An Efficient Implementation of Next Generation Access Control for the Mobile Health Cloud

Rejina Basnet*, Subhojeet Mukherjee †, Vignesh M. Pagadala‡, Indrakshi Ray§
Department of Computer Science, Colorado State University
Fort Collins, CO, 80523, USA
Email: *Rejina.Basnet@colostate.edu, †Subhojeet.Mukherjee@colostate.edu,
‡Vignesh.Pagadala@colostate.edu, §Indrakshi.Ray@colostate.edu

***Abstract*—In today's era health informatics is a major contributor to the advancements in ubiquitous computing. Of late, the concept of mobile health (mHealth) systems has attracted considerable attention from both medical computer science communities. mHealth devices generate a significant amount of patient data on a timely basis. This data is often stored on cloud-based EHR and PHR systems to aid in timely and better quality healthcare service. However, as has been seen lately, stored personal records act as honeypots for malicious entities and the internet underground. It is thus imperative to prevent unauthorized leakage of mHealth data from cloud-based E/PHR systems. As observed from some of our preliminary research, NIST's policy machine (PM) framework suits the access control modeling requirements posed by mHealth systems. Moreover, the graph-based model adopted by this framework allows efficient policy management through advanced graph search techniques. In this paper, we leverage the policy machine model to propose a cloud-based service that achieves secure storage and fine-grained dissemination of mHealth data. The primary goal of this work is to demonstrate the applicability of the PM framework to the mHealth domain and illustrate the workflow of an algorithm to resolve access decisions in theoretically faster time than achieved by existing implementations.**

I. Introduction

A. Mobile Health Systems

Recent years have witnessed a significant growth in the field of healthcare information technology. This, coupled with advances made in mobile computing, have essentially laid the foundation for the advent of mobile-healthcare systems, abbreviated as 'mHealth'. mHealth can be broadly described as a system which involves the use of mobile computing devices in discharging healthcare-related services [1]. The architecture of this system, as described by Avancha et al. [2], comprises of a set of sensor nodes, mobile nodes, and a remote, centralized repository. Sensor nodes (SN) are basically attached to the patient's body in different configurations (based on what type of health-data is required), and are responsible for detecting and relaying information in real-time. For example, a sensor node attached to a person's wrist could be used in assessing the pulse rate of the user and relaying it to the appropriate end-point. Mobile nodes (MN), on the other hand, capture the information sent by the SNs and transmit the same to the remote repository, possibly after performing some processing, on the received data, and aggregating the same. These MNs could either be a mobile phone or a Mobile Internet Device (MID), and this is, in addition to the SN's, carried by the patient. The remote repository is referred to as the Health Records System (HRS), which is the centralized storage system holding the records of all the patients who have registered to the mHealth service, where data received from different MNs are stored after performing some preprocessing. Any entities such as physicians, researchers, insurance companies or lawyers who require access to patients' data, can obtain it from the HRS, with suitable permissions from the patient.

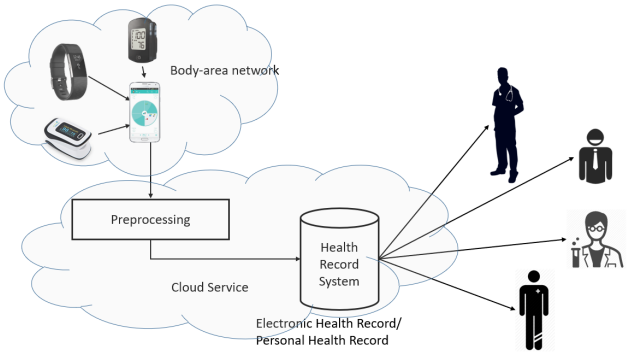


Fig. 1: mHealth Architecture

B. Threat Model and Problem Description

The mHealth system could be under threat by a myriad of sources in different ways. The sources of threat can, however, be broadly categorized as (1) Patient (the patient himself or herself), (2) Internal entities like doctors, insurance companies and HRS cloud service providers who can gain authorized access to some portion of the patient's data and (3) External entities like hackers and thieves who do not have authorized access to any portion of the patient's data [3]. As seen from Fig. 1, the assets in consideration are the MID, SNs, the preprocessing unit and the HRS itself. Any threat sources can get unauthorized access to stored records, perform malicious alterations or deny services provided by the critical units in the mHeatlh infrastructure (from Fig. 1). Although, all of the above mentioned cases are possible, in this paper, we only address the privacy concerns raised by unauthorized access to the mHealth data stored on the HRS. As observed by Kotz et al. [3], the concerned threat agents can be any of the three entities listed in the beginning of this paragraph.

From above mentioned threat model it is evident that it is imperative to prevent unauthorized access or disclosure of

978-1-5386-5896-3/18/$31.00 ©2018 IEEE

stored mHealth data. There are, however, two critical aspects to this problem that need attention. Firstly, an authorized entity may only require data access for a limited number of features in the patient's data record, which means, fine-grained access to data is required. An insurance company, for example, might have a need to peruse cardiovascular data only. Secondly, in the context of healthcare applications, fast access decisions are an absolute requirement. A slow HRS server can lead to catastrophic consequences for a patient in an emergency situation.

C. Existing Approaches Based on the Role and Attribute-based Access Control Paradigms

Role-based access control (RBAC) is made use of in traditional mHealth systems [2]. RBAC-based systems use user roles (eg. doctor, insurance-agent etc.) to make access decisions. Kulkarni et al. [4] have implemented the use of an RBAC mechanism for their mobile-healthcare solution. They argue that, RBAC would be more meaningful in the context of healthcare when compared with Mandatory Access Control (MAC). MAC's mechanism involves designating varying levels of sensitivity to an object based upon the information contained in it, and granting access to any subject based upon the subjects clearance. This mechanism is not suitable in the context of healthcare applications, since in this case, it is not only necessary to consider the information's sensitivity, but it is also required to ensure control over individual action, which leads to preference for RBAC. Kotz [3] suggests RBAC as well, for controlling access to a patient's records. RBAC, however, suffers from its own share of pitfalls, Firstly, role explosion and the presence of multiple domains will complicate policy management. In the context of a large organization such as a healthcare institute with many different administrative domains, RBAC is going to complicate things when it comes to making an access decision based on a role [5]. This is because, the concept of a role might differ across organizations or different domains withing an organization. Furthermore, the process of assigning roles and managing them is very difficult in a large organization [2]. Finally, RBAC does not consider object-level attributes in making access decisions, and therefore, is incapable of achieving finer levels of granularity[6].

Lu et al. [7] suggests the use of Attribute-Based Access Control (ABAC) in implementing an efficient access-control methodology to achieve privacy in their mHealth system. As an example, in RBAC all doctors can get access to certain objects, while in ABAC doctors who are employees of a certain hospital ("employeeOf" can be treated as an attribute here) can only get access to those objects. ABAC successfully overcomes the drawback presented by RBAC by making access decisions based upon different attributes of the user, object and the environment [8]. But it is still not a possibility to enforce multiple policy classes with their approach. We resolve this problem using PM.

D. Policy Machine for mHealth Systems

The problem domain addressed in this paper requires effective policy representation and enforcement in the mHealth cloud service's access-control system. PM [9] provides us with a suitable framework for defining policies and implementing them. In the context of mHealth, using PM can help us overcome drawbacks with respect to an RBAC methodology where it's difficult to identify and manage roles. Enforcement of policies are ensured using PM. Then again, as observed by Mell et al. [10], the PM framework can be represented using directed acyclic graphs (DAG), thereby opening up a wide range of possibilities for optimizing search and retrieval techniques. This, in turn, can speed up the process of evaluating access decisions. Considering these advantageous aspects of PM, we propose using PM to model the access control framework in mHealth systems. PM also forms the basis for the development of Next Generation Access Control (NGAC), and hence we use these two terms (PM and NGAC) interchangeably throughout the rest of the paper.

Using the PM framework we attempt to answer two types of user queries [10]:

1) Can a particular user u perform an operation op on an object o, i.e. $< u, op, o >$? As an example, "can Dr. Jekyll, read patient Mr. Hyde's cardiovascular records?".
2) What privileges does a user u have, i.e. $< u, *, * >$? As an example, "can Dr. Jekyll review all objects in the system that he can read or write?".

Apart from establishing the significance of both these types of queries, Mell et al. [10] also make the first attempt to implement graph search techniques to evaluate them. Their algorithms run in linear time with respect to the number of edges, under the assumption that the number of nodes are lesser than the number of edges in the PM graph. To demonstrate the efficiency of their approach, they compare its performance characteristics with that of the existing implementations and show significant improvement. Since the PM graph is a directed acyclic graph, in the worst case, the number of edges can reach up to $\binom{n}{2}$ and since this approach is very close to traditional exhaustive search techniques the time complexity can be anywhere from linear to quadratic in the number of nodes ($O(n^2)$). Consequently, in this paper we present an approach that is theoretically faster than their approach. To achieve this we make use of a modified version of an algorithm [11] that finds common predecessors of two nodes in a DAG. It must be noted, the performance evaluation shown in [10] are for queries executed on an in-memory representation of the PM graph. In contrast, we store and evaluate the same queries on persistent storage.

In this paper, we also devise a model for representing access control policies that allows fine-grained release of patient records. For example, if a patient's record consists of more than one item such as cardiovascular and mental health-related data, our model enables authorized release of individual items. Finally, we implement our NGAC framework on a graph database. This allows us to execute graph search algorithms natively on persistently stored access control policies.

E. Paper Outline

The rest of the paper is organized as follows. In section II we present, in details, the background knowledge required to clearly comprehend the work done in this paper. This includes a brief overview of the NIST NGAC [9] framework, the Neo4j graph database and the common predecessor finding algorithm [11]. Section III presents the salient features of the solution

architecture presented in this paper. We start by describing our solution architecture, followed by a thorough description of the NGAC graph generation process and algorithms used in answering access queries. In section IV we present a detailed analysis of our algorithms. Finally in section V, we summarize the contributions made in this paper, followed by a concise description of the tasks we aim to accomplish in the future.

II. Background

A. Policy Machine and Next Generation Access Control

PM is, in general, termed as a "logical machine" which consists of a fixed set of relations and functions between the policy elements.

1) NGAC Constructs: NGAC is constructed with basic policy elements and a fixed set of relationships. The policy elements and relationaships used in this paper are mentioned below.

a) Policy Elements:

- Users (U) The individuals authenticated by the system are called Users. They are identified by unique identifier in the system.
- Objects (O) Objects can be defined as system entities that are governed by one or more defined policies. They have a unique identifier in the system.
- Operations (OP) Operations denote actions that can be performed on the contents of objects that represent resources, or on NGAC data elements and relations, that represent policy.
- Access Right (AR) For users to carry out any operation, they need to have proper access rights. Access rights can be categorized as administrative and non-administrative.
- Policy Classes (PC) A policy class is used to organize and distinguish between different types of policies being expressed and enforced. A policy class can be thought of as a container for policy elements and relationships, that pertain to a specific policy.
- User Attribute (UA) User attributes also play a similar role as policy classes, that is, they act as containers which help with organizing and distinguishing between distinct classes of users.
- Object Attribute (OA) The role of user and object attributes are the same, besides the fact that object attributes help in distinguishing objects. Every object serves as an object attribute in the PM model, that is, objects are a subset of object attributes. However, this isn't true for user and user attributes.

b) Relationships: Relationships in the NGAC model are categorized into two groups.

- Assignment (ASSIGN) Assignments are used as a means to express relationships between users and user attributes, objects and object attributes, user (object) attributes and user (object) attributes, and user (object) attributes and policy classes.

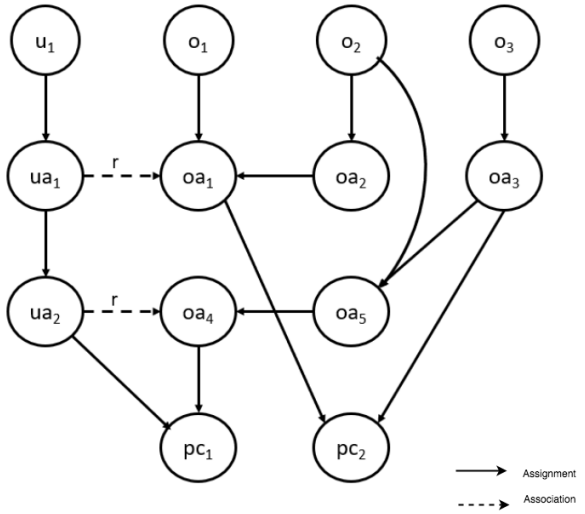


Fig. 2: Example PM Instantiation

- Association (ASSOC) Associations define relationships that involve the authorization of access rights between policy elements. They are the policy settings that govern which users are authorized to access which objects, and exercise which access rights. An association is represented as a ternary relation (UA x ARs x OA).

PM also consists of a notion of prohibitions and obligations which further assist in making the policies granular. However, we do not consider these constructs in our implementation. Also, an authorization graph can consist of more than one policy class. For the purpose of this paper, we utilize only one policy class.

2) Evaluating User Privileges: We make use of the sample policy shown in Fig. 2 to demonstrate the process of evaluating user privileges. Policy machine [9] defines a user privilege as the triple (u, op, o), where $u \in U, op \in OP$ and $o \in O$. In our model, $OP = \{r, w\}$. Using this notion, an access request of the form (u_1, r, o_2) would be authorized from Fig. 2, if the user and object nodes share a common operation node of the form r and policy classes. Mell et al. provide a more intuitive graph-theory oriented definition in [10]. According to the authors, the user u_1 can perform an operation r on the object o_2 if there exists an edge labeled r, the tail of which can reach u_1 and the head of which can reach o_2. Mell et al. also mandate that, a privilege is considered granted (for a user u_1 to access an object o_1) if, the set of all policy class nodes reachable from the head nodes is a superset of the set of policy class nodes reachable from o_1. However, since we use only one policy class in this work, our evaluation does not incorporate this aspect.

B. Graph Database: Neo4j

Neo4j is a NoSQL native (graph specific storage and processing engine) graph database that implements property graph model [12] to the storage label. In addition, it provides

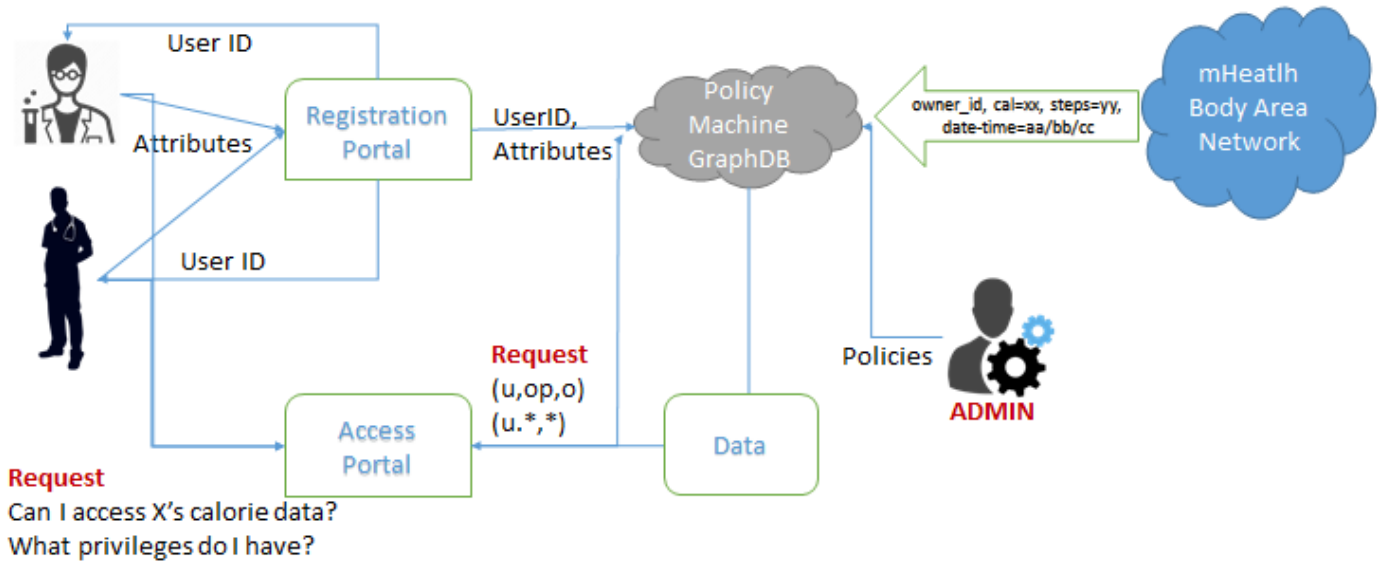


Fig. 3: Proposed Solution Architecture

facilities such as ACID properties. A property graph model is comprised of nodes, relationship, properties and labels. Nodes are main data elements connected with each other via relationships. The nodes, as well as the connecting relationship, can have properties of their own. In order to categorize the nodes into the related groups, labels are used. A node can have multiple labels at the same time. All the labels are indexed, in order to accelerate the process of finding nodes in the graph. Neo4j makes use of Cypher Query Language for its operation. Cypher is a declarative graph query language, inspired by SQL, with pattern-matching capabilities. Finally, Neo4j allows computation of a record's location in O(1) time. This is made possible by storing the file records into multiple node storage files. The records stored are fixed-sized (9 bytes). Therefore, if we have a node with an id equal to 100, then we know that the record begins 900 bytes into the file. Also, the use of relationships instead of indexes further aids into fast traversal of the graph. This feature of Neo4j is our motivation behind selecting it as our primary choice for storing and processing policies.

C. Common Predecessor Finding Algorithm

This algorithm is a graph traversal algorithm that allows us to find the common predecessor given a node in a directed, acyclic graph. It utilizes basic depth-first search to do so. The following paragraph describes the algorithm in a nutshell. Let us consider a graph G(V, E), where V represents vertices and E, the edges. G^t would represent the transpose of the graph G where all the directed edges of graph G are inversed. The vertex having zero incoming edges are the source nodes. Similarly, a vertex v is said to be a successor of a vertex w, if there exists a direct path from w and v. A vertex v is said to be a predecessor of a vertex w, if there is a direct path from v to w. Therefore, given two vertices u,v, this algorithm effectively solves the problem of finding whether u and v have a common predecessor. To solve this problem, a very naive way would be to run a depth-first search from u in G^t marking all the visited vertices and then performing a depth-first search from v in G^t, and determining whether any marked vertices are encountered. The time complexity of the algorithm, which is O(m), might increase to quadratic with increase in number of queries. Therefore, an indexed version of common predecessor is used which runs depth-first search from each source node, labeling all its successors with the source nodes. In cases where there are multiple sources in same node, they are appended to the back. If the labels to each vertices are at most p then the time complexity of the labeling would be O(pm) and the query takes O(p) to determine if the queried vertices share a common label.

III. Methodology

A. Solution Architecture

Fig. 3 shows the architecture of our reference implementation. As mentioned earlier in section II-A, the NGAC model includes user (u), user attribute (ua), object (o) and object attribute (oa) nodes. We obtain information pertaining to o and oa nodes from the incoming traffic that is received from the MN nodes (refer to section I-A). For the purpose of instantiating the model, we make use of a Fitbit dataset that is obtained from [13]. The dataset includes attributes such as 'Calories' (amount of calories burned by the wearer), 'Steps' (number of steps taken), 'Date-Time' (date and time of the observation), 'Distance' (distance traversed), 'PID' (patient ID), 'Elevation' (elevation at which the wearer is) etc. For the sake of simplicity and brevity, in this work, we only consider the features 'PID', 'Date-Time', 'Calories' and 'Steps'. This can be seen in Fig. 3 where, to be more application specific, we rename the 'PID' attribute to 'ownerid'. Each measurement (eg. "xx" for calorie or "yy" Step) is envisioned as a data object $o \in O$.

The architecture also consists of a *Registration Portal* via which we collect user information. Users are those entities which request for access to the data from HRS. Users can

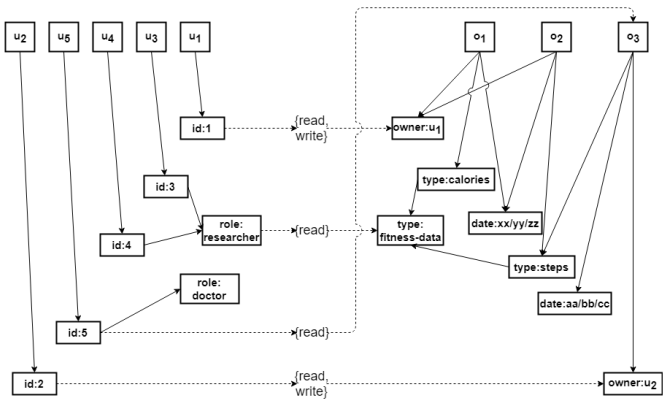


Fig. 4: An Example NGAC Policy for the mHealth Cloud

include, but are not restricted to, physicians, insurance companies, lawyers and patients. User attributes can be *role*, *name* etc. Once a user is registered, they are assigned and provided with a system-wide unique identifier. At the time of data retrieval users query via the *Access portal* and their queries are converted into NGAC formalized access requests. These requests are then evaluated using algorithms demonstrated later in the section III-D. If granted access, authorized objects (or set of objects) are returned to the user. Although, it is recommended to have user-centric policies in an mHealth system [14], in this work we assume this responsibility to be fulfilled by the system administrator. We however, plan to address this issue in future.

B. An Example NGAC Policy for the mHealth Domain

An institute using our mHealth-based cloud service can have its policies as shown in Fig. 4. The system protects unauthorized access to *fitness-data*, which in turn consists of the features *calories* and *steps*. Each object node has three attributes - *date* (the date on which the observation was made), *owner* (the patient with whom the data is associated to) and *type* (the type of data being recorded). There are a total of five users in the system. Users u_1 and u_2 are *owners*, that is, they are the patients in the mHealth system. Patients are allowed to possess read and write privileges over their records, since the institute does not consider knowledge of these features to be in any way a detrimental to the patient, and also believes that incorrect alteration of this data by the patient would not result in any significant harm. Users u_3 and u_4 identify as *researchers*, and are allowed read access to all fitness-data features of all the patients, by the institute. User u_5 identifies as a *doctor*, and according to the policies of this institute, is only allowed read access to his own patient u_2's *steps* on the *date* "aa/bb/cc".

C. Constructing the PM Graph

To construct the authorization graph for NGAC, we need to figure out the users, objects, user attributes, object attributes and permissions. Since, we have only one policy class we assign every element in the system to a single policy class. For the purpose of explaining the construction process we will be using the example policy from Fig. 4 as a reference.

1) Adding Nodes and Edges: As mentioned earlier, we collect objects and object attributes from the incoming traffic. As data is received, new object and object attribute nodes are added to the NGAC graph. Although, every incoming data item is considered to be a unique object, this might not be same for object attributes. This is because, in our simulation, we receive the same *type* attributes (calories and steps) every time. We however, might receive different *ownerid*s and *date-time* attributes, in which case, new *oa* nodes are created. When objects are created, they are assigned to all three types of object attributes namely, *type* (representing the type of recording it represents), *ownerid*s (representing the ownership) and *date-time* representing the time at which it was received. For users and user attributes we utilize the information received from the *Access portal*. It should be noted that we do not allow object creation before a patient's information is received. This is because, once patient data is received from the MNs, we assign patients default r, w privileges to the objects they own. For other users, once they register, they are provided unique *id*s, which become their primary attribute in the graph. The other user attribute supported in our simulation is *role* that determines the designation of a user in the system. Finally, at the time of installing policies the administrator makes changes to the NGAC graph by adding association relationships from user attributes to objects or object attributes. Unlike, the original NGAC guidelines, association relationships are represented as nodes (referred to as *op* nodes) which contain privileges to be assigned and connect them by incoming edges from user and object attributes. This makes it possible to find the required privileges through the modified common predecessor algorithm which will be explained later.

The system updates every time a new node or edge is added to the PM graph. And as it is seen above, a new node or edge is added once new object attributes/assignments (like "Calories" and "ownerid") or user attributes/assignments (like a new role) are being added. If new policies are requested, they are review by the administrator and corresponding associations are created. The Neo4j representation will contain all the policy elements as nodes relationships as edges. Each element of the graph are labeled respectively. The association relation represented as node will be labeled "action".

2) Constructing the Index: During the indexing process the authorization graph is indexed by two different kind of indexes depending on the nature of node being indexed. The user and object nodes i.e. (source nodes with in-degree 0) are indexed with the reachable op nodes, and the op nodes themselves are indexed with the source nodes they can reach.

In algorithm 1 we first traverse the PM graph in a depth-first manner from both user and object source nodes and add the visited nodes to *userlist* and *objectlist* (lines 1-7). This phase involves all the disk access and the next steps of processing in the indexing phase are all performed in-memory. Next we intersect the each *userlist* and *objectlist* and store every common node on the corresponding user and objects nodes in sorted order of id (note that each node in Neo4J is provided a unique identifier). This process indexes *op* nodes on user and object nodes (lines 8-14). At the same time, we also store the same user and object nodes on the *op* nodes (lines 15-22) in two separated sorted indexes. Both these indexes are used in next phases of the algorithm.

Algorithm 1 Indexing

```
for all source nodes do
    if source is an instance of user then
        Assign user userlist=set of all the nodes visited during DFS traversal from user
    else
        Assign object objectlist=set of all the nodes visited during DFS traversal from object
    end if
end for
for all nodesA in user.userlist do
    for all nodesB in object.objectlist do
        if nodeA.id = nodeB.id then        ▷ nodeA is an op node
            user.cp_list ← user.cp_list ∪ nodeA
            sort(user.cp_list)
            object.cp_list ← object.cp_list ∪ nodeA
            sort(object.cp_list)
            if nodeA.userlist does not contain user node then
                nodeA.u_cp_list ← nodeA.u_cp_list ∪ user
                sort(nodeA.u_cp_list)
            end if
            if nodeA.objectlist does not contain object node then
                nodeA.o_cp_list ← nodeA.o_cp_list ∪ object
                sort(nodeA.o_cp_list)
            end if
        end if
    end for
end for
```

Note that the indexing procedure is done for every change in the graph, i.e. new either a new node or a new edge is added. However, since the index is precomputed we do not consider the indexing time for the purpose of performance evaluation. Furthermore, assuming unbounded disk space, since the index is stored in-disk with the nodes we do not consider the space overhead for the indexing process.

D. Evaluating Access Requests

After every indexing the graph is ready to be queried. As already mentioned earlier in section I-D, we address two types of queries in this paper. The following two subsections delineate the procedure of evaluating these queries.

1) Queries of the form $< u, op, o >$: For the query of the form $< u, op, o >$ we first get the indexes for each of the u and o nodes as shown in lines 1-2 of algorithm 2. We then perform an intersection on the two lists. Intersection of two lists of length n and m respectively, typically take $n * m$ operations. However, the same process can be done on sorted lists using $n + m$ operations by traversing each list only once. This is shown in lines 4-15. We start by initiating two pointers i and j to point at the two indexes. If the items pointed to by the pointers are matching and of the form of op ("r" or "w" for example) we return at line 7 thereby granting the operation. Otherwise, increment pointer which points to the lower value.

Algorithm 2 Evaluate $< u, op, o >$

```
1  cplist_for_u ← get cp_list property of node u
2  cplist_for_o ← get cp_list property of node o
3  Let i = 0, j = 0
4  while i < len(cplist_for_u) ∨ j < len(cplist_for_o) do   ▷ Intersection of sorted list
5      if cplist_for_u[i] = cplist_for_o[j] then
6          if cplist_for_u[i] is the op node in the query then
7              return Granted
           end if
       else
           if cplist_for_u[i] < cplist_for_o[j] then
               increment i by 1
           else
               increment j by 1
           end if
       end if
   end while
   return Denied
```

In this way, we only traverse both the indexes once. Eventually, if no matching node is found, we deny access.

Algorithm 3 Evaluate $< u, *, * >$

```
for all op nodes do
    if op.userlist contains user then        ▷ Perform binary search
        return op and op.objectlist
    end if
end for
```

2) Queries of the form $< u, *, * >$: Algorithm 3 shows the final procedure where the user attempts to review all his/her rights. In this case, we traverse the list of all op nodes and for each node we perform a binary search on the sorted index of reachable user nodes (refer to section III-C2). If a match is found, the indexed object list is returned, denoting the user can perform op on the returned set of o nodes.

IV. Analysis and Discussion

A. Complexity Evaluation

Let us consider the number of user nodes to be $|U|$, object nodes to be $|O|$, user attribute nodes to be $|UA|$, object attribute nodes to be $|OA|$ and operation nodes $|OP|$. In this section we evaluate the worst case runtime complexities of the two types of queries namely, $< u, op, o >$ and $< u, *, * >$.

a) Evaluating $< u, op, o >$: For this query request, we pull the sorted index property of both u and o nodes, which is O(1) for Neo4j. An intersection, which is performed on this sorted list, would have less complexity than a regular intersection. In algorithm 2, since we only traverse each index once for the intersection, the complexity at the worst case is O($|OP| + |OP|$) or $O(|\ OP\ |)$.

b) Evaluating $< u, *, * >$: As seen in algorithm 3, for each op node that is scanned we perform a binary search on the indexed user nodes. This leads to a time complexity of $O(|OP|log(|U|))$.

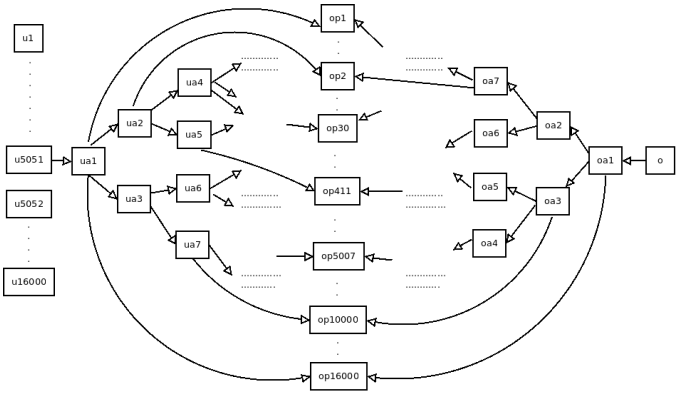


Fig. 5: Custom PM Graph for Worst Case Experiments

B. Performance Analysis

1) Experimental Setup: To evaluate the performance characteristics of our approaches, we performed a few experiments in two categories namely, the *worst case* scenario and *real world (average case)* scenario. It must be noted for conducting experiments in both categories, we used a

a) Worst Case Scenario: The worst case scenario demonstrates the theoretical upper bounds of our approaches, as deduced in the the previous section. For this purpose we conducted 3 experiments with a custom generated PM graph (Fig. 5) that reflects the worst case scenario. To generate this graph we first created two separate single-rooted tree structures using ua and oa nodes respectively. We then mapped each ua node to each oa node thus creating a total of $|UA| * |OA|$ op nodes. We then added new u_1 and o_1 nodes and made them new roots as shown in Fig. 5. Under this model, the user in the graph is allowed to perform each operation on the o_1 node and hence the indexing processes would index every op node for user node. For the first two experiments (querying $< u, op, o >$ and $< u, *, * >$ respectively), we steadily increased the height (h) of both trees to generate $2^{h+1} - 1$ ua and oa nodes in each iteration and connected each combination to increase the number of op nodes. Eventually, we stopped at a total of 16000 op nodes. For both experiments we disabled the return statement at line 7 of algorithm 2 to enable full scan of the index and noted the time elapsed after each query thereby plotting the results as shown in figures 6a and 6b respectively. For the third experiment in this category, we steadily created a large number u nodes and assigned each u node to the root ua node, thus allowing the indexing process to index every op node for each user and vice-versa. We then executed the $< u, *, * >$ query for each incrementing value of $|U|$ and noted the time elapsed after each query and plotted the results as shown in figures 6c. Finally, for this experiment, we query by setting the user node identifier to the minimum so as to enable full binary search on the index.

b) Real World (average case) Scenario: In their work, Mell et al. [10] evaluated the performance of their approach on simulated PM graphs that reflect a real world scenario. In accordance with their experimental setup, we increased 4 u nodes, 4 user attributes, 20 o nodes and 12 object attributes in each iteration. Mell et al. generated operation edges, the number of which is determined by the probability where Edros-Renyi graph will have mean edges per node less than five. Since, in our approach, we convert these edges into nodes (op nodes), we did the same but for the number of nodes. With this setup we noted the time taken for each type of query ($< u, op, o >$ and $< u, *, * >$) with the number of nodes in each iteration. The rate of increment could not be fixed to 1 because of the randomness introduced by Edros-Renyi principle.

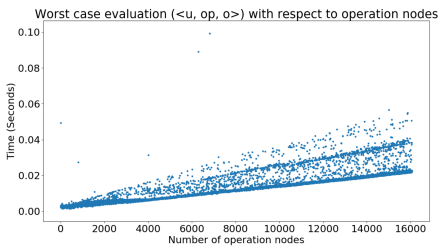
2) Discussion: As can be seen in Fig. 6, in the worst case our approach performs linearly with increasing number of operation nodes and in a slowly rising manner for the user nodes. Although, the curve in Fig. 6c did not exactly follow in a logarithmic pattern, our approach performed significantly well, achieving an almost constant curve with respect to increasing number of user nodes. We believe that the most significant revelation from these experiments was that, even in the worst case and in spite of the disk access overhead, the performance of our approach was in order of a much lower value than a tenth of a second for about 16000 op or u nodes.

The performance curves observed in Fig. 6 prompted us to believe that in a real world scenario our algorithm would perform even better. Fig. 7 validates the same. Our average case analysis did not show any increasing trend with increasing the number of nodes. This leads us to believe that, in spite of the disk access times, our approach can run in almost constant time with increasing number of nodes, thus yielding a significant performance boost while evaluating access requests in a real world scenario. This performance also aligns with what has been observed by Mell et al. in [10]. Although, we achieve increased performance, we must note that Mell et al. perform an extra step for validating the containment of policy classes, which we avoid in this paper.

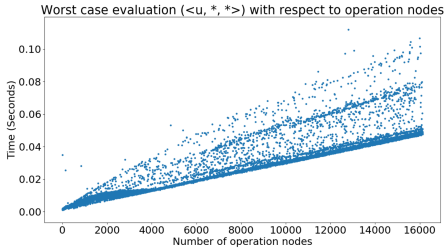
V. Conclusion and Future Work

In this work, we looked into the prospect of harnessing NGAC for designing an access-control framework for mHealth systems. We examined some basic concepts involved in NGAC, and assessed its feasibility for use in mHealth. An example policy use-case was used to construct the NGAC graph and evaluate access requests. Our investigations revealed that using NGAC gives us a boost in terms of speed, and that our approach could theoretically achieve a lesser time complexity (as compared to the previous work by Mell et al. [10]) in evaluating access decisions. The complexity to evaluate access decisions achieved by Mell et al. was reported to be linear in the number of edges while in our case, it is linear in the number of operation nodes (we converted operation labeled edges into nodes) for making access decisions and loglinear in the number of user and operation nodes to review access rights. Furthermore, NGAC supports the enforcement of multiple policy classes in one policy and is a less complicated access-control system to implement and manage, thereby justifying its use for an mHealth system.

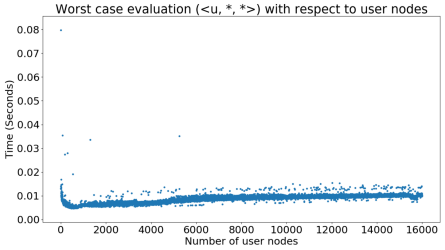
In future we aim to extend our reference implementation in more than one way. Firstly, we want to make policy management more owner-centric. We plan to use web-based authentication schemes like o-auth/openid-connect to achieve this. Secondly, we aim to further optimize the process of query evaluation. Thirdly, we plan to integrate more policy classes



(a) Worst Case Performance for $< u, op, o >$ with increasing $|OP|$

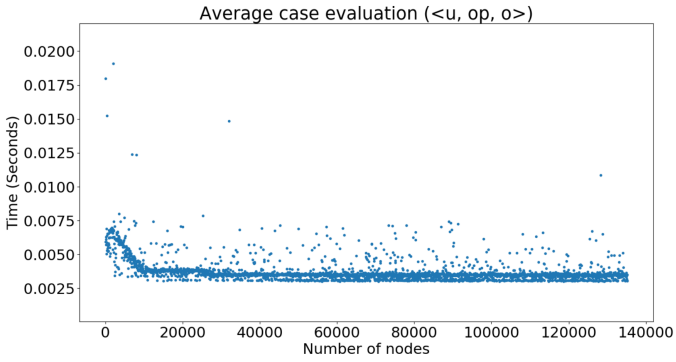


(b) Worst Case Performance for $< u, *, * >$ with increasing $|OP|$

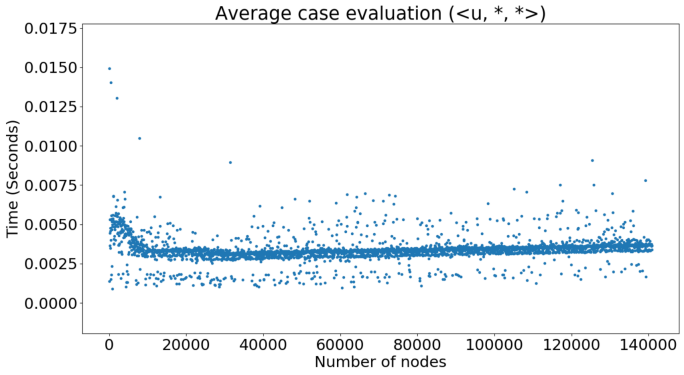


(c) Worst Case Performance for $< u, *, * >$ with increasing $|U|$

Fig. 6: as



(a) Real World Performance for $< u, op, o >$



(b) Real World Performance for $< u, *, * >$

Fig. 7

into the PM graph thereby better utilizing its features. Fourthly, we plan to implement a faster, non-exhaustive search technique to construct the index. This will allow better and more timely policy management. Finally, we aim to formalize and extend the current model to inculcate more versatile models like spatio-temporal schemes for access control.

ACKNOWLEDGEMENT

The work is supported in part by grants from NIST under contract number 60NANB16D250, NSF under award number CNS 1650573, AFRL, SecureNok, CableLabs, and Furuno Electric Company.

REFERENCES

[1] S. Adibi, *Mobile Health: A Technology Road Map*. Springer Publishing Company, Incorporated, 2015.

[2] S. Avancha, A. Baxi, and D. Kotz, "Privacy in mobile technology for personal healthcare," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 3:1–3:54, Dec. 2012.

[3] D. Kotz, "A threat taxonomy for mhealth privacy," in *Proceedings of the 3rd International Conference on Communication Systems and Networks*. IEEE, Jan 2011, pp. 1–6.

[4] P. Kulkarni and Y. Ozturk, "mphasis: Mobile patient healthcare and sensor information system," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 402–417, 2011.

[5] A. Karp, H. Haury, and M. Davis, "From abac to zbac: The evolution of access control models," in *Proceedings of the 5th International Conference on Information Warfare and Security 2010*. Academic Conferences Ltd, 2010, pp. 202–211.

[6] S. Mukherjee, I. Ray, I. Ray, H. Shirazi, T. Ong, and M. G. Kahn, "Attribute based access control for healthcare resources," in *Proceedings of the 2Nd ACM Workshop on Attribute-Based Access Control*, ser. ABAC '17. ACM, 2017, pp. 29–40.

[7] R. Lu, X. Lin, and X. Shen, "Spoc: A secure and privacy-preserving opportunistic computing framework for mobile-healthcare emergency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 3, pp. 614–624, March 2013.

[8] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone *et al.*, "Guide to attribute based access control (abac) definition and considerations (draft)," *NIST special publication*, vol. 800, no. 162, 2013.

[9] D. Ferraiolo, V. Atluri, and S. Gavrila, "The policy machine: A novel architecture and framework for access control policy specification and enforcement," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 412–424, 2011.

[10] P. Mell, J. M. Shook, and S. 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*. ACM, 2016, pp. 13–22.

[11] M. Toahchoodee, I. Ray, and R. M. McConnell, "Using graph theory to represent a spatio-temporal role-based access control model," *International Journal of Next-Generation Computing*, vol. 1, no. 2, 2010.

[12] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, vol. 2324, 2013, p. 36.

[13] R. D. Furberg, "Self-generated Fitbit dataset 10.22.2011-09.20.2014," Feb. 2015. [Online]. Available: https://doi.org/10.5281/zenodo.14996

[14] B. Martínez-Pérez, I. De La Torre-Díez, and M. López-Coronado, "Privacy and security in mobile health apps: a review and recommendations," *Journal of medical systems*, vol. 39, no. 1, p. 181, 2015.