stoller. Figure 1c shows the running time (log scale) of both the algorithms. We can see that ABAC-SRM has a constant running time across all the datasets and also increasing rule count does not have any effect to the run time of ABAC-SRM but Xu-Stoller takes significantly longer time to run than ABAC-SRM and also the running time fluctuates a lot which can be seen in the graph shown in Figure 1c.

Finally, we see the effect of varying the DivFactor (D_F), which controls the maximum number of user-attributes and object-attributes that can be in each rule. Table 9 gives the results. We see that both the algorithms discover almost the same number of rules which are less than the number of input rules that are fed to the data generator. Also the average WSC of ABAC-SRM is better than Xu-Stoller. Figure 1d shows the running time(log scale) of both the algorithms. We can see that ABAC-SRM has a slightly decreasing running time across all the datasets but Xu-Stoller fluctuates a lot. Overall, we can conclude that increasing rule size does not have any effect in the run time of ABAC-SRM.

Apart from the synthetic data, we also examined the performance of our algorithm on the university.abac dataset described in [4]. We converted the data to match ABAC-SRM input (we also incorporated the user-id and resource-id present in the university data when we converted it to our format). We got the same number of rules (10 rule) as output by Xu-Stoller algorithm. The average WSC of rules generated by ABAC-SRM for this is 3.9 where as that of Xu-Stoller algorithm is 3.8 but from the perspective of time, ABAC-SRM took on average .062 seconds whereas Xu-Stoller took on average 1.05 seconds to run. For sake of simplicity, we have performed this for the manual data set, however in future we will perform the same for synthetic dataset.

The key benefit of ABAC-SRM is that the run time is not significantly affected by different attributes. The generalization step is fast for ABAC SRM because it eliminates the unnecessary attributes in each rule by checking one by one if an attribute is subset of permission0 rule. In comparison, Xu-Stoller takes a longer time in the generalization step while eliminating conjuncts in attribute expression with constraints. In all of the four graphs we can see the average run time of ABAC-SRM is almost constant with increasing rule number, rule complexity and user-object attribute size which is supported by 95% confidence interval. The only time when running time increases (linearly) is when the user count is increased, which makes sense.

6. Related Work

There is a significant body of work on security policy mining in the context of RBAC, called role mining. Specifically, several top-down, bottom-up and hybrid role mining approaches have been proposed in the recent past. The bottom-up approaches [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21] follow a similar principle of deriving the policy from the existing user-permission assignment. [22] provides a comprehensive literature survey on this topic. There has also been work on mining ABAC policies [4], [23], [24], [25]. Specifically, Xu et al. [4] proposed an approach (Xu-Stoller) for ABAC policy

mining, that we have directly compared to. Effectively, our algorithm finds a simpler set of ABAC rules in a far more efficient way.

The work by Medvet et al. [23] uses the same ABAC language and case studies as in Xu-Stoller and employs an evolutionary and separate and conquer approach, where at every iteration, a new rule is generated and the set of access requests is reduced to a smaller size. This has the same efficiency as that of Xu-Stoller. The work by Mocanu et al. [24], which uses deep learning techniques (Restricted Boltzmann Machines) to generate a proper dataset from logs that account for not only the positive authorization rules, but also the under-assigned, over-assigned and negative authorization rules as well, and then feed this dataset to Xu-Stoller algorithm, thus requiring more time. The work of Benkaouz et al. [25] attempts to reduce the number of ABAC policy rules and chooses the granularity of ABAC policies using K-Nearest Neighbor technique. This work is complementary to our work and can be used as a post processing step.

7. Conclusions and Future Work

In this paper, we have examined the problem of mining ABAC policies in a bottom up fashion. We formalize this problem and propose an efficient algorithm (ABAC-SRM). We experimentally evaluate the performance of the proposed algorithm and show that it takes an order of magnitude less time than the existing state of the art in most cases. In the future, we plan to examine constraints such as separation of duty, and the notion of negative permissions.

Acknowledgments

Research reported in this publication was supported by the National Institutes of Health under award R01GM118574 and by the National Science Foundation under awards CNS-1422501 and CNS-1624503. The content is solely the responsibility of the authors and does not necessarily represent the official views of the agencies funding the research.

References

1. Graham G, Denning P. Protection principles and practice. AFIPS Spring Joint Computer Conference; 1972. 417–429.
2. Bell D, LaPadula L. Secure computer systems: Unified exposition and multics interpretation. The Mitre Corporation; Bedford, MA: Mar, 1976 Technical Report MTR-2997
3. Ferraiolo DF, Sandhu R, Gavrila S, Kuhn DR, Chandramouli R. Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security (TISSEC). 2001; 4(3): 224–274.
4. Xu Z, Stoller SD. Mining attribute-based access control policies. IEEE Transactions on Dependable and Secure Computing. 2015; 12(5):533–545.
5. Hu VC, Ferraiolo D, Kuhn R, Friedman AR, Lang AJ, Cogdell MM, Schnitzer A, Sandlin K, Miller R, Scarfone K, et al. Guide to attribute based access control (abac) definition and considerations (draft). NIST Special Publication. 2013; 800(162)
6. Jha S, Sural S, Atluri V, Vaidya J. Enforcing separation of duty in attribute based access control systems. International Conference on Information Systems Security; Springer; 2015. 61–78.
7. Gary MR, Johnson DS. Computers and intractability: A guide to the theory of np-completeness. 1979

8. Vaidya J, Atluri V, Guo Q. The role mining problem: finding a minimal descriptive set of roles. *Proceedings of the 12th ACM symposium on Access control models and technologies*; ACM; 2007. 175–184.
9. Vaidya J, Atluri V, Warner J, Guo Q. Role engineering via prioritized subset enumeration. *IEEE Transactions on Dependable and Secure Computing*. 2010; 7:300–314.
10. Vaidya J, Atluri V, Guo Q. The role mining problem: A formal perspective. *ACM Trans Inf Syst Secur*. 2010; 13(3):1–31.
11. Vaidya J, Atluri V, Guo Q, Lu H. Edge-rmp: Minimizing administrative assignments for role-based access control. *Journal of Computer Security*. 2009; 17(2):211–235.
12. Xu Z, Stoller SD. Mining attribute-based access control policies from rbac policies. *Emerging Technologies for a Smarter World (CEWIT), 2013 10th International Conference and Expo on*; IEEE; 2013. 1–6.
13. Frank M, Streich AP, Basin D, Buhmann JM. Multi-assignment clustering for boolean data. *Journal of Machine Learning Research*. Feb.2012 13:459–489.
14. Huang H, Shang F, Zhang J. Approximation algorithms for minimizing the number of roles and administrative assignments in RBAC. *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*; IEEE; 2012. 427–432.
15. Lu H, Hong Y, Yang Y, Duan L, Badar N. Towards user-oriented rbac model. *Journal of Computer Security*. 2015; 23(1):107–129.
16. Frank M, Streich AP, Basin D, Buhmann JM. A probabilistic approach to hybrid role mining. *Proceedings of the 16th ACM conference on Computer and communications security*; ACM; 2009. 101–111.
17. Colantonio A, Di Pietro R, Ocello A, Verde NV. A probabilistic bound on the basic role mining problem and its applications. *IFIP International Information Security Conference*; Springer; 2009. 376–386.
18. Molloy I, Chen H, Li T, Wang Q, Li N, Bertino E, Calo S, Lobo J. Mining roles with semantic meanings. *Proceedings of the 13th ACM symposium on Access control models and technologies*; ACM; 2008. 21–30.
19. Blundo C, Cimato S. A simple role mining algorithm. *Proceedings of the 2010 ACM Symposium on Applied Computing*; ACM; 2010. 1958–1962.
20. Colantonio A, Di Pietro R, Ocello A, Verde NV. Visual role mining: A picture is worth a thousand roles. *IEEE Transactions on Knowledge and Data Engineering*. 2012; 24(6):1120–1133.
21. Colantonio A, Di Pietro R, Verde NV. A business-driven decomposition methodology for role mining. *Computers & Security*. 2012; 31(7):844–855.
22. Mitra B, Sural S, Vaidya J, Atluri V. A survey of role mining. *ACM Computing Surveys*. 2016; 48(4):50:1–50:37.
23. Medvet E, Bartoli A, Carminati B, Ferrari E. Evolutionary inference of attribute-based access control policies. *International Conference on Evolutionary Multi-Criterion Optimization*; Springer; 2015. 351–365.
24. Mocanu D, Turkmen F, Liotta A. Towards ABAC policy mining from logs with deep learning. In *proc. of the 18th International Multiconference, IS2015*; 2015. 124–128.
25. Benkaouz Y, Erradi M, Freisleben B. Work in progress: K-nearest neighbors techniques for abac policies clustering. *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*; ACM; 2016. 72–75.

Figure 1.
Running Time for ABAC-SRM and Xu-Stoller

TABLE 1 $\mathcal{U}, \mathcal{A}$

User (u)	Speciality =Medicine (uc_1)	Role=Doctor (uc_2)	Speciality=Cardiology (uc_3)	Role=Nurse (uc_4)
u_1	0	0	1	1
u_2	0	1	1	0
u_3	1	1	0	0
u_4	1	0	0	1

TABLE 2

 $\mathcal{O}, \mathcal{A}$

Object (o)	Department=Cardiology (oc_1)	Department=Medicine (oc_2)	Recordof=Patient (oc_3)
o_1	1	0	1
o_2	0	1	1

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

TABLE 3

The set of Authorizations and the corresponding $\mathcal{U}\mathcal{O}\mathcal{P}_{Read}$

(a) Authorizations ($\mathcal{A}$)		
User u	Object o	Permission p_i
u_1	o_1	p_1
u_2	o_1	p_1
u_2	o_1	p_2
u_3	o_2	p_1
u_3	o_2	p_2
u_n	o_2	p_1

(b) $\mathcal{U}\mathcal{O}\mathcal{P}_{Read}$								
User-Object $u-o$	Speciality=Medicine (uc_1)	Role=Doctor (uc_2)	Speciality=Cardiology (uc_3)	Role=Nurse (uc_4)	Department=Cardiology (oc_1)	Department=Medicine (oc_2)	RecordOf=Patient (oc_3)	$p_1=Read$
$u_1 o_1$	0	0	1	1	1	0	1	1
$u_2 o_1$	0	1	1	0	1	0	1	1
$u_3 o_1$	1	1	0	0	1	0	1	0
$u_4 o_1$	1	0	0	1	1	0	1	0
$u_1 o_2$	0	0	1	1	0	1	1	0
$u_2 o_2$	0	1	1	0	0	1	1	0
$u_3 o_2$	1	1	0	0	0	1	1	1
$u_4 o_2$	1	0	0	1	0	1	1	1

TABLE 4

Illustrative example

(a) Initial $\mathcal{UOP}$							
$u - o$	uc_1	uc_2	uc_3	oc_1	oc_2	oc_3	p
$u_1 o_1$	0	1	0	1	0	0	0
$u_2 o_1$	1	0	1	1	0	0	0
$u_3 o_1$	1	1	0	1	0	0	1
$u_1 o_2$	0	1	0	1	1	0	0
$u_2 o_2$	1	0	1	1	1	0	1
$u_3 o_2$	1	1	0	1	1	0	1
$u_1 o_3$	0	1	0	0	1	1	0
$u_2 o_3$	1	0	1	0	1	1	1
$u_3 o_3$	1	1	0	0	1	1	0

(b) Candidate FDs at iteration $i=1$							
Candidate FDs	Coverage						
None	None						

(c) FDs at iteration $i=1$							
FDs							
None							

(d) Covered $\mathcal{UOP}$ at iteration $i=2$							
$u - o$	uc_1	uc_2	uc_3	oc_1	oc_2	oc_3	p
$u_1 o_1$	0	1	0	1	0	0	0
$u_2 o_1$	1	0	1	1	0	0	0
$u_3 o_1$	1	1	0	1	0	0	1
$u_1 o_2$	0	1	0	1	1	0	0
$u_2 o_2$	1	0	1	1	1	0	1
$u_3 o_2$	1	1	0	1	1	0	1
$u_1 o_3$	0	1	0	0	1	1	0
$u_2 o_3$	1	0	1	0	1	1	1
$u_3 o_3$	1	1	0	0	1	1	0

(e) Candidate FDs at iteration $i=2$							
Candidate FDs	Coverage						
uc_3, oc_2 ✓	2						
uc_3, oc_3	1						

(f) FDs at iteration $i=2$							
FDs							
uc_3, oc_2							

(g) Covered $\mathcal{UOP}$ at iteration $i=3$							
$u - o$	uc_1	uc_2	uc_3	oc_1	oc_2	oc_3	p
$u_1 o_1$	0	1	0	1	0	0	0
$u_2 o_1$	1	0	1	1	0	0	0
$u_3 o_1$	1	1	0	1	0	0	1
$u_1 o_2$	0	1	0	1	1	0	0
$u_2 o_2$	1	0	1	1	1	0	1
$u_3 o_2$	1	1	0	1	1	0	1
$u_1 o_3$	0	1	0	0	1	1	0
$u_2 o_3$	1	0	1	0	1	1	1
$u_3 o_3$	1	1	0	0	1	1	0

(h) Candidate FDs at iteration $i=3$							
Candidate FDs	Coverage						
uc_1, uc_2, oc_1 ✓	2						
uc_1, oc_1, oc_2	1						

(i) FDs at iteration $i=3$							
FDs							
uc_3, oc_2							
uc_1, uc_2, oc_1							

TABLE 5

Illustrative example for ABAC-SRM

(a) Permission1 Table									
$u - o$	uc_1	uc_2	uc_3	oc_1	oc_2	oc_3	p		
$u_3 o_1$	1	1	0	1	0	0	1		
$u_2 o_2$	1	0	1	1	1	0	1		
$u_3 o_2$	1	1	0	1	1	0	1		
$u_2 o_3$	1	0	1	0	1	1	1		

(b) Permission0 Table									
$u - o$	uc_1	uc_2	uc_3	oc_1	oc_2	oc_3	p		
$u_1 o_1$	0	1	0	1	0	0	0		
$u_2 o_1$	1	0	1	1	0	0	0		
$u_1 o_2$	0	1	0	1	1	0	0		
$u_1 o_3$	0	1	0	0	1	1	1		
$u_3 o_3$	1	1	0	0	1	1	1		

(c) Result Table	
Final Rules	
$\{uc_1, uc_2, oc_1\}$	
$\{uc_3, oc_2\}$	

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

TABLE 6

Increasing User Count

$ U $	$ O $	$ U_G $	$ O_G $	$ R_G $	D_F	$Avg\ III$	$AvgRF_{ABAC-SRM}$	$AvgRF_{Xu-Stoller}$	$AvgWSC_{ABAC-SRM}$	$AvgWSC_{Xu-Stoller}$
100	100	25	25	30	5	25	25	24.67	112	128
1000	100	25	25	30	5	26	25.67	25.67	127.67	160.67
10000	100	25	25	30	5	26	26	27	129.33	793.67

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

TABLE 7

Increasing User-Object Attributes

$ U $	$ O $	$ UO $	$ R_G $	D_F	$Avg\ [III]$	$AvgRF_{ABAC-SRM}$	$AvgRF_{Xu-Stoller}$	$AvgWSC_{ABAC-SRM}$	$AvgWSC_{Xu-Stoller}$
1000	100	20	30	5	25.33	25.33	27	115.67	145.67
1000	100	25	30	5	26	25.67	25.67	127.67	160.67
1000	100	30	30	5	26.67	27	26.67	139.33	202
1000	100	35	30	5	27.33	26.67	26.67	135.67	152.67

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

TABLE 8

Increasing Rule Count

$ U $	$ C $	$ M $	$ R_G $	D_F	Avg $ I $	Avg $RF_{ABAC-SRM}$	Avg $RF_{Xu-Stoller}$	Avg $WSC_{ABAC-SRM}$	Avg $WSC_{Xu-Stoller}$
1000	100	25	25	20	5	17.67	17.67	86.67	101
1000	100	25	25	30	5	26	25.67	127.67	160.67
1000	100	25	25	40	5	36.33	36	178	295
1000	100	25	25	50	5	43.33	43.67	215.33	321

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

TABLE 9

Increasing Rule Size

$ U $	$ C $	$ U_C $	$ R_C $	D_F	Avg $ I $	Avg $RF_{ABAC-SRM}$	Avg $RF_{Xu-Stoller}$	Avg $WSC_{ABAC-SRM}$	Avg $WSC_{Xu-Stoller}$
1000	100	25	30	5	26	26.67	26.67	127.67	160.67
1000	100	25	30	4	26	25.67	25.67	132.33	161
1000	100	25	30	3	25	25	24.67	140	148
1000	100	25	30	2	24.33	24.67	24	139.67	146.67