Incremental Maintenance of ABAC Policies

Gunjan Batra
Rutgers University, USA
gunjan.batra@rutgers.edu

Vijayalakshmi Atluri
Rutgers University, USA
atluri@rutgers.edu

Jaideep Vaidya
Rutgers University, USA
jsvaidya@rbs.rutgers.edu

Shamik Sural
IIT Kharagpur, India
shamik@cse.iitkgp.ernet.in

ABSTRACT

Discovery of Attribute Based Access Control policies through mining has been studied extensively in the literature. However, current solutions assume that the rules are to be mined from a static data set of access permissions and that this process only needs to be done once. However, in real life, access policies are dynamic in nature and may change based on the situation. Simply utilizing the current approaches would necessitate that the mining algorithm be re-executed for every update in the permissions or user/object attributes, which would be significantly inefficient. In this paper, we propose to incrementally maintain ABAC policies by only updating the rules that may be affected due to any change in the underlying access permissions or attributes. A comprehensive experimental evaluation demonstrates that the proposed incremental approach is significantly more efficient than the conventional ABAC mining.

ACM Reference Format:
Gunjan Batra, Vijayalakshmi Atluri, Jaideep Vaidya, and Shamik Sural. 2021. Incremental Maintenance of ABAC Policies. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21), April 26–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3422337.3447825

1 INTRODUCTION

The flexibility, scalability, dynamic nature, portability and identity-less features of Attribute Based Access Control (ABAC) make it an attractive choice to be employed as a means to enforce access control in many traditional and emerging application domains [8]. Under ABAC, security policies (also called rules in this paper) are specified based on subject, object and environmental attribute conditions. However, a key problem in deploying ABAC is to precisely configure it for effective access control. The problem of automatically discovering the best set of minimum ABAC rules to configure the system using existing permissions of users on the resources is known as ABAC Policy Mining.

While the ABAC mining problem has been well studied in the literature, all the approaches assume the system to be static in nature. However, in reality, all systems change in due course of time, which necessitates updating (or maintaining) the ABAC rules since the original rules may no longer be valid. Such maintenance requires *re-mining* of the ABAC policies that reflect the changes, which is often quite expensive and inefficient, especially when the number of users and objects is large and changes to the data are quite frequent.

In this paper, we propose an efficient alternative where ABAC policy maintenance can be performed in an incremental fashion whenever a change occurs. In particular, we consider the following changes (Δ). Note that other changes such as adding subjects and objects do not affect the ABAC policies.

(1) Addition of a permission: This means granting access to a user on an object that did not previously exist.
(2) Deletion of a permission: This means revoking a user's existing permission to access an object.
(3) Addition of an attribute value: This means assigning a new attribute value to either a subject or an object that currently is not assigned.
(4) Deletion of an attribute value: This means removing an existing attribute value assignment from a subject or an object.

Given the current set of ABAC policies and the set of changes Δ, the two alternative approaches to discover the new set of ABAC policies have been depicted in Figure 1. More specifically:

(1) *Redo ABAC Mining* is an intuitive method to discover new set of ABAC rules in which ABAC mining process is performed whenever there is a change.
(2) *Incremental Maintenance* is an efficient means of handling changes and discovering new ABAC rules by taking into account only the part of the data that was affected by the change. However, this process requires us to first extract and maintain some intermediary data structures using which we obtain the new set of rules that cover the changes.

In this paper, we have employed the ABAC-SRM mining algorithm proposed by Talukdar et al. [20] as this is one of the fastest approaches thus far proposed. We have compared the efficiency of the above two approaches - the redo ABAC mining and the incremental maintenance. Since the data used to mine the policies in the latter case is significantly smaller than the former approach, is very efficient as shown in our experimental evaluation.

The rest of this paper is organized as follows. In Section 2, we review the preliminaries of ABAC definition and concepts, as well as the ABAC-SRM mining approach [20]. In Section 3, we discuss the types of changes and formalize the problem of incremental maintenance of ABAC policies. We also provide an overview of the approach to solve the problem. In Section 4 we present the proposed

CODASPY '21, April 26–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8143-7/21/04…$15.00
https://doi.org/10.1145/3422337.3447825

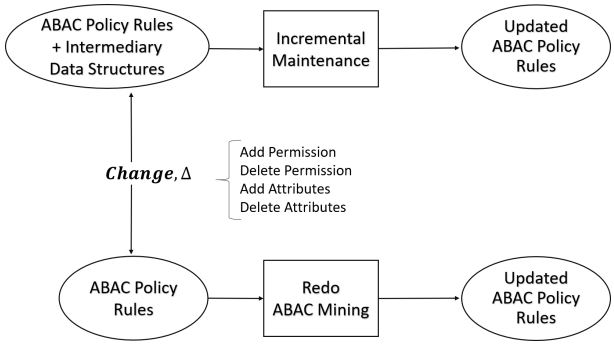


Figure 1: Alternative Approaches to Maintenance of ABAC Policies

incremental maintenance approach with detailed algorithms. This includes a pre-processing phase to extract the necessary intermediate data structures and algorithms to handle the four types of changes mentioned above. In Section 5, we experimentally compare the cost of maintaining ABAC system using our proposed incremental maintenance approach versus employing the redoing ABAC mining approach. In Section 6 we discuss the related work and in Section 7, we present conclusions and future research.

2 PRELIMINARIES

In this section, we briefly review the attribute based access control model (ABAC) [9], and the ABAC mining approach from Talukdar et al. [20], called the ABAC-SRM. The notations of ABAC and ABAC Mining approach have been borrowed from [20].

2.1 ABAC

In an ABAC system, the authorization to perform an operation (e.g., read, write) is granted based on the attributes of the requesting user, requested object, and the environment in which a request is made. The ABAC policy rules are comprised of the attributes, which include user attribute conditions and object attribute conditions that determine granting of a permission to the user.

The basic components of an ABAC system are as follows:

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

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

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

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

O_A: Represents a set of object attribute names. Members of these sets are represented as oa_i, for $1 \leq j \leq |O_A|$. Each oa_i is associated with a set of possible values it can acquire. For instance, if an object folder with records of student has object attribute *Major* associated with a set of values {*Computer Science*, *Electronics*, *Mechanical*}, then for every $o \in O$, value of the attribute *Major* can be either *Computer Science*, *Electronics* or *Mechanical*.

E_A: Represents a set of environment attribute names. Members of these sets are represented as ea_i, for $1 \leq i \leq |E_A|$. Each ea_i is associated with a set of possible values it can acquire. For instance, if an environment attribute *Campus* is associated with a set of values {*USA*, *UK*, *Australia*}, then for every $e \in E$, value of the attribute *Campus* can be either *USA*, *UK* or *Australia*.

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 P = {*read*, *write*}. Each member of P is represented as p_i, for $1 \leq i \leq |P|$.

U_C: Represents a set of all possible user attribute conditions denoted as uc_j, for $1 \leq j \leq |U_C|$. 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 *Title* has possible values {*Instructor, TA, Student*} and user attribute *Department* has possible values as {*Computer Science, Electronics, Mechanical*}, then U_C will be a set comprising of {*Title=Instructor, Title=TA, Title=Student, Title=Any, Department=Computer Science, Department=Electronics, Department=Mechanical, Department=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.

O_C: Represents a set of all possible object attribute conditions denoted as oc_k, for $1 \leq k \leq |O_C|$. 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 *Major* has possible values {*Computer Science, Electronics, Mechanical*} and object attribute *RecordsOf* has possible values {*Instructor, TA, Student, Staff*}, then O_C will be a set comprising of {*Major=Computer Science, Major=Electronics, Major=Mechanical, Major=Any, RecordOf= Instructor, RecordOf=Student, RecordOf=TA, 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 of the form $\langle uc, oc, p \rangle$.

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*.

UA: User attribute relation $UA \subseteq U \times U_C$ is a many-to-many mapping of users and user attribute conditions. We use a $m \times n$ binary

Table 1: UA

User (u)	*Department =Computer Science* (uc_1)	*Title= Instructor* (uc_2)	*Department= Electronics* (uc_3)	*Title= TA* (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: *OA*

Object (o)	*Major =Electronics* (oc_1)	*Major =Computer Science* (oc_2)	*Recordof =Student* (oc_3)
o_1	1	0	1
o_2	0	1	1

Table 3: Π

Rule ID	Rule
1	uc_3, oc_1
2	uc_1, oc_2

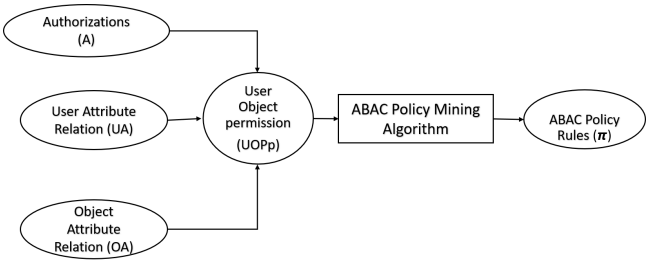


Figure 2: ABAC Mining Process

matrix to represent UA, where UA[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 *TA* whose department is *Electronics*.

OA: Object attribute relation, $OA \subseteq O \times O_C$ is a many-to-many mapping of objects and the set of all attributes conditions, where an $m \times n$ binary matrix represents OA. OA[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 *recordof Student* in *Electronics Major*.

2.2 ABAC Mining

A: An authorization a is of the form of $\langle u, o, p\rangle$ and represents user $u \in U$, object $o \in O$, and permission $p \in P$, respectively, where a denotes that 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 A. For each permissions $p_i \in P$, we define $A_{p_i} \subset A$ such that for every $a \in A_{p_i}$, $p.a = p_i$. For example, if P = {read,write}, we have $A_{\texttt{read}}$ and $A_{\texttt{write}}$ such that $A_{\texttt{read}} \cup A_{\texttt{write}} = A$.

Given A, we construct a table UOP_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 A, respectively, a column for p and column for *Row Id*. There is a row in 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 $u - o$. For the remaining rows, the p column is 0. Given $P = \{p_1, p_2 \ldots p_n\}$, $UOP = \cup_{i=1}^{n} UOP_{p_i}$.

UOP_p: User Object Permission Matrix UOP_p, is a $M \times N$ matrix, where $M = |U| \times |O|$ comprising of a row for each user-object pair, and $N = |U_C| + |O_C| + 1$, comprising of a column for each object attribute condition, a column for each user attribute condition, and a column for the permission p and a column for *Row Id*. For the UA in table 1, OA in table 2 and A in table 4a, table 4b shows the $UOP_{\texttt{read}}$ constructed.

ABAC Mining: Given the set of authorizations A, the set of user attribute conditions (U_C), the set of object attribute conditions(O_C), ABAC Mining discovers 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 $a = \langle u, o, p\rangle \in A$. Table 3 shows the rules Π corresponding to the $UOP_{\texttt{read}}$ in table 4b.

In this work, to find out Π, the ABAC-SRM Mining algorithm has been used. This algorithm finds minimum possible ABAC rules through subset enumeration. It is being used in the form of the following function: ABAC-SRM($Perm1Rules$,$Perm0Rules$,$SpecificRules$), where $Perm1Rules$ are the Permission One rows, $Perm0Rules$ are Permission Zero rows and $SpecificRules$ are current ABAC rules. This function evaluates the ABAC rules Π and returns them.

3 INCREMENTAL MAINTENANCE PROBLEM

In this section, we define the problem of maintenance of ABAC policies in an incremental fashion, and discuss the basic idea behind our proposed approach.

3.1 Types of Changes

Δ: Represents a set of possible changes allowed in an ABAC system. Each member is denoted as Δ_i for $1 \leq i \leq |\Delta|$. The change Δ could be to authorization set (A) or User Attribute Relation (UA) or Object Attribute Relation (OA) in ABAC, and makes the current rules Π invalid. Types of changes, Δ are:

(1) Addition of an Access Permission (Δ_{ap}): This means authorizing a user access to an object which previously didn't exist. $\Delta_{ap} : A \leftarrow A \cup a$, where $a = \langle m\_user, m\_object, p\rangle$.
This will lead to the permission column of a row in UOP corresponding to user m_user and object m_object to be changed from 0 to 1.

(2) Deletion of an Access Permission (Δ_{dp}): This means removing a user's existing access to an object. This will lead to the permission column of a row in UOP to be changed from 1 to 0. $\Delta_{dp} : A \leftarrow A \setminus a$ where $a = \langle m\_user, m\_object, p\rangle$.
This will lead to the permission column of a row in UOP corresponding to user m_user and object m_object to be changed from 1 to 0.

(3) Addition of an Attribute value (Δ_{aa}): This means assigning a user/object an attribute value which previously it did not have, i.e., $\Delta_{aa} : UA$[i,j] $\leftarrow 1$ where user m_user satisfies user attribute condition uc_j.
This will lead to the attribute condition being 1 for all of the user's/object's rows in the UOP.

(4) Deletion of an Attribute value (Δ_{da}) : This means removing an existing attribute value of a user/ object, i.e., $\Delta_{da} : UA$[i,j] $\leftarrow 0$, where user m_user does not satisfy user attribute condition uc_j.
This will lead to the attribute condition being 0 for all of the user's/object's rows in the UOP.

We assume that during the addition and deletion of permission changes, the user/object attribute values remain the same. Similarly,

(a) Authorizations (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_4	o_2	p_1

(b) UOP_{Read}

Row Id	*User - Object* u - o	*Department =Computer Science* (uc_1)	*Title= Instructor* (uc_2)	*Department= Electronics* (uc_3)	*Title= TA* (uc_4)	*Major =Electronics* (oc_1)	*Major =Computer Science* (oc_2)	*RecordOf =Student* (oc_3)	p_1= *Read*
1	u_1 o_1	0	0	1	1	1	0	1	1
2	u_2 o_1	0	1	1	0	1	0	1	1
3	u_3 o_1	1	1	0	0	1	0	1	0
4	u_4 o_1	1	0	0	1	1	0	1	0
5	u_1 o_2	0	0	1	1	0	1	1	0
6	u_2 o_2	0	1	1	0	0	1	1	0
7	u_3 o_2	1	1	0	0	0	1	1	1
8	u_4 o_2	1	0	0	1	0	1	1	1

Table 4: The set of Authorizations and the corresponding UOP_{Read}

we assume that during the addition and deletion of attribute values, the permissions of users on objects remain unchanged. We would also like to note that addition/deletion of environment attributes will be handled similar to user/object attributes.

Definition 1 (Incremental ABAC Policy Maintenance). *Given a set of ABAC rules* Π *that encapsulate the original access policy, and the change* $\triangle$*, the Incremental ABAC Maintenance Problem aims to discover a set of rules* Π' *that encapsulates the new access policy derived by applying the changes* Δ *more efficiently than re-mining the new policy.*

3.2 Overview of the approach

The concept behind performing incremental mining is that whenever a change occurs, it impacts only a particular row in the *UOP* or a small set of rows in the *UOP*. We only need to work on that part of the *UOP* to update the ABAC rules. Intuitively, we need to replace a few rows in the *UOP* with new ones which have the change incorporated in them. We can have two types of rows in *UOP* that can be added or deleted from the *UOP*:

(1) Rows with value of Permission column as 1, called Permission One Rows.
(2) Rows with value of Permission column as 0, called Permission Zero Rows.

For example, in Table 5, the Permission One Rows are [3,5,6,8] and Permission Zero Rows are [1,2,4,7,9]. For every type of change, we delete the Permission Zero and Permission One rows impacted and add the modified/changed Permission Zero and Permission One rows back to the data set (or data structures that we maintain for efficient updation). Our approach to incremental ABAC maintenance is based on the above concept.

4 INCREMENTAL MAINTENANCE APPROACH

In this section, we describe in detail our proposed approach to perform the incremental maintenance of ABAC policies. We first discuss the steps performed during the pre-processing phase and then discuss the incremental maintenance algorithms for the four change types.

4.1 Pre-Processing

The incremental ABAC maintenance approach is based on the idea that we maintain certain intermediary data structures on which we can operate to find out the new set of ABAC rules Π'. Specifically, given the set of ABAC rules Π (i.e, the original policy) and the *UOP* derived from the policy, we maintain the following data structures:

(1) *ZeroRowIds*: The Row Ids of Permission Zero Rows in the *UOP*.
(2) *ZeroRules*: This is a minimum set of Permission Zero Rows which covers all the Permission Zero rows, such that each Zero row is a subset of atleast one rule in *ZeroRules*.
(3) *ZeroRowCoverage*: This is a list of size |*ZeroRules*| where each element is a set that gives the Row Ids in the *UOP* of the Permission Zero rows that are a subset of the corresponding *ZeroRule* (i.e., it provides a mapping of the *ZeroRules* to the Rows Ids of the Permission Zero rows covered by the corresponding rule in *ZeroRules*).
(4) *ZeroRuleIds*: The Row Ids in the *UOP* of the *ZeroRules* rows.
(5) *OneRowIds*: The Row Ids in the *UOP* of the Permission One Rows.
(6) *OneRowCoverage*: This is a list of size |Π| where each element is a set that gives the Row Ids in the *UOP* of the Permission One rows that are a super set of the corresponding ABAC rule in Π (i.e., it provides a mapping of the Π to the Row Ids of the Permission One Rows authorized by the corresponding ABAC rule in Π).

Algorithm 1 Pre-Processing Steps

Require: Dataset $D = UOP$
Require: Π
1: *ZeroRowIds* ← Row Ids of Permission Zero Rows
2: *ZeroRules* ← findZeroRules (*ZeroRowIds*)
3: *ZeroRowCoverage* ← *CoverageCalc*(*ZeroRowIds*, *ZeroRules*, 0)
4: *ZeroRuleIds* ← Row Ids of *ZeroRules*
5: *OneRowIds* ← Row Ids of Permission One Rows
6: *OneRowCoverage* ← *CoverageCalc*(*OneRowIds*, Π, 1)

We will denote the entire set of the above data structures by I. Algorithm 1 gives the specific steps to construct these, and utilizes

Algorithm 2 findZeroRules(*ZeroRowids*)

1: *ZeroRules*, *ZeroRowList* ← ϕ
2: *ZeroRowList* ← List of set of attribute conditions of *Ids* in *ZeroRowIds*
3: Sort *ZeroRowList* in descending order based on the number of attributes in each row
4: **for** each *row* in *ZeroRowList* **do**
5: **if** $\nexists$*rule* ∈ *ZeroRules* such that *row* ⊆ *rule* **then**
6: *ZeroRules* ← *ZeroRules* ∪ *row*
7: **end if**
8: **end for**
9: return *ZeroRules*

Algorithm 3 CoverageCalc(*RowIds*, *Rules*, *type*)

1: *coverage* ← {$\phi, \ldots, \phi$}
2: **for** each *rule* in *Rules* **do**
3: **for** each *Id* in *RowIds* **do**
4: **if** (*type*==0 and attributes conditions of *Id* ⊆ *rule*) OR (*type*==1 and *rule* ⊆ attributes conditions of Id) **then**
5: *coverage*[*rule*] ← *coverage*[*rule*] ∪ *Id*
6: **end if**
7: **end for**
8: **end for**
9: return *coverage*

two helper functions: i) findZeroRules (Algorithm 2) and ii) CoverageCalc (Algorithm 3). findZeroRules takes as input *ZeroRowIds* and returns the minimum set of Permission Zero Rows that encapsulate all the Zero rows such that each Zero row is a subset of atleast one *ZeroRule*. This is done by sorting the Zero Rows in descending order based on the number of attribute conditions in each row and then by adding each row that is not already covered by some rule to the *ZeroRules*.

The CoverageCalc function is used to find *OneRowCoverage* and *ZeroRowCoverage*. CoverageCalc takes as input the *OneRowIds* or *ZeroRowIds*, Π or *ZeroRules* and *type* as 1 for *OneRowCoverage* and 0 for *ZeroRowCoverage*. It returns the corresponding list of sets called *coverage* in Line 9 (*OneRowCoverage* or *ZeroRowCoverage* depending on the type). This is accomplished by simply iterating over each rule and each row provided as input and checking if the rule is a subset (superset) of the role if the *ZeroRowCoverage* (correspondingly, *OneRowCoverage*) is requested.

Example 1. Consider the *UOP* given in Table 5. The existing ABAC rules in the system are : [(uc_3, oc_2), (uc_1, uc_2, oc_1)]. By performing the **Pre-processing Steps**, we get:
ZeroRowIds = [1, 2, 4, 7, 9]
ZeroRules = [(uc_1, uc_2, oc_2, oc_3), (uc_2, oc_1, oc_2), (uc_1, uc_3, oc_1)]
ZeroRowCoverage = [(7, 9), (1, 4), (2)]
ZeroRuleIds = [9, 4, 2]
OneRowIds = [3, 5, 6, 8]
OneRowCoverage = [(3, 6), (5, 8)]

Table 5: Illustrative example: *UOP*

Row Id	*u - o*	uc_1	uc_2	uc_3	oc_1	oc_2	oc_3	*p*
1	u_1 o_1	0	1	0	1	0	0	0
2	u_2 o_1	1	0	1	1	0	0	0
3	u_3 o_1	1	1	0	1	0	0	1
4	u_1 o_2	0	1	0	1	1	0	0
5	u_2 o_2	1	0	1	1	1	0	1
6	u_3 o_2	1	1	0	1	1	0	1
7	u_1 o_3	0	1	0	0	1	1	0
8	u_2 o_3	1	0	1	0	1	1	1
9	u_3 o_3	1	1	0	0	1	1	0

Algorithm 4 AddPermission($D, \Pi, m\_user, m\_object, I$)

1: *m_rowid* ← Row Id of *m_user* and *m_object* in the *UOP*
2: *m_row* ← attribute condition set of *m_row*
3: Delete_Perm0Rows ([*m_rowid*])
4: Add_Perm1Rows ([*m_row*],[*m_rowid*])

4.2 Handling Add Permission

Algorithm 4 gives the procedure to grant *m_user* the permission to access *m_object*. This is done by removing the corresponding row from the list of permission zero rows and adding to the list of permission one rows. Two helper functions are used to accomplish this: i) Delete_Perm0Rows (Algorithm 5) and ii) Add_Perm1Rows (Algorithm 6).

Algorithm 5 Delete_Perm0Rows(*del0rowsIds*)

1: *R_ZeroRules*, *R_RuleIds* ← ϕ
2: **for** each *Id* ∈ *del0rowsIds* **do**
3: *ZeroRowIds* ← *ZeroRowIds* \ *Id*
4: **for** each *cover* ∈ *ZeroRowCoverage* **do**
5: **if** *Id* ⊆ *cover* **then**
6: **if** *Id* ∈ *ZeroRuleIds* **then**
7: *R_ZeroRules* ← *R_ZeroRules* ∪ findZeroRules (*cover* \ *Id*)
8: *R_RuleIds* ← *R_RuleIds* ∪ Index of *cover*
9: **else**
10: *cover* ← *cover* \ *Id*
11: **end if**
12: **end if**
13: **end for**
14: **end for**
15: Delete rules in *ZeroRules* at row numbers in *R_RuleIds*
16: Delete covers in *ZeroRowCoverage* at row numbers in *R_RuleIds*
17: *R_ZeroRowCoverage* ← *CoverageCalc*(*ZeroRowIds*, *R_ZeroRules*, 0)
18: *ZeroRules* ← *ZeroRules* ∪ *R_ZeroRules*
19: *ZeroRowCoverage* ← *ZeroRowCoverage* ∪ *R_ZeroRowCoverage*
20: *ZeroRuleIds* ← findZeroRuleRows (*ZeroRules*)

Delete_Perm0Rows (*del0rowsIds*) Algorithm 5 gives the specific steps for removing a permission. First, we remove the row from

the permission zero rows (Line 3). Now, we update the intermediary data structures. First we update the *ZeroRowCoverage* mapping. If the permission row to be removed is a rule in *ZeroRules* then we need to recompute the rule by removing it from the set of zero permission rows covered by that rule (lines 7-8). Off course, if this is the only zero permission row represented by that rule, then it is eliminated altogether. Otherwise the permission row is removed from the cover of every rule in *ZeroRules* it belongs to (line 10). Lines 15-20 update the *ZeroRules*, *ZeroRuleIds*, and the *ZeroRowCoverage* by first removing the appropriate rules and affected covers, and then adding in the additional rules prior calculated (at line 7) and recalculating the cover corresponding to those rules (line 17).

Add_Perm1Rows(*add1rows*,*add1rowsIds*): Algorithm 6 gives the specific steps for adding a permission one row. This function adds the rows given in *add1rows* to the ABAC system and re-mines the policy. To do this, the ABAC-SRM function is used, but instead of passing in the entire *UOP*, only the *ZeroRules*, *add1rows* and Π are used since they encapsulate the entire *UOP*, and therefore the mining is much more efficient, as confirmed by the experimental evaluation. We then update the *OneRowIds* and the *OneRowCoverage* using CoverageCalc (Algorithm 3).

Example 2 (Add Permission: u_3 gets access o_3). Granting u_3 access to o_3, means we have to make the permission in RowId 9 equal to 1. This effectively means we have to delete the Permission Zero, Row Id 9 from the *UOP* and add a Permission One row with the same attribute conditions. Therefore, Algorithm 4 calls Delete_Perm0Rows with [9] and Add_Perm1Rows with argument ([(uc_1,uc_2,oc_2,oc_3)],[9]), obtaining the output given below:

m_row = (uc_1, uc_2, oc_2, oc_3) , m_rowid = 9

Delete_Perm0Rows([9])

ZeroRowIds = [1,2,4,7]

ZeroRules = [(uc_2, oc_1, oc_2), (uc_1, uc_3, oc_1), (uc_2, oc_2, oc_3)]

ZeroRowCoverage = [(1,4),(2),(7)]

ZeroRuleIds = [4,2,7]

Add_Perm1Rows([(uc_1,uc_2,oc_2,oc_3)],[9])

Π = [(uc_3, oc_2), (uc_1, uc_2)]

OneRowIds = [3,5,6,8,9]

OneRowCoverage = [(5,8),(3,6,9)]

4.3 Handling Delete Permission

Algorithm 7 gives the specific steps to revoke *m_user*'s permission to access *m_object*. This is done by removing the corresponding row from the list of permission one rows and adding to the list of permission zero rows. Two helper functions are used to accomplish this: i) Delete_Perm1Rows (Algorithm 8) and ii) Add_Perm0Rows (Algorithm 9).

Delete_Perm1Rows (*del1rowsIds*): Algorithm 8 gives the specific steps to delete permission one rows from the ABAC system. Note that whenever a permission one row is deleted, some ABAC rules become invalid. These rules are found in Line 2, using *OneRowCoverage*. Every *Id* in *del1rowsIds* is checked in every *cover* of *OneRowCoverage*. If an *Id* is in a cover, the index of the cover is added to *ridinvalid*. The numbers in *ridinvalid* signify that those rules in Π have become invalid. Next, in Lines 3 - 5, for all *r* in *ridinvalid*, *minerowids* is evaluated, which is *cover* at *r* (in *OneRowCoverage*) after the *del1rowsIds* are deleted from it. The Π and *covers* in *OneRowCoverage* are deleted at row numbers in *ridinvalid* in Line 6.

Algorithm 6 Add_Perm1Rows(*add1rows*,*add1rowsIds*):

1: Π ← ABAC-SRM (*add1rows*,*ZeroRules*,Π)
2: *OneRowIds* ← *OneRowIds* ∪ *add1rowsIds*
3: *OneRowCoverage* ← CoverageCalc(*OneRowIds*, Π, 1)

Algorithm 7 DeletePermission(D, Π, *m_user*, *m_object*)

1: *m_rowid* ← Row Id of *m_user* and *m_object* in the *UOP*
2: *m_row* ← Attribute condition set of *m_rowid*
3: Delete_Perm1Rows ([*m_rowid*])
4: Add_Perm0Rows ([*m_row*],[*m_rowid*])

Algorithm 8 Delete_Perm1Rows (*del1rowsIds*)

1: *ridinvalid*, *minerowids*, *minerows* ← ϕ
2: *ridinvalid* ← Indices of *covers* in *OneRowCoverage* having any *Id* from *del1rowsIds*
3: **for** *r* in *ridinvalid* **do**
4: *minerowids* ← *minerowids* ∪ (*cover* of *OneRowCoverage* at *r* \ *del1rowsIds*)
5: **end for**
6: Delete rules in Π and covers in *OneRowCoverage* at row numbers in *ridinvalid*
7: *minerows* ← List of attribute condition sets of rows in *minerowids*
8: Π ← AbacMining (*minerows*,*ZeroRules*,Π)
9: *OneRowIds* ← *OneRowIds* \ *del1rowsIds*
10: *OneRowCoverage* ← CoverageCalc (*OneRowIds*,Π,1)

Algorithm 9 Add_Perm0Rows(*add0rows*, *add0rowsIds*)

1: *ridinvalid*, *minerowids*, *minerows* ← ϕ
2: *ridinvalid* ← Indices of *rules* in Π that are subset of any *row* from *add0rows*
3: *minerowids* ← row Ids covering all *ridinvalid* (from *OnerowCoverage*)
4: *minerows* ← List of attribute condition sets of rows in *minerowids*
5: Π ← Π \ *ridinvalid*
6: Delete cover in *OneRowCoverage* at all numbers in *ridinvalid*
7: **for** *row* in *add0rows* **do**
8: **for** *zrule* in *ZeroRules* **do**
9: **if** *zrule* ⊂ *row* **then**
10: replace *zrule* with *row*
11: **else if** *row* $\not\subseteq$ *zrule* **then**
12: *ZeroRules* ← *ZeroRules* ∪ *row*
13: **end if**
14: **end for**
15: **end for**
16: *ZeroRowIds* ← *ZeroRowIds* ∪ *add0rowsIds*
17: Update *ZeroRuleIDs*, *ZeroRowCoverage*
18: Π ← ABAC-SRM(*minerows*, *ZeroRules*, Π)

The list of attribute conditions of the rows in *minerowids* is put in *minerows* in Line 7. In Line 8, *rowsleftattributes* are then used to mine new Π using the algorithm ABAC-SRM. In Lines 9 - 10, *OneRowIds* are updated by removing the *del1rowsIds* from them and *OneRowCoverage* is also updated by using *CoverageCalc* function in Algorithm 3.

Add_Perm0Rows(*add0rows*, *add0rowsIds*): Algorithm 9 gives the specific steps to add permission zero rows. It adds the *add0rows* to the ABAC system. When permission zero rows are added, some rule in Π may become invalid. If an ABAC rule is subset of a Permission Zero row, the rule becomes invalid. In Line 2, we find any such rules that become invalid. For each *row* in *add0rows*, and for all Π, if any *rule* in Π is a subset of any *row* in *add0rows*, index of that *rule* is added to *ridinvalid*. In Line 3, for all rules in *ridinvalid*, the row Ids in *OneRowCoverage* are added to *minerowids* and in Line 4, the attribute conditions of all rows in row Ids of *minerowids* are made into list and added to *minerows*. Next, in Line 5 and 6, at the row numbers in *ridinvalid*, we delete the Π and *covers* in *OneRowCoverage*. In Lines, 7-15, we update the *ZeroRules* by adding the *add0rows*. If a rule in *ZeroRules* is a subset of a row in *add0rows*, then the rule is replaced with the row in *add0rows*; the row in *add0rows* is added to *ZeroRules* if it is not subset of any rule in *ZeroRules*. In Line 16, the *add0rowsIds* are added to *ZeroRowIds*. Also in Line 17, *ZeroRuleIds* and *ZeroRowCoverage* are updated. Finally, in Line 18, Π are updated using the ABAC-SRM mining algorithm. As earlier, the mining is much more efficient in practice since it only utilizes the *ZeroRules* and the Π as opposed to the entire *UOP*.

Example 3 (Removing u_2's access to o_3). This means we have to make the permission in RowId 8 equal to 0. This effectively means we have to delete the Permission One, Row Id 8 from the *UOP* and add a Permission Zero row with same attributes conditions. Therefore, Algorithm 7 calls Delete_Perm1Rows with [8] and Add_Perm0Rows with [(uc_1,uc_3,oc_2, oc_3)],[8]), obtaining the output given below:
m_row = (uc_1,uc_3,oc_2,oc_3), m_rowid = 8
Delete_Perm1Rows([8])
Π = [(uc_1, uc_2, oc_1), (uc_3, oc_2)]
OneRowIds = [3,5,6]
OneRowCoverage = [(3,6),(5)]
Add_Perm0Rows([(uc_1,uc_3,oc_2, oc_3)],[8])
ZeroRules = [(uc_1, uc_2, oc_2, oc_3), (uc_2, oc_1, oc_2), (uc_1, uc_3, oc_1), (uc_1, uc_3, oc_2, oc_3)]
ZeroRuleRows = [9,4,2,8]
ZeroRowIds = [1,2,4,7,9,8]
Π = [(uc_1, uc_2, oc_1), (uc_3, oc_1, oc_2)]
OneRowCoverage = [(3,6),(5)]
ZeroRowCoverage = [(7,9),(1,4),(2),(8)]

4.4 Handling Add Attribute Value

Algorithm 10 gives the specific steps to add the attribute value *m_attribute* to user *m_user*. This is done in four steps: i) we extract all permission zero rows and permission one rows for m_user and the corresponding attribute sets (lines 1-4); ii) we remove the corresponding rows from the permission zero and permission one rows; iii) we update the attributes sets; iv) we insert the updated rows into the permission zero rows and permission one rows. To ensure that the intermediary data structures are consistently maintained, the four prior defined functions i) Delete_Perm0Rows (Algorithm 5); ii) Delete_Perm1Rows (Algorithm 8); iii) Add_Perm0Rows (Algorithm 9) and iv) Add_Perm1Rows (Algorithm 6) are used to accomplish this. Note that an attribute value can also be added to an object in exactly the same way.

Algorithm 10 AddAttribute(D, Π, *m_user*, *m_attribute*)

1: *m_rowids*1 ← List of Row Ids of Permission One Rows of *m_user* in the *UOP*
2: *m_rowids*0 ← List of Row Ids of Permission Zero Rows of *m_user* in the *UOP*
3: *m_rows*1 ← List of attribute condition sets of all rows in *m_rowids*1
4: *m_rows*0 ← List of attribute condition sets of all rows in *m_rowids*0
5: Delete_Perm0Rows (*m_rowids*0)
6: Delete_Perm1Rows (*m_rowids*1)
7: Add the *m_attribute* to the *m_rows*0 and *m_rows*1
8: Add_Perm0Rows (*m_rows*0,*m_rowids*0)
9: Add_Perm1Rows (*m_rows*1,*m_rowids*1)

Example 4 (Add Attribute value: u_1 gets uc_3). Adding attribute condition uc_3 to user u_1, requires removing all the rows of user u_1, i.e. Row Ids 1, 4 and 7. Then we have to add the attribute condition uc_3 to the attribute condition set of these rows and add them back to the data set. Therefore, Algorithm 10 calls Delete_Perm0Rows with [1, 4, 7], Delete_Perm1Rows with ϕ, Add_Perm0Rows with [(uc_2,uc_3,oc_1),(uc_2,uc_3,oc_1,oc_2), (uc_2,uc_3,oc_2,oc_3)],[1,4,7] and Add_Perm1Rows with (ϕ, ϕ) obtaining the output given below:
*m_rowids*0= [1,4,7], *m_rowids*1= ϕ
*m_rows*0= [(uc_2,oc_1),(uc_2,oc_1,oc_2),(uc_2,oc_2,oc_3)], *m_rows*1= ϕ
Delete_Perm0Rows ([1,4,7])
ZeroRowIds = [2,9]
ZeroRules = [(uc_1, uc_2, oc_2, oc_3), (uc_1, uc_3, oc_1)]
ZeroRowCoverage = [(9),(2)]
ZeroRuleIds = [2,9]
Delete_Perm1Rows (ϕ)
Add_Perm0Rows ([(uc_2,uc_3,oc_1),(uc_2,uc_3,oc_1,oc_2), (uc_2,uc_3,oc_2,oc_3)],[1,4,7])
ZeroRules =[(uc_1, uc_2, oc_2, oc_3), (uc_1, uc_3, oc_1), (uc_2, uc_3, oc_1, oc_2), (uc_2, uc_3, oc_2, oc_3)]
ZeroRowIds = [2,9,1,4,7]
ZeroRuleIds = [9,2,4,7]
Π = (uc_1, uc_2, oc_1)(uc_1, uc_3, oc_3)
ZeroRowCoverage = [(9),(2),(4,1),(7)]
OneRowCoverage = [(3,6),(5,8)]
Add_Perm1Rows (ϕ,ϕ)

4.5 Handling Delete Attribute Value

Algorithm 11 gives the specific steps to remove attribute value *m_attribute* from user *m_user*. This works exactly the same as Algorithm 10, except that in line 7, we delete *m_attribute* from

the permission zero and permission one rows for *m_user*, before updating all the intermediary data structures.

Algorithm 11 DeleteAttribute($D, \Pi, m\_user, m\_attribute$)

1: $m\_rowids1 \leftarrow$ List of Row Ids of Permission One Rows of m_user in the UOP
2: $m\_rowids0 \leftarrow$ List of Row Ids of Permission Zero Rows of m_user in the UOP
3: $m\_rows1 \leftarrow$ List of attribute condition sets of all rows in $m\_rowids1$
4: $m\_rows0 \leftarrow$ List of attribute condition sets of all rows in $m\_rowids0$
5: Delete_Perm0Rows($m\_rowids0$)
6: Delete_Perm1Rows ($m\_rowids1$)
7: Delete the $m\_attribute$ from the m_rows0 and m_rows1
8: Add_Perm0Rows (m_rows0,$m\_rowids0$)
9: Add_Perm1Rows (m_rows1,$m\_rowids1$)

Example 5 (Delete Attribute value: u_2 loses uc_1). Removing attribute condition uc_1 from user u_2 requires removing all the rows of user u_2, i.e. Row Ids 2, 5 and 8. Then we have to remove the attribute condition uc_1 from the attribute condition set of these rows and add them back to the dataset. Therefore, Algorithm 11 calls Delete_Perm0Rows with [2], Delete_Perm1Rows with [5, 8], Add_Perm0Rows with [(uc_3,oc_1)], [2] and Add_Perm1Rows with [(uc_3,oc_1,oc_2),(uc_3,oc_2,oc_3)],[5,8] obtaining the output given below:

$m\_rowids1$ = [5,8], $m\_rowids0$ = [2]
m_rows1 = [(uc_1,uc_3,oc_1,oc_2),(uc_1,uc_3,oc_2,oc_3)]
m_rows0 = [(uc_1,uc_3,oc_1)]
Delete_Perm0Rows([2])
$ZeroRules$ = $(uc_1, uc_2, oc_2, oc_3), (uc_2, oc_1, oc_2)$
$ZeroRowCoverage$ = [(7,9),(1,4)]
$ZeroRuleIds$ = [9,4]
Delete_Perm1Rows([5,8])
Π = $[(uc_1, oc_1)]$
$OneRowIds$ = [3,6]
$OneRowCoverage$ = [(3,6)]
Add_Perm0Rows([(uc_3,oc_1)], [2])
$ZeroRules$ = $[(uc_1, uc_2, oc_2, oc_3), (uc_2, oc_1, oc_2), (uc_3, oc_1)]$
$ZeroRowIds$ = $[1, 4, 7, 9, 2]$
$ZeroRuleIds$ = $[9, 4, 2]$
Π = $[(uc_1, oc_1)]$
$OneRowCoverage$ = [(3,6)]
$ZeroRowCoverage$ = [(7,9),(1,4),(2)]
Add_Perm1Rows([(uc_3,oc_1,oc_2),(uc_3,oc_2,oc_3)],[5,8])
Π = $[(uc_1, oc_1), (uc_3, oc_2)]$
$OneRowIds$ = [3,6,5,8]
$OneRowCoverage$ = [(3,6),(5,8)]

5 EXPERIMENTAL EVALUATION

We carried out detailed experiments to study the effectiveness of the proposed approach in a wide variety of settings, and to understand the effect of different factors. There are no real datasets available that capture the incremental change in ABAC policies. So we ran experiments on synthetic datasets and controlled the degree of changes and examined the effect of each parameter independently.

Parameter	Default Value
Users	40
Objects	60
User Attributes	60
Object Attributes	60
Rules	40

Table 6: Default values of Parameters

We have used the synthetic dataset generator introduced by Talukdar et al. in [20] to create ABAC policies. The input data set for the four ABAC maintenance algorithms and ABAC Mining (using ABAC-SRM) are the same.

ABAC Maintenance algorithms for each type of change were implemented considering a single modification at a time and without putting constraints to perform a fair comparison with ABAC-SRM algorithm. The ABAC Maintenance algorithms can handle multiple permissions and constraints.

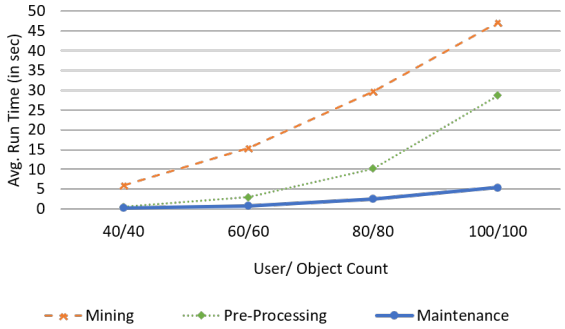
For each set of parameter values, synthetic data was created. The default values for the various parameters are given in Table 6. In each experiment, we vary one parameter of interest while the rest are set to the default values given in Table 6.

ABAC rules were mined from the dataset using the ABAC-SRM algorithm. While using the proposed ABAC Maintenance algorithm, pre-processing steps were carried out and recorded as Pre-Processing Time. Further, for every change type (four ABAC Maintenance Algorithms: Add Permission, Delete Permission, Add Attribute value, Delete Attribute value), a random possible input (user and object or user and attribute value) is identified and both ABAC Maintenance and ABAC-SRM algorithms are used to accommodate the change. For all four change types, this process was repeated for 20 random possible inputs on a data set. For Add attribute value and Delete attribute value, we have performed experiments for addition and deletion of attribute in users. The results will be similar for object attributes and environment attributes.

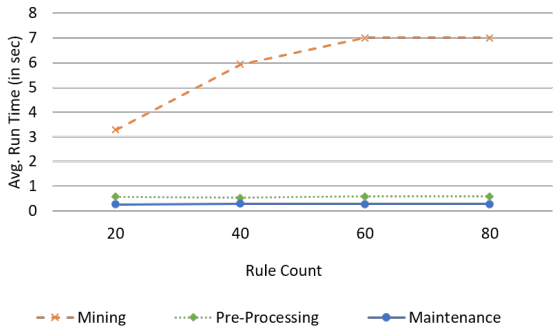
Our objective is to compare the difference in time taken by the two approaches for different types of changes. Therefore, we observed: (I) the time taken for redoing the ABAC Mining from scratch; (II) the time taken for pre-processing to generate the intermediary data structures; and (III) the time taken for incrementally updating the policy as well as the intermediary data structures.

Note that without incremental maintenance, the time required for any change is given by (I). If incremental maintenance is used, then the time given by (II) is a one-time cost, while the time given by (III) is incurred for each change. From the figures, it is observed that for all the cases, ABAC Maintenance (III) is more efficient than ABAC-SRM (I). In fact, ABAC Maintenance (III) with the Pre-processing steps (II) also performs better than ABAC-SRM (I).

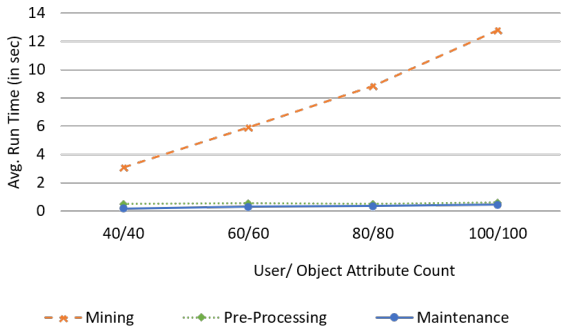
Figure 3 gives the average running time for adding a permission. Figure 3a shows the time taken when the User-Object Count is varied. It can be observed that the time taken for mining, pre-processing and incremental update all increase linearly. However, the overall time taken by the incremental approach is significantly less. Figure 3b shows the time taken when the rule count is varied.



(a) Varying User-Object Count



(b) Varying Rule Count



(c) Varying User-Object Attribute Count

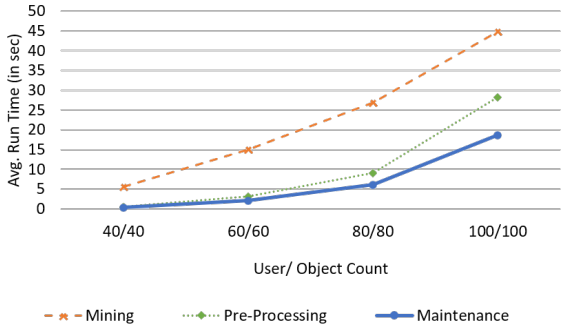
Figure 3: Running Time for Add Permission

Here, it can be observed that the time taken by the incremental approach is independent of the number of rules, where as the time taken for mining increases, though it plateaus after about 60 rules. The overall time taken by the incremental approach is again significantly less. Finally, Figure 3c shows the results when the user-object attribute count is varied. Here, it is observed that the time taken by the incremental approach is again independent of the number of attribute values, whereas the time required for mining increases linearly with it. Figures 4, 5, and 6 give the average running time for deleting a permission, adding an attribute value, and deleting an attribute value respectively. The behavior in all three cases is very similar to that in Figure 3.

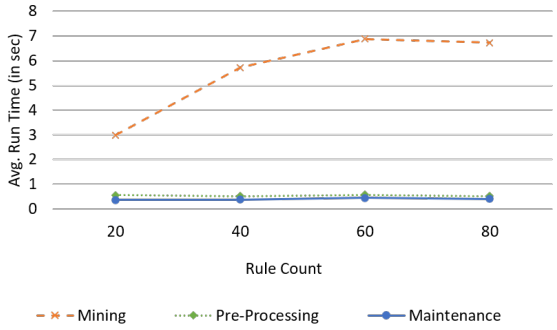
Quality of the results:
We also looked at the rules generated by our incremental approach and compared them to the rules obtained directly through mining. The experiments show that over the course of all 960 runs, in 598 of the cases, the rules generated are *exactly the same*, and only in 362 cases are the results different. Furthermore, even in those 362 cases, the number of rules generated by our incremental approach is no more than 3% more than the number of rules obtained directly through mining. This clearly shows that the quality of the results obtained through the incremental approach is very close to that obtained through mining.

6 RELATED WORK

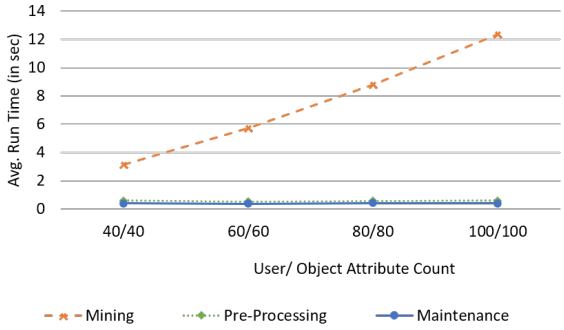
To the best of our knowledge, maintenance of ABAC policies has been studied for the first time in this paper. Hence, there are no previous work directly related to this problem. However, there are works in field of ABAC Mining, incremental FD mining and incremental maintenance of databases, which are discussed below.
ABAC Mining: The standard for ABAC model can be found in [9]. ABAC Mining has been studied by many researchers. Xu.et al. [23] are the first to propose an approach for ABAC Mining. Their approach iterates over tuples in the user-permission relation and constructs candidate rules. Then it generalizes the candidate rule to cover additional tuples by using merging techniques. Medvet et al. [16] have proposed an evolutionary approach that is separate and conquer algorithm, where at every iteration, a new rule is generated and the set of access requests is reduced to a smaller size. The efficiency of this approach is not very different when compared to that in [23]. Mocanu et al. [17] have proposed a deep learning model (Restricted Boltzmann Machines) trained on logs to generate candidate rules. Their system is still under development. Iyer et al. [10] have presented an approach to mine ABAC policies having both positive and negative authorization rules. The approach is based on the rule mining algorithm called PRISM. Gautam et al. [7] have discussed a constrained policy mining algorithm that generates a set of ABAC rules, such that the total weight of all the rules in the mined policy is minimum and no individual rule



(a) Varying User-Object Count



(b) Varying Rule Count



(c) Varying User-Object Attribute Count

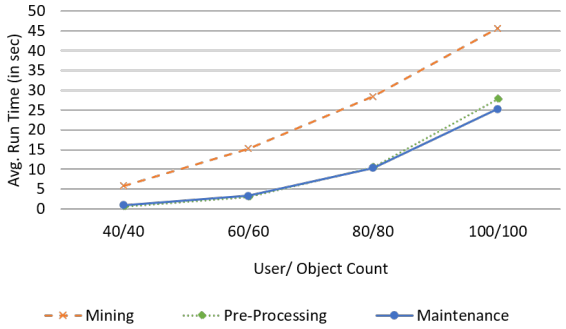
Figure 4: Running Time for Delete Permission

can have weight greater than a pre-specified constraint. Cotrini et al. [5] have proposed an algorithm for mining ABAC rules from sparse access logs. The algorithm, called Rhapsody, is built upon subgroup discovery algorithm called APRIORI-SD.

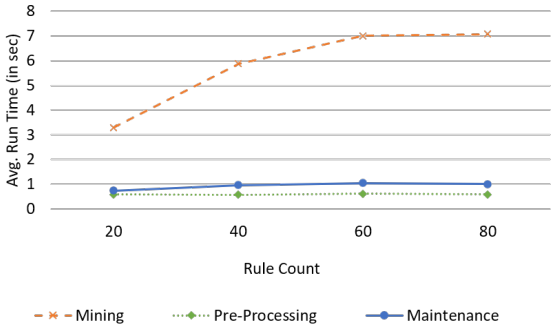
Karimi and Joshi [13] have proposed an approach to apply clustering algorithms over the decision examples to predict rules. Karimi et al. [14] have also proposed an unsupervised learning-based technique for detecting patterns in a set of access records and extracting ABAC policy rules from these patterns. They have presented two algorithms, rule pruning and policy refinement that improve quality of mined policy. The latter is useful in ABAC policy maintenance. Bertino et al. [12] have proposed an approach, called Polisma, for learning ABAC policies. It combines data mining, statistical, and machine learning techniques, with potential context information obtained from external sources to learn a better model. Iyer et al. [11] have proposed an algorithm for mining ReBAC policies, and an approach to mine graph transition policies. Gupta et al. [8] have proposed dynamic groups with attribute-based access control model for smart cars ecosystem.

Talukdar et al. [20] have used a subset enumeration approach to discover the ABAC rules in a bottom-up fashion. In this paper, we have chosen to develop our incremental maintenance approach on the ABAC-SRM mining algorithm proposed by [20] as they have shown that they take an order of magnitude less time than [23]. Also, both the algorithms ([20] and [23]) discover almost the same number of rules; whereas [20] has better WSC [18] of the rules than [23]. It is to be noted that the mining approaches in [20] and [23] do not consider unrepresented attribute value combinations [2] as these do not exist in the current authorization set. Better mining algorithms will need to be devised to tackle this issue. However, with respect to maintenance, if an existing user is authorized access to an object with attribute values from unrepresented set, the incremental mining approach can handle this in terms of updating the policy by deleting the previous attribute and adding the new attribute to the user/object. We require the maintenance algorithm to be as fast as possible and to the best of our knowledge, the time results of this algorithm are one of the best. While all the above approaches focus on mining of ABAC policies, none of them attempt to perform incremental maintenance of the policies when changes occur. The idea in [20] is that ABAC rules are nothing but a set of Functional Dependencies. Based on this premise, we have performed a review of incremental FD mining algorithms which also inspired us to develop and implement our approach.

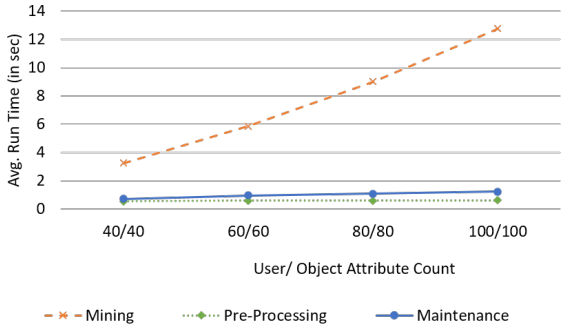
Incremental FD Mining: Caruccio et al. [1] represent FDs in a bit-vector. Their approach employs an upward and downward search strategy, and updates the set of Functional Dependencies on addition of new tuples to the database. Wang et al. [21] have proposed to add new set of tuples to the database based on concept of tuple partitions and the monotonicity of the Functional Dependencies and avoid re-scanning of the database. In [22], Wang et al. have



(a) Varying User-Object Count



(b) Varying Rule Count



(c) Varying User-Object Attribute Count

Figure 5: Running Time for Add Attribute Value

also given a Functional Dependency maintenance algorithm which deals with tuple deletion. Also, Gasmi et al. [6] have proposed an algorithm to maintain the canonical Functional Dependencies incrementally when a new tuple is appended to the original database. Our approach is mostly inspired by the work of Schirmer et al.'s DynFD [19]. They maintain Functional Dependencies for a dynamic dataset using a bottom up and top down approach, and indices to evolve the Functional Dependencies. For a batch of operations, insert, update and delete of data, the algorithm adapts its validation data structures, and a positive and negative cover of Functional Dependencies.

Incremental Maintenance in Databases: Besides the above, incremental maintenance has been studied extensively in data bases. Cheung et al. [3] use incremental updating technique to update the association rules in large databases whenever new transactions are added to the database. Also in [4], Cheung et al. have proposed algorithms for frequent pattern mining in a database and to allow mining in a single pass over the database as well as insertion or deletion of transactions in an efficient manner.

Lin et al. [15] have proposed an efficient incremental algorithm for transaction insertion. The algorithm reduces computations without candidate generation and is based on the utility-list structures.

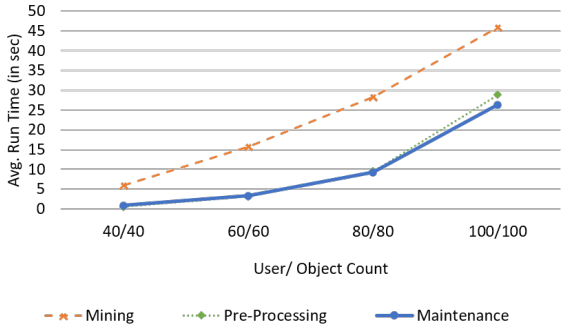
7 CONCLUSIONS AND FUTURE WORK

In this paper, we have examined the problem of maintenance of ABAC policies in an incremental fashion. We have formalized this problem and proposed an efficient algorithm to maintain ABAC policies considering four types of changes. We have experimentally evaluated the performance of the proposed algorithm and showed that it takes significantly less time to perform a change when compared to redoing ABAC mining from scratch.

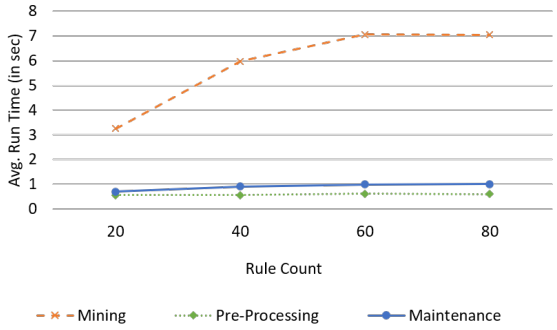
In the future, we plan to develop an approach that would allow batching of changes and optimally switch between incremental maintenance and re-mining. We plan to study the the consistency and scalability aspects of this approach. We would also like to implement the algorithm for negative authorizations and by adding noise. Also, we plan to implement the ABAC maintenance approach when there are constraints such as mutual exclusion, preconditions, obligations, SoD, attribute-attribute conditions, etc.

ACKNOWLEDGMENTS

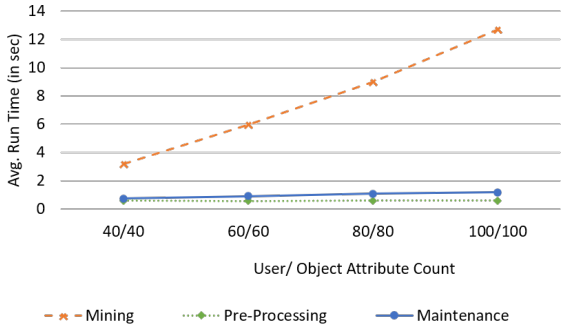
Research reported in this publication was supported by the National Science Foundation awards CNS-1564034, CNS-1624503 and CNS-1747728 and the National Institutes of Health awards R01GM118574 and R35GM134927. The work of Shamik Sural was partially supported by the Fulbright Program. The content is solely the responsibility of the authors and does not necessarily represent the official views of the agencies funding the research.



(a) Varying User-Object Count



(b) Varying Rule Count



(c) Varying User-Object Attribute Count

Figure 6: Running Time for Delete Attribute Value

REFERENCES

[1] Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, and Giuseppe Polese. 2019. Incremental Discovery of Functional Dependencies with a Bit-vector Algorithm.. In *SEBD*.

[2] Shuvra Chakraborty, Ravi Sandhu, and Ram Krishnan. 2019. On the feasibility of attribute-based access control policy mining. In *IEEE IRI*. 245–252.

[3] David W Cheung, Jiawei Han, Vincent T Ng, and CY Wong. 1996. Maintenance of discovered association rules in large databases: An incremental updating technique. In *International conference on data engineering*. IEEE, 106–114.

[4] William Cheung and Osmar R Zaiane. 2003. Incremental mining of frequent patterns without candidate generation or support constraint. In *Seventh International Database Engineering and Applications Symposium*. 111–116.

[5] C. Cotrini, T. Weghorn, and D. Basin. 2018. Mining ABAC Rules from Sparse Logs. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 31–46.

[6] Ghada Gasmi, Lotfi Lakhal, and Yahya Slimani. 2012. An incremental approach for maintaining functional dependencies. *Intelligent Data Analysis* 16, 3 (2012), 365–381.

[7] Mayank Gautam, Sadhana Jha, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. 2017. Poster: Constrained policy mining in attribute based access control. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*. 121–123.

[8] Maanak Gupta, James Benson, Farhan Patwa, and Ravi Sandhu. 2019. Dynamic groups and attribute-based access control for next-generation smart cars. In *ACM CODASPY*. 61–72.

[9] Vincent Hu. 2014. *Attribute based access control (ABAC) definition and considerations.* Technical Report. National Institute of Standards and Technology.

[10] Padmavathi Iyer and Amirreza Masoumzadeh. 2018. Mining positive and negative attribute-based access control policy rules. In *ACM SACMAT*. 161–172.

[11] Padmavathi Iyer and Amirreza Masoumzadeh. 2019. Generalized mining of relationship-based access control policies in evolving systems. In *ACM SACMAT*. 135–140.

[12] Amani Abu Jabal, Elisa Bertino, Jorge Lobo, Mark Law, Alessandra Russo, Seraphin B. Calo, and Dinesh C. Verma. 2020. Polisma - A Framework for Learning Attribute-Based Access Control Policies. In *25th European Symposium on Research in Computer Security, Proceedings, Part I*. Springer, 523–544.

[13] Leila Karimi, Maryam Aldairi, James Joshi, and Mai Abdelhakim. 2020. An Automatic Attribute Based Access Control Policy Extraction from Access Logs.

[14] Leila Karimi and James Joshi. 2018. An unsupervised learning based approach for mining attribute based access control policies. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 1427–1436.

[15] Chun-Wei Lin, Guo-Cheng Lan, and Tzung-Pei Hong. 2012. An incremental mining algorithm for high utility itemsets. *Expert Systems with Applications* 39, 8 (2012), 7173–7180.

[16] Eric Medvet, Alberto Bartoli, Barbara Carminati, and Elena Ferrari. 2015. Evolutionary inference of attribute-based access control policies. In *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 351–365.

[17] DC Mocanu, Fatih Turkmen, and Antonio Liotta. 2015. Towards ABAC Policy Mining from Logs with Deep Learning. *In proc. of the 18th International Multiconference, IS2015* (2015), 124–128.

[18] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin Calo, and Jorge Lobo. 2008. Mining roles with semantic meanings. In *ACM SACMAT*. 21–30.

[19] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. 2019. DynFD: Functional Dependency Discovery in Dynamic Datasets.. In *EDBT*. 253–264.

[20] Tanay Talukdar, Gunjan Batra, Jaideep Vaidya, Vijayalakshmi Atluri, and Shamik Sural. 2017. Efficient bottom-up mining of attribute based access control policies. In *IEEE International Conference on Collaboration and Internet Computing*. 339–348.

[21] Shyue-Liang Wang, Ju-Wen Shen, and Tzung-Pei Hong. 2001. Incremental discovery of functional dependencies using partitions. In *Proceedings Joint 9th IFSA World Congress and 20th NAFIPS International Conference (Cat. No. 01TH8569)*, Vol. 3. IEEE, 1322–1326.

[22] Shyue-Liang Wang, Wen-Chieh Tsou, Jiann-Horng Lin, and Tzung-Pei Hong. 2003. Maintenance of discovered functional dependencies: Incremental deletion. In *Intelligent Systems Design and Applications*. Springer, 579–588.

[23] Zhongyuan Xu and Scott D Stoller. 2015. Mining attribute-based access control policies. *IEEE TDSC* 12, 5 (2015), 533–545.