THESIS

TOWARDS EFFICIENT IMPLEMENTATION OF ATTRIBUTE-BASED ACCESS CONTROL

Submitted by

Vignesh M. Pagadala

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2021

Master's Committee:

Advisor: Indrakshi Ray

Indrajit Ray
Charles Anderson
Leo Vijayasarathy

Copyright by Vignesh M. Pagadala 2021

All Rights Reserved

ABSTRACT

TOWARDS EFFICIENT IMPLEMENTATION OF ATTRIBUTE-BASED ACCESS CONTROL

Attribute-Based Access Control (ABAC) is a methodology which allows or prohibits a subject (user or process) from performing actions on an object (resource), based upon the attributes of the subject and the object. The inherent versatility of ABAC, as opposed to other access control methods such as Role-Based Access Control (RBAC), has ensured the availability of a wide range of use-cases for applying the same, including but not limited to, healthcare, finance, government and military. Of late, more and more organizations are settling for ABAC as their choice of access control scheme. In order to implement ABAC, standards such as the eXtensible Access Control Markup Language (XACML) and Next-Generation Access Control (NGAC) are typically employed. Though these standards allow organizations to implement an access control scheme which is fine-grained, easily manageable and devoid of problems such as role explosions, certain bottlenecks still exist in terms of the time taken to evaluate access requests, and pre-computations being performed to prepare the mechanism for answering queries. These issues become apparent only when the number of entities involved in the organization (subjects and objects) begin to scale. Previous works based on NGAC have been proposed, which manage to ensure efficient evaluation of access requests. However, the procedures outline the need to perform pre-computations, whose time complexity scales rapidly with respect to growing number of entities and policies. We argue that this implementation can be done better, by dexterous use of specific data-structures. Our ABAC implementation (using NGAC) not only answers queries in $O(1)$, but also quickens the pre-computation process to practicable levels, thereby making this more suitable for implementation. We also propose secondary contributions - a mechanism to respond to access requests while a policy update is underway, and procedures to enforce policies from a subset of several policy classes.

ACKNOWLEDGEMENTS

I would like to express gratitude to my advisor Dr. Indrakshi Ray for her encouragement and support during the course of my master's program at Colorado State. She was primarily responsible for introducing me to the world of research, and provided me with tremendous assistance and support.

DEDICATION

To my parents.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
DEDICATION iv
LIST OF TABLES vii
LIST OF FIGURES viii

Chapter 1 Introduction 1
1.1 ABAC and its Significance 1
1.2 Problems and Objectives 2
1.3 Previous Approaches 2
1.4 Problem Statement and Hypothesis 3
1.5 Thesis Outline 4

Chapter 2 Background 5
2.1 NIST Next-Generation Access Control 5
2.2 NGAC Policy Elements 5
2.3 NGAC Relationships 6
2.4 User-Privilege Evaluation 6

Chapter 3 Related Work 8
3.1 Hierarchical Group Attribute-Based Access Control 8
3.2 XACML 9
3.3 Restricted HGABAC 10
3.4 Mell et al. 10
3.5 Basnet et al. 11
3.6 Pagadala et al. 13

Chapter 4 Methodology 16
4.1 Index Construction 16
4.2 Adding a User/Object 16
4.3 Deleting a User/Object 16
4.4 Adding a User/Object Attribute 17
4.5 Deleting a User/Object Attribute 17
4.6 Adding a (u to ua) / (o to oa) ASSIGN 19
4.7 Deleting a (u to ua) / (o to oa) ASSIGN 19
4.8 Adding a (ua_1 to ua_2) / (oa_1 to oa_2) ASSIGN 22
4.9 Deleting a (ua_1 to ua_2) / (oa_1 to oa_2) ASSIGN 22
4.10 Adding a ua to oa ASSOC 22
4.11 Deleting a ua to oa ASSOC 26
4.12 Processing Access Requests 26
4.13 Queries of the form $< u, op, o >$ 26

4.14 Queries of the form $< u, *, * >$ . . . 27
4.15 Queries of the form $< *, *, o >$ . . . 27
4.16 Handling Multiple Policy Classes . . . 27
4.17 Parallelizing Policy Updates and Query Responses . . . 28

Chapter 5 Simulation and Analysis . . . 31
5.1 Experimental Setup . . . 31
5.2 Analysis . . . 31
5.3 Results . . . 32

Chapter 6 Conclusion and Future Work . . . 35

Bibliography . . . 36

LIST OF TABLES

5.1 Time Complexity of Policy Updates 32

LIST OF FIGURES

2.1 NGAC Authorization Graph for an Example Policy Class 7

3.1 Authorization Graph Implementation by [1] 12
3.2 Authorization Graph Implementation by [2] 15

4.1 Index Construction - Storing reachable user and object nodes at every user and object attribute node respectively. 17

5.1 Evaluation Time of $< u, op, o >$ as a Response to Increasing Privileges 33
5.2 Evaluation Time of $< u, *, * >$ as a Response to Increasing Privileges 33
5.3 Evaluation Time of $< *, *, o >$ as a Response to Increasing Privileges 34
5.4 Evaluation Time of $< u, *, * >$ as a Response to Increasing Number of Users 34

Chapter 1

Introduction

1.1 ABAC and its Significance

According to [3], an Attribute-Based Access Control (ABAC) mechanism involves the following entities - a subject (which could be a user or a process), an object (representing the resource to whom access is controlled), an operation (such as read or write) and a policy (governs what operations a subject can perform on an object). Also, a set of subject and object attributes are involved which could represent various properties on subjects and object respectively. For example, subject age could be a subject attribute, and access control policies can be formulated based upon these attributes. Due to flexibility, manageability, and fine grained access control (which it can afford), ABAC is slowly beginning to emerge as a preferable alternative to RBAC. Sectors such as finance, healthcare, insurance and airlines all provide good use-cases for enforcing ABAC [4]. To establish its significance, let us look at two major sectors - finance and healthcare. In addition to the advantages supplied by ABAC mentioned above, banks can ensure Regulatory Compliance by using ABAC, simply by adopting those policy classes which would satisfy said compliance. Furthermore, financial institutions are in need of strong bulwarks against information leaks caused by both external attackers, and insiders. These restrictions are essential not only to maintain the integrity of the institution, but also to ensure the privacy of customer data. In the event of such leaks, not only is the reputation of the company affected, important customer information such as social security numbers, driver's licenses, phone and credit card numbers are lost to the attackers, who may then use these to commit identity theft. The Equifax Breach is a very infamous occurrence of the latter scenario [5]. The Bitglass 2018 Financial Breach Report indicates an incident at SunTrust Banks where an employee stole information associated with around 1.5 million customers. The same report also specifies another incident at the Royal Bank of Canada, leading to the loss of data associated with around 66,000 customers. In the healthcare sector, once again, ABAC would

help in ensuring ease of Regulatory Compliance, considering the fact that the majority of institutions have migrated towards the use of Electronic Health Records (EHR). Furthermore, medical information is considered to be a lot more sensitive than in the previous case. It is not only crucial to ensure confidentiality of data to prevent misuse, but also to ensure that the institution does not violate government-enforced legislation (such as HIPAA in the United States). Traditionally, medical institutions make use of RBAC since well-defined roles are known to exist in this context. However, when such institutions begin to grow, roles may become more difficult to manage. This, in conjunction with the fact that RBAC cannot be as fine-grained as ABAC, may offer motivation for using ABAC.

1.2 Problems and Objectives

In spite of advantages stated previously, one major issue with implementing ABAC stems from the time complexity associated with processing access request queries. Zhang et al. [6] have considered the use of an attribute-based composite EHR model, wherein, EHRs stored in the cloud are subdivided, and each fragment is considered to be an object when enforcing ABAC policies. This ensures fine-grained authorization and access control. However, policy enforcement is burdened by a high complexity costs, due to which they fall back to RBAC. We not only require access control to be manageable and at low-level granularity, we also need a very efficient approach, taking into consideration a sizable system, with a large number of subjects and objects.

1.3 Previous Approaches

In the methodology formulated by Basnet et al. [1], NGAC [7] is employed, for enforcing ABAC. We shall also refer to subjects as ‘users’, for the sake of convenience. According to Mell et al. [8], a set of NGAC policies can be effectively represented as a Directed Acyclic Graph (DAG), and this feature is used extensively to optimize the complexity associated with access-request evaluation. In their work, two types of access requests are considered - (1) a single user requesting to perform an operation on a single object and (2) a single user requesting to view all privileges at

their disposal. Evaluation of the first type of request performs linearly with respect to the number of policies, and for the second type of request, it's log-linear with respect to the number of policies and users, both at worst-case. When an inordinate number of users and operations are present in the DAG, query evaluation would take a long time. In a real-world scenario, this would be disastrous. In the healthcare context, for example, this evaluation time would essentially translate into longer wait-times for emergency-response personnel, nurses, and physicians dealing with an urgent situation. Therefore, an alternative methodology is needed, where such disadvantages are overcome. [2] have built upon this earlier work by [1], wherein, they propose modifying the pre-computation procedure on the DAG, and maintaining dictionaries containing allowed privileges, at every user and object node to optimize access-request evaluation. Using this architecture, they are able to achieve a time complexity of $O(1)$ for answering every type of access request, which is an order of magnitude better than the earlier approach by [1]. [2] also offer optimized procedures for performing these pre-computations, based upon different types of changes which can be carried out on the DAG.

1.4 Problem Statement and Hypothesis

The solution offered by [2] is novel and undoubtedly better than the earlier approach. However, certain loopholes still exist in this proposition. The algorithms provided with respect to different changes in the NGAC DAG for the pre-computations, though optimized, still scale rapidly with respect to increasing number of nodes. This would cause problems when the administrator wishes to enforce certain policy changes, wherein, it might take an inordinate amount of time to enforce the change. For example, let us say that the administrator notices suspicious behavior by a user, or is convinced that a user is operating maliciously, and wishes to immediately restrict said user's privileges. However, if the graph is large enough, by the time the rules are updated, the user might have gotten away with confidential information. Therefore, it is crucial that a mechanism is put in place to ensure faster policy updates. Also, while the graph is undergoing policy changes, it is completely locked out, and is unable to respond to access requests. Any ground covered through

being able to answer queries in $O(1)$ might be offset by the need to wait for the graph to finish updating. This implies a need for a method by which the graph is still able to fully or partially answers some types of queries, while undergoing updates. In this work, we propose modifying these pre-computation procedures in order to ensure much faster enforcement of policy changes, while still retaining aspects of the earlier work which would allow us to answer queries in constant time. We intend to achieve this by storing additional information in those nodes of the DAG which represent user and object attributes. Furthermore, we also provide procedures which identify which parts of the DAG are being changed owing to a policy update operation being carried out. This would allow us to answer certain types of queries even when the graph is undergoing policy changes, and don't have to queue all of them. Additionally, we will add more details with respect to how multiple policy classes can be handled, and provide procedures for the same.

1.5 Thesis Outline

The organization of the subsequent sections of the thesis is described as follows. Chapter 2 contains a detailed description on the NGAC framework. Chapter 3 describes other related work, and the procedure adopted in earlier versions of this work, both of which are required to comprehend the improvements made in this thesis. We also review details involving real-time update of access-control policies and contributions made in [9]. In chapter 4, we propose our optimized methodology, and describe the procedures involved in constructing the NGAC DAG, the novel pre-computation technique we use, and how access requests are evaluated. In chapter 5, a detailed examination and complexity evaluation is done for the algorithms proposed, followed by simulations and results. Chapter 6 succinctly captures the contributions made in this thesis, and the ways by which this work can be improved upon in the future.

Chapter 2

Background

2.1 NIST Next-Generation Access Control

The NGAC specification by NIST [7] allows us to define several types of ABAC policies through explicit specification of policy elements and relationships between them. In this section, we shall look into what these elements and relationships are.

2.2 NGAC Policy Elements

The different elements which makes up the NGAC DAG will be described in this section. Users (u) and Objects (o) are those elements which represents subjects (one who requests access to a resource) and resources, respectively. Both these elements types are associated with attributes. User attributes (ua) include different properties which describe a user, and a user attribute may 'contain' several users. For example, the user attribute *researcher* might be a container with users $u1$, $u2$ and $u3$ belonging to it, in which case, $u1$, $u2$ and $u3$ are considered researchers, and any privilege in possession of *researcher*, is also considered to be in possession by the former. Likewise, object attributes (oa) also act as containers for different objects. However, object attributes differ from user attributes in such a way that every object can also acts as an object attribute. Operations (op) represent actions which can be done on an object by a user. They are further subdivided into resource (ROP) and administrative operations (AOP). Some examples of resource operations are $read$ and $write$. Administrative operations are those which modify the NGAC structure itself, such as elements and relationships. Similar to user and object attributes, policy classes (pc) act as containers for different types of policies. Access rights (ar) determine if a user u has permissions to carry out an operation op over one or more objects. Similar to operations, two types of access rights exist - Resource Access Rights (RAR) and Administrative Access Rights (AAR). Usually,

every single ROP is associated with a RAR, i.e. a one-to-one mapping exists between ROP and RAR.

2.3 NGAC Relationships

The NIST specification defines two relationship types - (1) Assignment (ASSIGN) and (2) Association (ASSOC). An ASSIGN is a binary, directional relationship, which is visually represented with a solid arrow in the NGAC DAG. The node at the head of the relation arrow is essentially an attribute of the node at the tail of the relation. Therefore, an ASSIGN can only go from a user/object node or an attribute node to an attribute node or a policy class. It is irreflexive, meaning that an element cannot have an ASSIGN to itself. Also, every policy class element acts as a sink, with a path of ASSIGNs coming into it from every attribute node. The directed graph formed by the set of ASSIGNs and NGAC elements is also called a policy element diagram or a policy graph.

An ASSOC is a directional, labelled relationship conventionally represented with a dashed arrow, always originating from a user attribute node and ending at an object attribute node, labelled with the allowed access right ar. This relationship indicates that users assigned to the user attribute at the tail of the relation have access right ar over objects assigned to the object attribute at the head of the relation. The directed acyclic graph formed with the different NGAC elements, assignments and associations would effectively represent a finite set of policies which can fundamentally answer questions along the lines of, “Can user u perform operation op on object o?". Since this graph indicates allowed set of authorizations for different users, we shall call the implemented form of the NGAC graph as the ‘authorization graph’. We shall also refer to user and object nodes collectively as ‘source nodes’.

2.4 User-Privilege Evaluation

Let us represent the question “Can user u perform operation op on object o?" as $< u, op, o >$. As stated earlier, the authorization graph can be used to answer if $< u, op, o >$ is allowed or not. The following assumptions are considered. The user u is assigned to the user attribute ua, object

o is assigned to the object attribute oa, and access right ar is mapped to the operation op (that is, a user with ar has the privilege to perform op). The request is considered granted, if an ASSOC exists, with access right ar, between ua and oa. Let us visualize this with an example. Figure 2.1 shows an authorization graph, with different policy elements, assignments and associations. It consists of users u_1, u_2, u_3, u_4, user attributes $ua_1, ua_2, ua_3, ua_4, ua_5$, objects o_1, o_2, o_3, o_4, o_5, object attributes $oa_1, oa_2, oa_3, oa_4, oa_5, oa_6, oa_7, oa_8$, and policy class pc. There are also three associations between ua_1 and oa_3, ua_3 and oa_1, and ua_4 and oa_6. Each association provides access rights mapped to the operations read r and write w. Let us consider the access request $< u_4, r, o_1 >$, which represents user u_4 requesting to perform operation r on object o_1. We know that user u_4 is assigned to user attribute ua_3, and object o_1 is assigned to object attribute oa_1. Since an ASSOC exists between ua_3 and oa_1, and the ASSOC allows operations r and w, this access request is considered granted.

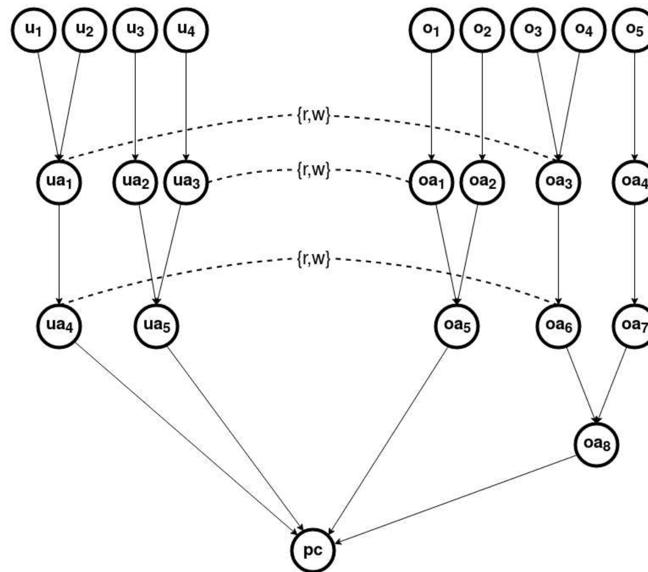

Figure 2.1: NGAC Authorization Graph for an Example Policy Class

Chapter 3

Related Work

3.1 Hierarchical Group Attribute-Based Access Control

Hierarchical Group and Attribute-Based Access Control (HGABAC) is an ABAC model formalized by Servos et al. [10]. In their paper, they propose solutions to deal with a plethora of issues, including the lack of a formalized ABAC model at that time, and the lack of widespread adoption, in-spite of several advantages offered by ABAC over RBAC, as mentioned in section 1.1. This model also offers improved flexibility of the access-control scheme over RBAC. In this model, access requests are granted based upon the attributes of the user, object, and environment. Policies can take the shape of conditional statements involving user and object attributes, for example 'user.height > 165'. Also, the framework presented accommodates a hierarchy of attributes (user and object). This model includes basic elements, attributes, groups, relations and mappings.

Basic Elements

Users, objects, operations, policies, sessions, permissions

Attributes

User attributes, object attributes, environment attributes, connection attributes, administrative attributes, total attributes

Groups

User groups, object groups

Relations

Direct User Attribute Assignment (UAA), Direct Object Attribute Assignment (OAA), User Group Attribute Assignment (UGAA), Object Group Attribute Assignment (OGAA).

Mappings

Direct, consolidate, member, inherited, effective, name, parents, authorized

Groups are used to simplify administrative tasks by allowing assignments to groups of attributes at once making use of the attribute hierarchy. The policy language used to described the access control policies in HGABAC is a boolean rule based policy language, which can evaluate to either TRUE, FALSE or UNDEF, where UNDEF is considered the same as 'FALSE' when evaluating a policy. This model successfully solves the problem of (1) successfully formalizing an ABAC model and (2) ensuring more flexibility than RBAC. However, with respect to finding out the requisite attributes for a user to access a particular resources, this solution scales poorly, and is np-complete [11].

3.2 XACML

eXtensible Access Control Markup Language [12] or XACML is a framework for defining fine-grained ABAC policies, which also specifies a detailed architectural model on the construction and evaluation of ABAC policies. The idea is heavily centered upon dissociating the point of access decision from the actual application itself, ensuring seamless policy updates. The XACML architecture specified the following elements:

- Policy Administration Point (PAP): Access control policies are handled here.
- Policy Decision Point (PDP): Policies are used to evaluate incoming access requests.
- Policy Enforcement Point (PEP): The PEP contacts the PDP upon receiving an access request and and based upon the response, decides to allow or deny a user from accessing a specific resource.
- Policy Information Point (PIP): Attribute store
- Policy Retrieval Point (PRP): Authorization policy store

In XACML, once an access request is received, the PEP forwards this request to the PDP in the form of an XACML request. The request is evaluated at the PDP. The PDP communicates with the PRP (policy store) and the PIP (attribute store). Using this information, the PDP is able to make a decision on whether to allow or deny. This decision is then communicated to the PEP.

Although XACML is widely adopted, and is capable of handling fine grained ABAC policies, XACML lacks scalability as the number of attribute nodes scale in the system.

3.3 Restricted HGABAC

Restricted HGABAC or rHGABAC [13] is a modified HGABAC model which consists of user and object groups, as well as a group hierarchy. This work makes use of NIST's Policy Machine (PM) framework and suggests a suitable implementation methodology. The novelty of this model lies in the fact that policies are formalizes as single-value enumerate policies as opposed to a logical-formula authorization policy. The primary motivation behind this is to accommodate features such as groups, group-attributes and hierarchies with enumerated policy. The basic elements of this model include users, user groups, user attributes, objects, object groups and object attributes, operations, policies and authorization. User and object group hierarchies also exist in this model. Their novel implementation involves the use of the PM architecture.

3.4 Mell et al.

A more solid definition is supplied by Mell et al. [8], for processing access requests. This definition is based on a slightly different version of the authorization graph described before, wherein associations are represented as labeled, directed edges, with the tail of the edge at a ua node, the head at an oa node, and labeled with the allowed operation. It also takes into account scenarios with multiple policy classes, represented with an equivalent number of pc nodes in the authorization graph. Every object, object attribute and user attribute can be assigned to a pc node. An assignment from an object or an object attribute to a pc node implies that, the object or the objects belonging to the object attribute are governed by policies contained in policy class pc. Also, every

u, ua, o and oa node should have a set of assignments leading to at least one pc node. pc nodes also have an out-degree of zero, that is, they are sinks in the graph. If an object or object attribute has assignments leading to multiple policy classes, then each policy class should allow the request, for it to be granted. Even if one of these policy classes has a policy which does not allow the request, it is denied.

According to Mell et al., a user u is allowed an operation op on object o, if

- A set of association edges, with the label op exists, from a set of ua nodes to a set of oa nodes.
- The ua nodes (tail of the edges) and oa nodes (head of the edges) are reachable from u and o respectively.
- Every pc node reachable from o should also be reachable from the oa (head) nodes.

3.5 Basnet et al.

It is clear that ABAC policies can be effectively represented using NGAC graphs, due to which a graph-based platform can be utilized for implementation of the same. This would provide great flexibility with optimizing the speed with which access decisions are made, by making use of traditional graph-traversal techniques. Basnet et al. [1] have employed the Neo4j graph-database platform for this purpose. Their primary motivation for doing this stems from the fact that, Neo4j allows for fast graph traversal, and more importantly, allows for the look-up of a record's location in constant time, using the record's ID. This is possible in Neo4j due to the fact that records (nodes) are stored as fixed-size chunks in memory, and every node's ID points directly to the record. Property graph model is also enforced in Neo4j [14], which allows for the use of attributed, labeled and directed multi-graphs [15]. In the authorization graph which is constructed, every distinct operation (allowed by an ASSOC) are also considered to be nodes, along with u, ua, o, oa and pc nodes. Each of these operation nodes have an out-degree of zero. Attributes are also classified into various types. The authorization graph is 'indexed', meaning, three types of lists are maintained at each node of the type user, object and operation.

At every user node (u node), a list of reachable operations nodes is maintained. The same is done for every object node (o node). On the other hand, every operation node (op node) contains two distinct lists. One is for every u node from where the given op node can be reached. The other is the same, but for every o node that can reach the op node. To perform this procedure, Basnet et al. provide us with algorithms where, a depth-first search is initially carried out from every u and o node, and all visited nodes are stored in $u.nodes$ and $o.nodes$ list respectively. An intersection is then performed between these two lists to determine the op nodes, and map the same with the users and objects which can reach them. It also has to be noted that, each list is stored in sorted order of node ID (Neo4j assigns every node with an ID). Figure 3.1 shows an authorization graph after the indexing has been performed, where user, object and operation nodes are all associated with lists as described above. Now, using this indexed graph, access decisions can be made. As stated earlier, two types of access requests are being considered, one of the type $< u, op, o >$ and $< u, *, * >$. Let us take up the procedure used to evaluate access requests of the type $< u, op, o >$. The fundamental requirement here is to perform an intersection between the op lists maintained at the user u and object o. If an intersection exists, and the op node in the intersection is the same as the one in the request, access is granted. If this condition is not met, access is denied. This algorithm makes use of a modified version of the common predecessor algorithm [16], to traverse each list only once, and reduce the complexity of the same. The time complexity of this algorithm is $O(|op|)$, where $|op|$ represents the number of operation nodes.

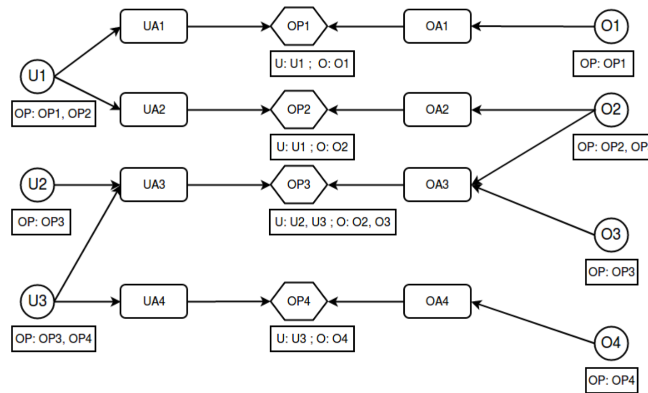

Figure 3.1: Authorization Graph Implementation by [1]

Next comes the problem of evaluating access requests of the type $< u, *, * >$. This is done by iterating through the op node list maintained at u, and returning the set of operations, and also the object nodes associated with the same. The time complexity of this algorithm is $O(|op|log|u|)$, where $|op|$ is the number of operation nodes and $|u|$ is the number of user nodes.

3.6 Pagadala et al.

In order to improve the time complexity associated with evaluating access requests, [2] have come up with a novel solution. They have proposed a pre-computation procedure wherein, at the end of indexing, instead of maintaining lists at user, object and operation nodes, information is only held in the source nodes. At every source node, a dictionary is maintained. In user nodes, the dictionary contains every allowed access request in the form of $< op, o >$, mapped with a self-referencing key. Therefore, if the dictionary at user u_1 contains the element $< op_1, o_1 >$, then it would imply that u_1 has access to perform operation op_1 on object o_1. [2] have also come up with a solution for addressing another query type $< *, *, o >$, which translates into "Which users have what access rights over object o?". In order to answer this, a dictionary is also maintained at every object node, containing sets of $< u, op >$, representing a user u, with an access right allowing operation op over the object. In their algorithms, they propose the following approach to deal with each type of query. For a query of the type $< u, op, o >$, two major look-ups have to be performed. Firstly, the user u has to be accessed. In Neo4j, any node in the graph can be obtained in $O(1)$, with the node ID. Secondly, the access request $< op, o >$ has to be searched in the dictionary maintained at u. Since this information is stored as a hash-map with a self referencing key, this can also be verified in $O(1)$, and hence, the entire query can be answered in constant-time. For queries of the type $< u, *, * >$, and $< *, *, o >$, we simply have to look-up the user and object nodes respectively, and return the hash-maps stored at the given node, an operation which can once again be achieved in $O(1)$.

In order to populate these hash-maps at every source node, the following approach is used. A depth-first search (DFS) traversal is initially performed from every source node, storing every

visited node at that source node. After this, every combination of user and object nodes are intersected, and in turn, the visited node list for each source-node combination is also intersected. This can be used in order to store every $< op, o >$ or $< u, op >$ combination which we come across, at every user and object node respectively. On top of this, [2] propose different, optimized algorithms for the following types of changes in the authorization graph - adding or removing users, objects, user attributes, object attributes, user - user attribute ASSIGNs, object - object attribute ASSIGNs, user attribute - user attribute ASSIGNs, object attribute - object attribute ASSIGNs and user attribute - object attribute ASSOCs. Though this approach answers all query types in constant time, the worst-case time-complexity of the procedures used for enforcing these different policy changes is still too high. The authors in [2] acknowledge this issue, and propose queuing different updates, and enforcing them at fixed time-intervals (for example, at the end of the day). However, in certain production environments, this principle will not be suitable. Let us consider the case of a popular net-banking application. In such a scenario, bank accounts and user profiles might get created and deleted once in every few seconds. Waiting till the end of the day to enforce these changes might not be a satisfactory prospect for a customer, who might want their account to be created instantly without delay. Therefore, we need a faster mechanism in order to deal with policy updates, and still be able to answer access control queries in constant time. Also, in order to prevent erroneous evaluation of requests, the authors propose shutting down the authorization graph when policy updates are taking place. This once again causes similar issues as discussed above.

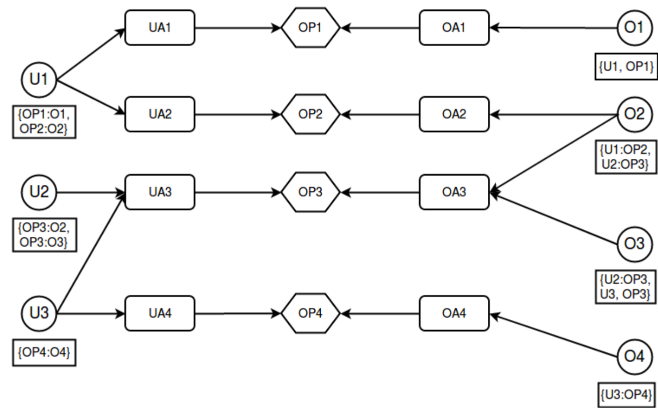

Figure 3.2: Authorization Graph Implementation by [2]

Chapter 4

Methodology

4.1 Index Construction

In our novel methodology, we propose storing additional information at every user attribute and object attribute node, in order to speed up policy updates even further. As shown in Figure 5.1, at every user attribute node, we shall store every user node which has a path of ASSIGNs to that user attribute node, in a 'user list'. Likewise, we do the same with object and object attribute nodes, storing said information in an 'object list'. The advantages of doing this additional step is described below, along with the algorithms. Also, we shall refer to a node which is an immediate predecessor to an operation nodes, as a 'boundary node'.

As stated before, the following types of changes can be effected in the authorization graph - add or remove users, objects, user attributes, object attributes, u - ua ASSIGNs, ua - ua ASSIGNS, o - oa ASSIGNs, oa - oa ASSIGNs and ua - oa ASSOCs. We shall describe algorithms to handle each type of these updates, below. It is to be noted that the 'label' associated with a node is used to store information on what type of node it is, and any data structures mapped to the node.

4.2 Adding a User/Object

A user/object node is added to the authorization graph with the label 'User'/'Object'. No additional indexing operations are necessary.

4.3 Deleting a User/Object

When a user or object gets deleted, we perform a DFS from the user/object, updating the user/object lists stored at the attribute nodes by deleting the entry for u/o. We also store every operation node encountered in $opList$. After this, we iterate through every user/object node, and check if an entry associated with any one of the operation nodes are present in their dictionar-

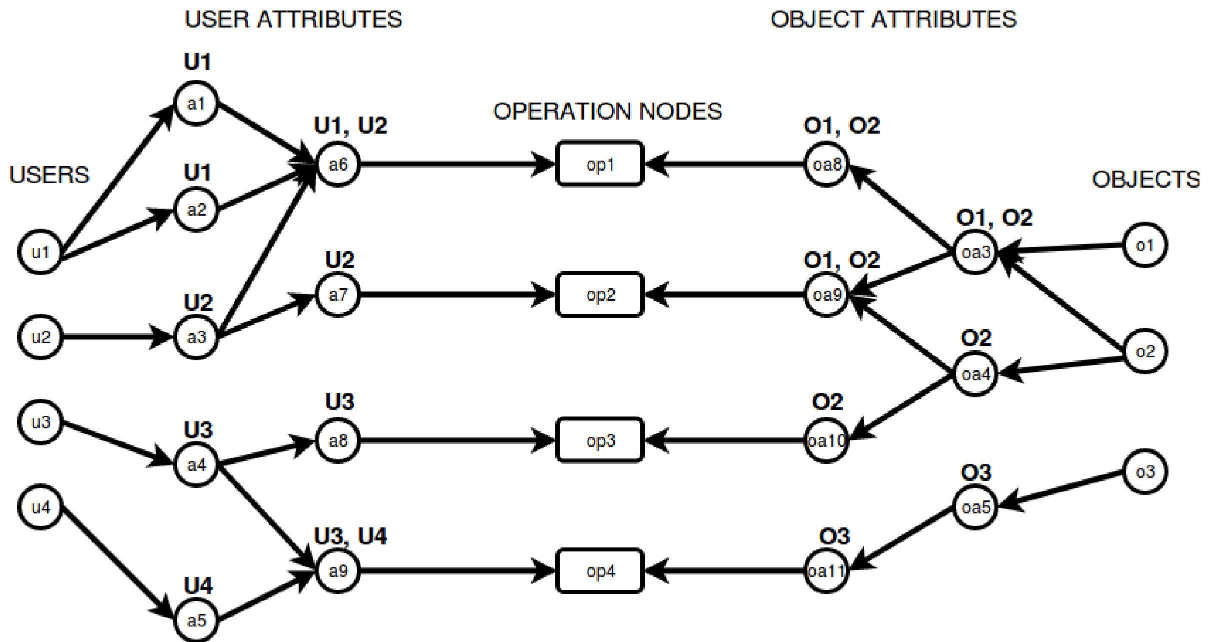

Figure 4.1: Index Construction - Storing reachable user and object nodes at every user and object attribute node respectively.

ies. This can be performed in constant time. If an entry is present, then it is removed from the dictionary.

4.4 Adding a User/Object Attribute

A ua/oa node can also be added with no additional required operations. As stated previously, a ua/oa node is created with two labels, one indicating that it is a node of they type 'User/Object Attribute', and the other which stores the user/object list.

4.5 Deleting a User/Object Attribute

If a user/object attribute needs to be deleted, we initially run a DFS from the attribute node being deleted, and update the successors attribute nodes by removing the users/objects stored in the respective node lists. Also, when an operation node is reached, we also update the dictionaries maintained on the objects/users, by performing an intersection between the node list maintained in the user/object attribute node being deleted, and the object/user attribute node which is the other

Algorithm 1 Add User/Object

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User/Object to be added = u/o]
3: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $u \in users^{'}$/$o \in objects^{'}$]
4: $u.permissions$/$o.permissions$ = { }

Algorithm 2 Delete User/Object

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User/Object to be deleted = u/o]
3: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $(u \notin users^{'})$/$(o \notin objects^{'})$]
4: $opList \leftarrow \emptyset$
5: **while** DFS traversal from u/o **do**
6:     $node \leftarrow$ current node
7:     **if** 'operation' $\in node.Label$ **then**
8:         $opList \leftarrow opList \cup node$
9:     **else**
10:         $(node.uList)$/$(node.oList) \leftarrow (node.uList - u)$/$(node.oList - o)$
11:     **end if**
12: **end while**
13: **for all** (o)/$(u) \in (objects^{'})$/$(users^{'})$ **do**
14:     **for all** $op \in opList$ **do**
15:         **if** (u, op)/$(op, o) \in (o.permissions)$/$(u.permissions)$ **then**
16:             $(o.permissions)$/$(u.permissions) \leftarrow (o.permissions - (u, op))$/$(u.permissions - (op, o))$
17:         **end if**
18:     **end for**
19: **end for**
20: **return** Updated Labelled Graph

Algorithm 3 Add User/Object Attribute

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]

2: [Input: User/Object attribute to be added = =ua/oa]

3: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $(ua \in user_attributes^{'})$ / $(oa \in object_attributes^{'})$]

4: $(ua.uList)$ / $(oa.oList)$ = { }

parent of the operation node, and deleting the entries. Now, it is possible that other paths might exist, which may give the users/objects present in the user/object attribute node's list, the access rights which were removed through the above process. Therefore, we run multiple DFSs from each user/object node in the attribute node's list, and populate the user and object dictionaries, and attribute node lists maintained in the user/object attribute nodes appropriately.

4.6 Adding a (u to ua) / (o to oa) ASSIGN

When adding a (u to ua) / (o to oa) ASSIGN, we need to perform the following. Begin a DFS from node ua / oa. Mark every attribute node visited by appending u / o to the nodes user/object list. If a boundary node is reached, access the oa / ua node on the other side of the op node, and use its object/user list to populate the dictionary at u / o. Also loop through the object/user list, and add u/o to each object's/users's dictionary.

4.7 Deleting a (u to ua) / (o to oa) ASSIGN

In order to delete a $(u - ua)/(o - oa)$ ASSIGN, we perform DFS traversal from ua / oa, and update every attribute node visited by removing the entry for u/o present the node. Once a boundary node is reached, we access the object/user attribute nodes on the other side of the operation node, and update their respective dictionaries (by removing entries). After this, we run

Algorithm 4 Delete User/Object Attribute

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User/Object attribute to be deleted =ua/oa]
3: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $(ua \notin user_attributes^{'})$ / $(oa \notin object_attributes^{'})$]
4: $(userList)/(objectList) \leftarrow (ua.uList)/(oa.oList)$
5: **while** DFS traversal from ua/oa **do**
6: $(userNode)/(objectNode) \leftarrow$ current node
7: **if** $(userNode)/(objectNode)$ is a boundary node **then**
8: $opNode \leftarrow$ child of $(userNode)/(objectNode)$
9: $(objectNode)/(userNode) \leftarrow (oa)/(ua)$ node which is the sibling of $(userNode)/(objectNode)$
10: $(oList)/(uList) \leftarrow (objectNode.oList)/(userNode.uList)$
11: **for all** $(o \in oList)/(u \in uList$ **do**
12: **for all** $(u \in userList)/(o \in objectList)$ **do**
13: **if** $((u, op) \in o.permissions)/((op, o) \in u.permissions)$ **then**
14: $(o.permissions)/(u.permissions) \leftarrow (o.permissions)/(u.permissions) - ((u, op)/(op, o))$
15: **end if**
16: **end for**
17: **end for**
18: **else**
19: $(userNode)/(objectNode) \leftarrow (userNode.uList)/(objectNode.oList) - (userList)/(objectList)$
20: **end if**
21: **end while**
22: Delete ua/oa node.

Algorithm 5 **Add (u to ua) / (o to oa) ASSIGN**

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User/Object =u/o]
3: [Input: User/Object Attribute =ua/oa]
4: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $(u - ua$ ASSIGN $\in E)$ / $(o - oa$ ASSIGN $\in E)$]
5: **while** DFS traversal from ua / oa **do**
6: $(uatt)/(oatt) \leftarrow$ current node
7: $(uatt.uList)/(oatt.oList) \leftarrow (uatt.uList)/(oatt.oList) \cup u/o$
8: **if** $(uatt)/(oatt)$ is a boundary node **then**
9: $(oatt)/(uatt) \leftarrow (oa)/(ua)$ node which is the sibling of uatt
10: **for all** $(o \in oatt.olist)/(u \in uatt.uList)$ **do**
11: $(u.permissions \leftarrow u.permissions \cup (op, o))/(o.permissions \leftarrow o.permissions \cup (u, op))$
12: $(o.permissions \leftarrow o.permissions \cup (u, op))/(u.permissions \leftarrow u.permissions \cup (op, o))$
13: **end for**
14: **end if**
15: **end while**
16: **return** Updated Labelled Graph

another DFS from u / o, updating the attribute-node lists, and the object/user dictionaries once we reach boundary nodes.

4.8 Adding a (ua_1 to ua_2) / (oa_1 to oa_2) ASSIGN

When adding a (ua_1 to ua_2) / (oa_1 to oa_2) ASSIGN, we need to perform the following. Begin a DFS from node ua_2 / oa_2. Mark every user attribute node visited by appending ua_1's / oa_1's user/object list to the current nodes user/object list, eliminating redundancies. If a boundary node is reached, access oa/ua on the other side of the operation node and update users/objects in ua_1's / oa_1's uList/oList with oa's/ua's object/user list, update objects/users in oa's / ua's oList/uList with users/objects in ua_1's / oa_1's uList/oList.

4.9 Deleting a (ua_1 to ua_2) / (oa_1 to oa_2) ASSIGN

In order to delete a $(ua_1 - ua_2)/(oa_1 - oa_2)$ ASSIGN, we follow a procedure similar to algorithm 6, where a DFS is used (from ua_2 / oa_2) to remove the uList/oList entries from the attribute nodes, and removing entries from the user and object hashmaps, once we reach boundary nodes during the DFS. After this, to account for alternative paths which might exist from ua_1/oa_1 we perform another DFS from ua_1/oa_1 and add the user/object nodes in the respective lists, in the list of every visited attribute node. Once boundary is reached, we update the hash-maps at both the users and objects, by adding the permissions deleted earlier.

4.10 Adding a ua to oa ASSOC

Things become very simple when adding a ua to oa ASSOC. Since we already have user and object node information in the ua and oa nodes, there is no need to perform any DFS at all.

Algorithm 6 Delete (u to ua) / (o to oa) ASSIGN

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User = u]
3: [Input: User Attribute = ua]
4: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $u - ua$ ASSIGN $\notin$ E]
5: $userList \leftarrow ua.uList$
6: **while** DFS traversal from ua **do**
7: $userNode \leftarrow$ current node
8: **if** $userNode$ is a boundary node **then**
9: $opNode \leftarrow$ child of $userNode$
10: $objectNode \leftarrow oa$ node which is the sibling of $userNode$
11: $oList \leftarrow objectNode.oList$
12: **for all** $o \in oList$ **do**
13: **for all** $u \in userList$ **do**
14: **if** $(u, op) \in o.permissions$ **then**
15: $o.permissions \leftarrow o.permissions - (u, op)$
16: **end if**
17: **end for**
18: **end for**
19: **else**
20: $userNode \leftarrow userNode.uList - userList$
21: **end if**
22: **end while**
23: Delete $u - ua$ ASSIGN.
24: **for all** $u \in userList$ **do**
25: **while** DFS traversal from u **do**
26: $uatt \leftarrow$ current node
27: **if** $uatt$ is a boundary node **then**
28: $opNode \leftarrow$ child of $uatt$
29: $oatt \leftarrow oa$ node which is the sibling of $uatt$
30: $objectList \leftarrow oatt.oList$
31: **for all** $o \in objectList$ **do**
32: $o.permissions \leftarrow o.permissions \cup (u, opNode)$
33: $u.permissions \leftarrow u.permissions \cup (opNode, o)$
34: **end for**
35: **else**
36: $uatt.uList \leftarrow uatt.uList \cup u$
37: **end if**
38: **end while**
39: **end for**

Algorithm 7 Add (ua to ua) / (oa to oa) ASSIGN

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User/Object Attribute 1 = ua_1 / oa_1]
3: [Input: User/Object Attribute 2 = ua_2 / oa_2]
4: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $(ua_1 - ua_2)/(oa_1 - oa_2)$ ASSIGN $\in$ E]
5: **while** $(uatt)/(oatt)$ in DFS traversal from $(ua_2)/(oa_2)$ **do**
6: $(uatt.uList)/(oatt.oList) \leftarrow (uatt.uList \cup ua_1.uList)/(oatt.oList \cup oa_1.oList)$
7: **if** $(uatt)/(oatt)$ is a boundary node **then**
8: $(oatt)/(uatt) \leftarrow$ attribute node on the other side of the operation node.
9: **for all** $(o \in oatt.oList)/(u \in uatt.uList)$ **do**
10: **for all** $(u \in ua_1.uList)/(o \in oa_1.oList)$ **do**
11: $(u.permissions \leftarrow u.permissions \cup (op, o))/(o.permissions \leftarrow o.permissions \cup (u, op))$
12: $(o.permissions \leftarrow o.permissions \cup (u, op))/(u.permissions \leftarrow u.permissions \cup (op, o))$
13: **end for**
14: **end for**
15: **end if**
16: **end while**
17: **return** Updated Labelled Graph

Algorithm 8 Delete (ua to ua) / (oa to oa) ASSIGN

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User/Object Attribute 1 = ua_1/oa_1]
3: [Input: User/Object Attribute 2 = ua_2/oa_2]
4: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $(ua_1 - ua_2)/(oa_1 - oa_2)$ ASSIGN $\notin E$]
5: $(userList_1)/(objectList_1) \leftarrow (ua_1.uList)/(oa_1.oList)$
6: $(userList_2)/(objectList_2) \leftarrow (ua_2.uList)/(oa_2.oList)$
7: **while** DFS traversal from $(ua_2)/(oa_2)$ **do** $uatt/oatt \leftarrow$ current node
8: **if** $uatt/oatt$ is a boundary node **then**
9: $opNode \leftarrow$ child of $uatt/oatt$
10: $(oatt)/(uatt) \leftarrow$ parent of $opNode$ which is not $(uatt)/(oatt)$
11: $(objectList)/(userList) \leftarrow (oatt.oList)/(uatt.uList)$
12: **for all** $(o \in objectList)/(u \in userList)$ **do**
13: **for all** $(u \in userList_1)/(o \in objectList_1)$ **do**
14: $(o.permissions)/(u.permission) \leftarrow (o.permissions)/(u.permission) - (u, opNode)/(opNode, o)$
15: $(u.permissions)/(o.permissions) \leftarrow (u.permissions)/(o.permission) - (opNode, o)/(u, opNode)$
16: **end for**
17: **end for**
18: **else**$(uatt.uList)/(oatt.oList) \leftarrow (uatt.uList)/(oatt.oList) - (userList_1)/(objectList_1)$
19: **end if**
20: **end while**
21: **while** DFS traversal from $(ua_1)/(oa_1)$ **do** $uatt/oatt \leftarrow$ current node
22: **if** $uatt/oatt$ is a boundary node **then**
23: $opNode \leftarrow$ child of $uatt/oatt$
24: $oatt/uatt \leftarrow$ parent of $opNode$ which is not $uatt/oatt$
25: $(objectList)/(userList) \leftarrow (oatt.oList)/(uatt.uList)$
26: **for all** $(o \in objectList)/(u \in userList)$ **do**
27: **for all** $(u \in userList_1)/(o \in objectList_1$ **do**
28: $(o.permissions \leftarrow o.permissions \cup (u, opNode))/(u.permissions \leftarrow u.permissions \cup (opNode, o))$
29: $(u.permissions \leftarrow u.permissions \cup (opNode, o))/(o.permissions \leftarrow o.permissions \cup (u, opNode))$
30: **end for**
31: **end for**
32: **else**$(uatt.uList)/(oatt.oList) \leftarrow (uatt.uList)/(oatt.oList) \cup (userList_1)/(objectList_1)$
33: **end if**
34: **end while**
35: **return** Updated Labelled Graph

Algorithm 9 Add ua to oa ASSOC

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User Attribute = ua]
3: [Input: Object Attribute = oa]
4: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $ua - oa$ ASSOC $\in E$]
5: **for all** $u \in ua.uList$ **do**
6:     **for all** $o \in oa.oList$ **do**
7:         $u.permissions \leftarrow u.permissions \cup (op, o)$
8:         $o.permissions \leftarrow o.permissions \cup (u, op)$
9:     **end for**
10: **end for**
11: **return** Updated Labelled Graph

4.11 Deleting a ua to oa ASSOC

Similar to the process when adding an ASSOC, there is no necessity to perform any DFS when deleting an ASSOC either. We simply access the $uList$ and $oList$ maintained at ua and oa respectively, perform an intersection, and update the hashmaps.

4.12 Processing Access Requests

In order to evaluate queries, we can use the same approach as in [2].

4.13 Queries of the form $< u, op, o >$

In order to answer these queries, we initially look up the user node with the ID u. After this we access the $permissions$ dictionary stored at user u and look-up the sub-string $< op, o >$. If this sub-string is found in the dictionary, then access is granted, otherwise access is denied.

Algorithm 10 Delete ua to oa ASSOC

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: User Attribute = ua]
3: [Input: Object Attribute = oa]
4: [Output: Updated Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$ and $ua - oa$ ASSOC $\notin E$]
5: $op \leftarrow$ operation node
6: **for all** $u \in ua.uList$ **do**
7:     **for all** $o \in oa.oList$ **do**
8:         $u.permissions \leftarrow u.permissions - (op, o)$
9:         $o.permissions \leftarrow o.permissions - (u, op)$
10:     **end for**
11: **end for**
12: **return** Updated Labelled Graph

4.14 Queries of the form $< u, *, * >$

For answering these queries, as stated before, we initially look up the user u in the Neo4j graph, and simply return the $permissions$ dictionary stored at u.

4.15 Queries of the form $< *, *, o >$

Similar to how the previous type of query is answered, we once again look up the object o using the Neo4j node ID, and return the hash-map stored in o as the output.

4.16 Handling Multiple Policy Classes

A policy class essentially represents a set of policies. It is useful to containerize sets of policies, as the administrator can quickly adapt the institutions policies to different scenarios, by enforcing pre-compiled policy classes. For example, we may need one set of policies during the regular operation of an institution, and another set of policies during an emergency situation. In other situations, a administrator might decide to enforce rules from two or more policy classes. In order

to deal with this, we propose maintaining multiple authorization graphs, one for every policy class. Now let us assume that the administrator wants policy classes pc_1 and pc_2 to be implemented. In this situation, we quite obviously cannot have a user accessing a resource through a policy allowed by pc_1 but not by pc_2. Therefore, we need an intersection of allowed policies in between both policy classes. To implement this in the context of our work, we need to intersect the hash-maps maintained at every source node in policy class pc_1, with their counterparts in policy class pc_2, to arrive at the correct access rights for each user. This procedure is outlined in algorithm 11. We get the source-node dictionaries of policy class pc_1 in lines 3-4, and perform the cross-policy-class intersection in lines 7 - 8.

Algorithm 11 Enforcing Two or More Policy Classes

1: [Input: Set of policy classes to be enforced $PC/gets pc_1, pc_2, pc_3, ...$]
2: [Output: Graph with combined labels = $(users' \cup objects')$ where $\forall u \in users', u.permissions \neq \emptyset$ and $\forall o \in objects', o.permissions \neq \emptyset$]
3: $users \leftarrow pc_1.users$
4: $objects \leftarrow pc_1.objects$
5: **for all** $u \in users \wedge o \in objects$ **do**
6:     **for all** $pc \in PC$ **do**
7:         $u.permissions \leftarrow u.permissions \cap pc.u.permissions$
8:         $o.permissions \leftarrow o.permissions \cap pc.o.permissions$
9:     **end for**
10: **end for**
11: **return** Intersected Labelled Graph

4.17 Parallelizing Policy Updates and Query Responses

In the previous work [2], every time a policy update is taking place, the authorization graph is taken offline, and does not respond to any access requests till the update is complete. This is done in order to ensure that access requests are not being evaluated erroneously. However, it is possible to respond to a subset of all possible access requests correctly, even while the graph is being updated. In this section we outline a suitable methodology to determine if an access request can be evaluated while a policy update is under progress, given an access request and the type of policy

update being carried out. Algorithm 12 outlines this procedure. Two inputs are considered - the access request $< u^{'}, op^{'}, o^{'} >$ and the policy update (U) taking place in the same instance of time the access request is received. Also, U $\in u, o, ua, oa, u - ua, o - oa, ua - ua, oa - oa, ua - oa$. We initially focus on identifying the subset of user and object nodes which are 'affected' by the policy update operation. For example, if a policy update of the type ua or oa is taking place (line 10), we search the uList/oList maintained at ua/oa to check if $u^{'}/o^{'}$ is present (line 12). If $u^{'}/o^{'}$ is present, then the access request and policy update operation may conflict with each other. Hence, algorithm 12 returns NO, the access request is queued, and we respond after the update operation is complete. If $u^{'}/o^{'}$ is not present, then we know for sure that no conflict would happen. Hence, the algorithm returns YES, after which we can respond to the access request using the procedure outline in section 4.13. If the policy update type U $\in$ u, o, $u - ua$, $o - oa$, then the algorithm resolves the solution in $O(1)$. If U $\in ua, oa, ua_1 - ua_2, oa_1 - oa_2, ua - oa$, then the algorithm runs in $O(log(n))$, where n is the number of elements in the attribute list. This is because, we have to check the lists maintained at the attribute node for the presence of a user or object node. Since we store these nodes in ascending order of node ID, a binary search can be used to determine presence in $O(log(n))$.

Algorithm 12 Handling Access Requests during Policy Updates

1: [Input: Labelled Graph = $(users^{'} \cup user_attributes^{'} \cup objects^{'} \cup object_attributes^{'} \cup operations, E)$ where $\forall u \in users^{'}, u.permissions \neq \emptyset$ and $\forall o \in objects^{'}, o.permissions \neq \emptyset$ and $\forall ua \in user_attributes^{'}, ua.uList \neq \emptyset$ and $\forall oa \in object_attributes^{'}, o.oList \neq \emptyset$]
2: [Input: policy update U $\in \{u, o, ua, oa, u-ua, o-oa, ua_1-ua_2, oa_1-oa_2, ua-oa\}$]
3: [Input: access request = $< u^{'}, op^{'}, o^{'} >$]
4: [Output: YES or NO]
5: **if** U == $(u \vee o \vee (u-ua) \vee (o-oa))$ **then**
6:     **if** $(u == u^{'}) \vee (o == o^{'})$ **then**
7:         **return** NO
8:     **end if**
9: **end if**
10: **if** U == $(ua \vee oa)$ **then**
11:     $nodeList \leftarrow ua.uList/oa.oList$
12:     **if** $(u^{'} \vee o^{'}) \in nodeList$ **then**
13:         **return** NO
14:     **end if**
15: **end if**
16: **if** U == $((ua_1-ua_2) \vee (oa_1-oa_2))$ **then**
17:     $nodeList \leftarrow ua_1.uList/oa_1.oList$
18:     **if** $u^{'} \vee o^{'} \in nodeList$ **then**
19:         **return** NO
20:     **end if**
21: **end if**
22: **if** U == $ua-oa$ **then**
23:     $uList \leftarrow ua.uList$
24:     $oList \leftarrow oa.oList$
25:     **if** $u^{'} \in uList \vee o^{'} \in oList$ **then**
26:         **return** NO
27:     **end if**
28: **end if**
29: **return** YES

Chapter 5

Simulation and Analysis

5.1 Experimental Setup

Our first set of experiments are intended to examine the complexity of access-request evaluation, by looking at the evaluation time and how it varies as a response to increasing number of entities in the authorization graph. We initially start off with one user and object node, and keep incrementally adding ua and oa nodes, creating associations between them (i.e. operation nodes). In order to simulate a suitable worst-case, every ua and oa nodes contains associations, and therefore op nodes, between them. This implies that as the number of ua and oa nodes grow, so do the number of op nodes in the graph. We shall record the access-request evaluation time for an increasing number of operation nodes and user nodes. All the access request types ($< u, op, o >, < u, *, * >, < *, *, o >$) will be tested in these experiments.

5.2 Analysis

In table 5.1, we compare the algorithmic time complexity of our approach with previous approaches, for the worst-case scenario. One of the most common policy update type an administrator might initiate would be of the type ADD or DELETE u - o ASSOC. Our improvements become clear when we observe that, adding or deleting associations can be done in constant time, in our approach, whereas the approach by [2] takes up quadratic complexity. This is because, since we store every user/object node which can reach a given attribute node, we completely eliminate the need to perform a DFS, therefore achieving higher speed. Likewise, adding an ASSIGN from u to ua can also be resolved in $O(1)$ in our approach, as opposed to linear time in the earlier approach. We also show significant improvement w.r.t. deleting user/objects, which can be resolved in linear time. This is primarily because, we only need to perform exactly one DFS from the source node

Table 5.1: Time Complexity of Policy Updates

Component	ADD (previous)	DELETE (previous)	ADD (current)	DELETE (current)
u / o	$O(1)$	$O(\|n\|^2)$	$O(1)$	$O(\|n\|)$
ua / oa	$O(1)$	$O(\|n\|)$	$O(1)$	$O(\|n\|)$
u-ua / o-oa	$O(\|n\|)$	$O(\|n\|)$	$O(1)$	$O(\|n\|)$
ua-ua / oa-oa	$O(\|n\|)$	$O(\|n\|)$	$O(n)$	$O(\|n\|)$
ua-oa	$O(\|n\|^2)$	$O(\|n\|^2)$	$O(1)$	$O(1)$

being deleted, as opposed to performing a DFS from every object/user node along with the DFS from the node being deleted.

5.3 Results

Figure 5.1 shows the results of the experiments with the request type $< u, op, o >$. It can clearly be observed that, with increasing number of operation nodes (privileges), increased up to 16,000 nodes, the evaluation time of access requests do not grow linearly with the nodes, at worst case. There does not appear to be a trend indicating $O(logn)$ either, but rather, proves our hypothesis of constant-time evaluation of access requests. Figures 5.3, 5.4 represent the outcome of the experiments evaluating the query type $< u, *, * >$ with increasing op nodes, and increasing users respectively. Once again, these response values undoubtedly indicate that the time does not linearly, or log-linearly vary based upon the number of entities in the graph, but is $O(1)$ with respect to the number of entities in the graph. Finally, figure 5.2 shows the results for increasing the number of privileges while assessing the evaluation time for the query type $< *, *, o >$. This result also lines up with our original premise, and shows that the query evaluation time does not show an increasing trend. Therefore, we can conclude based on these results that all three access request types can be evaluated in constant time, using the proposed methodology.

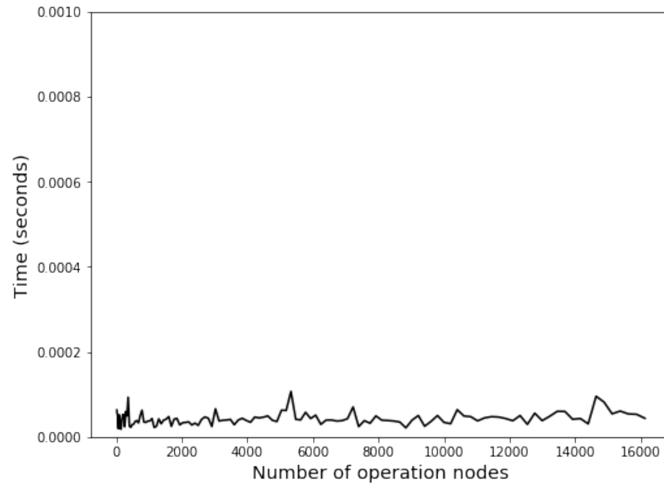

Figure 5.1: Evaluation Time of $< u, op, o >$ as a Response to Increasing Privileges

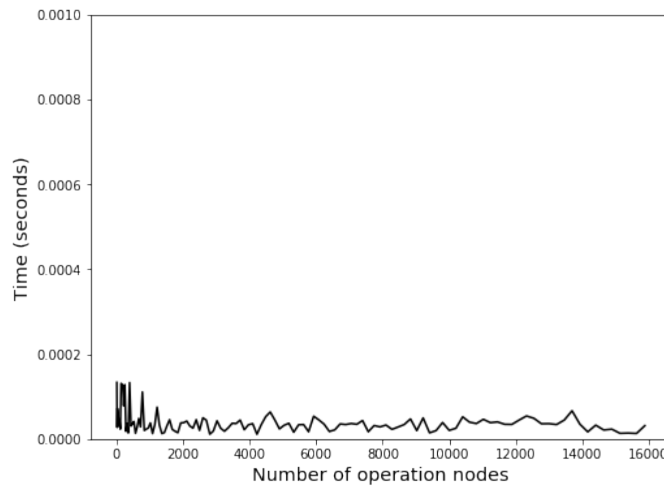

Figure 5.2: Evaluation Time of $< u, *, * >$ as a Response to Increasing Privileges

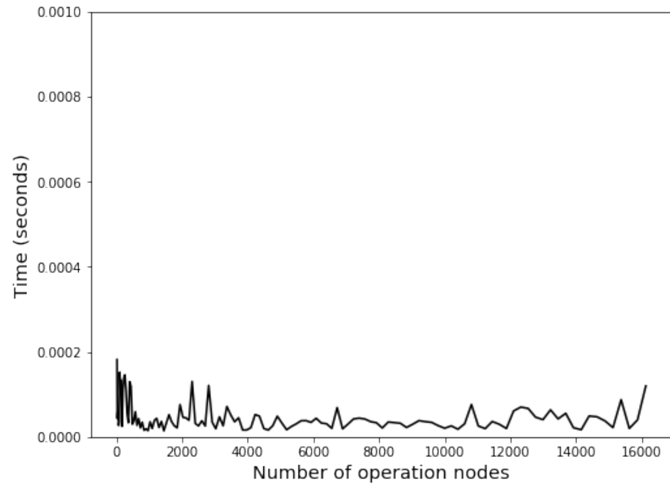

Figure 5.3: Evaluation Time of $< *, *, o >$ as a Response to Increasing Privileges

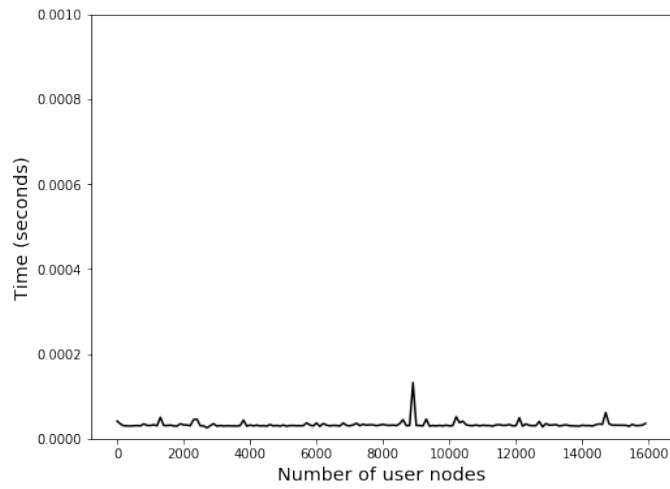

Figure 5.4: Evaluation Time of $< u, *, * >$ as a Response to Increasing Number of Users

Chapter 6

Conclusion and Future Work

In this work, we proposed a method for implementing ABAC in the production environment by expediting the process of, not only evaluating access control queries, but also ensuring that they can be updated far more quickly in real-time. We provided various algorithms for indexing the NGAC graph implementation, and showed that, through the use of lists maintained at every user and object attribute node in the authorization graph, we can quicken the process of updating the graph. Our contributions also ensure that certain types of queries can still be processed even when the graph is undergoing policy updates, by determining whether an update is causing policy relaxation or policy restriction to happen. On top of this, we have laid out a method by which an administrator can implement rules from two or more policy classes. This work can be extended in two ways. Firstly, by integrating features to ensure the confidentiality of the policies themselves, possibly through the use of encryption at the attribute level. Secondly, by looking into suitable authentication and verification schemes to establish user identity, and bootstrap this into our proposed ABAC implementation.

Bibliography

[1] Rejina Basnet, Subhojeet Mukherjee, Vignesh M Pagadala, and Indrakshi Ray. An efficient implementation of next generation access control for the mobile health cloud. In *Fog and Mobile Edge Computing (FMEC), 2018 Third International Conference on*, pages 131–138. IEEE, 2018.

[2] Vignesh Pagadala and Indrakshi Ray. Achieving mobile-health privacy using attribute-based access control. In *Springer LNCS, 2018 The 11th International Symposium on Foundations Practice of Security*. Springer-Verlag, 2018.

[3] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, 800(162), 2013.

[4] Ann Cavoukian, Michelle Chibba, Graham Williamson, and Andrew Ferguson. The importance of abac: Attribute-based access control to big data: Privacy and context. *Privacy and Big Data Institute, Ryerson University, Toronto, Canada*, 2015.

[5] Seena Gressin. The equifax data breach: What to do. *Federal Trade Commission, Washington, DC*, 2017.

[6] Rui Zhang and Ling Liu. Security models and requirements for healthcare application clouds. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 268–275. IEEE, 2010.

[7] David Ferraiolo, Vijayalakshmi Atluri, and Serban Gavrila. The policy machine: A novel architecture and framework for access control policy specification and enforcement. *Journal of Systems Architecture*, 57(4):412–424, 2011.

[8] Peter Mell, James M. Shook, and Serban Gavrila. Restricting Insider Access Through Efficient Implementation of Multi-Policy Access Control Systems. In *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats*, pages 13–22. ACM, 2016.

[9] Indrakshi Ray and Tai Xin. Concurrent and real-time update of access control policies. In *International Conference on Database and Expert Systems Applications*, pages 330–339. Springer, 2003.

[10] Daniel Servos and Sylvia L Osborn. Hgabac: Towards a formal model of hierarchical attribute-based access control. In *International Symposium on Foundations and Practice of Security*, pages 187–204. Springer, 2014.

[11] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. Label-based access control: An abac model with enumerated authorization policy. In *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*, ABAC '16, page 1–12, New York, NY, USA, 2016. Association for Computing Machinery.

[12] OASIS Standard. extensible access control markup language (xacml) version 3.0, 2013.

[13] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. Abac with group attributes and attribute hierarchies utilizing the policy machine. In *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control*, pages 17–28, 2017.

[14] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, page 36, 2013.

[15] Marko A Rodriguez and Peter Neubauer. Constructions from dots and lines. *Bulletin of the Association for Information Science and Technology*, 36(6):35–41, 2010.

[16] Manachai Toahchoodee, Indrakshi Ray, and Ross M McConnell. Using graph theory to represent a spatio-temporal role-based access control model. *International Journal of Next-Generation Computing*, 1(2), 2010.