Restricting Insider Access through Efficient Implementation of Multi-Policy Access Control Systems

Peter Mell James Shook Serban Gavrila

ABSTRACT

The American National Standards Organization has standardized an access control approach, Next Generation Access Control (NGAC), that enables simultaneous instantiation of multiple access control policies. For large complex enterprises this is critical to limiting the legally authorized access of insiders. However, the specifications describe the required access control capabilities but not the related algorithms. Existing reference implementations have inefficient algorithms and thus do not fully express the NGAC's ability to scale. For example, the primary NGAC reference implementation took several minutes to simply display the set of files accessible to a user on a moderately sized system. To solve this problem we provide a efficient algorithm, reducing the overall complexity from cubic to linear. Our other major contribution is to provide a novel mechanism for administrators and users to review allowed access rights. We provide an interface that appears to be a simple file directory hierarchy but in reality is an automatically generated structure abstracted from the underlying access control graph that works with any set of simultaneously instantiated access control policies. Our work thus provides the first efficient implementation of NGAC while enabling user privilege review through a novel visualization approach. It thereby enables the efficient simultaneous instantiation of multiple access control policies that is needed to limit insider access to information (and thereby limit information leakage).

1. INTRODUCTION

Most operating systems provide simple access control mechanisms that are focused on enabling users to specify which other users have access to their files (i.e., Discretionary Access Control (DAC) [13]). However, many other access control approaches exist that provide enhanced features, especially for enterprise environments. This includes capabilities relevant to particular paradigms (e.g., for scenarios with financial transactions, handling of classified data, and conflict of interest) as well as greater simplicity in administering access control at scale (e.g., Role Based Access Control (RBAC) [5]). However, methods to use multiple approaches within a single enterprise have been lacking, that can result in enterprises settling for using a single simple model (e.g., DAC).

This can result in restrictions on insider access being defined very loosely, increasing the risk of insiders having unnecessary access to sensitive information and sharing that information outside of the organization. To ensure that users don't inappropriately share data, enterprises may then resort to the costly and inefficient approach of separating different data types (e.g., military classification levels) into totally distinct and isolated networks. Alternately, they may accept the risk of data being leaked, which can have disastrous results (e.g., classified documents being made public).

The American National Standards Institute (ANSI) has addressed this problem by standardizing an access control approach, Next Generation Access Control (NGAC) [6]. The NGAC stems from and is in alignment with the Policy Machine (PM) [9], a research effort by the National Institute of Standards and Technology (NIST) to develop a general purpose Attribute Based Access Control (ABAC) framework [12]. The NGAC is designed to enable simultaneous instantiation of multiple access control policies. The specification describes what constitutes a valid implementation using set theoretic notation but does not provide implementation guidance. This approach then leaves room for multiple competing approaches and implementations. In this work we will explore how the inefficiencies of the existing approaches make it infeasible to tightly restrict user access to data in large enterprises and we will provide scalable replacement algorithms that solve this problem.

To do this, we improve upon the algorithms in the open source NGAC reference implementation provided by NIST [3] in order to test achievable efficiency and scalability. Among other increases in efficiency, ours is the first to provide quadratically bounded algorithms for the retrieval of the set of user accessible objects where the existing reference implementations take cubic time (we are linear in making an access control decision for a single object). Furthermore, our algorithms are bounded to operating only on the access control sub-graph pertinent to a particular user (not the entire enterprise access control graph). This work is thus the first to demonstrate that NGAC can be scalable through the use of efficient graph algorithms. This in turn makes it feasible to tightly restrict insider access to data through use of multiple types of access control policies (and thus limit the leakage of sensitive data).

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

For our initial efforts, we took the initial NGAC proof-of-concept implementation created by NIST [3] and evaluated the runtime of the algorithms. These were created by a highly skilled programmer who implemented the access control operations from the specification as they were stated in set theory notation (using a direct translation approach). This resulted in a functional system but one that could only scale to a couple of hundred users. At this scale, it took several minutes for a user to visualize their set of available objects. A complexity analysis of the code revealed cubic algorithms, explaining the lack of scalability of the implementation. We also looked at the other publicly available implementation of the NGAC which is provided as open source by Medidata [2]. Here, we found a user privilege determination algorithm that spent unnecessary time processing parts of the graph not accessible to the relevant user as well as a slow cubic algorithm for retrieval of all objects available to the user. It appears that with both implementations, it was an attempt to directly implement the NGAC set theory definitions that yielded the inefficient algorithms.

To solve this problem, we took a graph theoretic view to design an efficient algorithm for access control determination. We started by transforming the NGAC set theory into a graph representation (this was straightforward as the specifications themselves often use graphs to illustrate examples). Unfortunately, the resultant graphs had unusual features and constraints (with five different types of nodes, each with its own semantics). Thus, the primary challenge was in how to apply standard graph algorithms to this representation. Our solutions in general was to use breadth first search (BFS) and a depth first search (DFS) variant that performs a type of topological sort as primitive operations to allow us to cascade information from one type of node to another and percolate that information through the graph until the final answers are determined. The resultant algorithm is linear. Furthermore, it is not linear in relation to the entire access control graph, but only to the portion of the graph relevant to a particular user. This can offer even greater speedups, avoiding the need to even traverse the entire graph.

Besides not providing algorithms for calculating access control decisions, the NGAC standard does not provide any guidance on visualizing access control results to allow review of user privileges (see [11] and [15] for evidence of how this 'before the fact audit' capability is critically important). We presume the reason they do not provide this capability is because each access control policy may have its own preferred method for administrative review and user interaction. However, such a policy oriented approach isn't ideal in a system that simultaneously implements multiple policies (which is the whole point of NGAC). For example, the existing NIST PM approach requires users to choose a particular access control policy first and then navigate just within that policy structure to review user access (requiring the administrators and users to be knowledgeable about each access control policy and which files are covered by which policies). Because of these problems, there exists a need for a generic approach for user rights visualization that will work for any set of policies that can be instantiated within NGAC (without the staff having to understand said policies). Furthermore, this default visualization would ideally be automatically generated from the existing access control graph to avoid additional and excessive administrative burden.

To meet this need we provide the user (or the person reviewing the user's privileges) the visual experience of traversing a typical file directory hierarchy, as used by most major operating systems. However, under the hood the user is actually traversing the NGAC access control graph. We leverage one of the graph node types (object attributes) to act as file 'directories' enabling users to access their files. The user visually sees a tree but is actually traversing a graph with an exponential number of possible paths (where we generate local views on demand to avoid exponential calculations). Since the directory tree is automatically generated from the underlying graph, it can thus provide default user access to files simultaneously protected by multiple access control policies. An interesting side effect of our approach is that there can be multiple ways for a user to access the same file, without the need to explicitly create symbolic links. Thus, a document can be both stored under a person's personal directory and under a project directory with no duplication, system inconsistency, or need to explicitly create virtual links.

NGAC is not the only multi-policy access control system available. The current market leader appears to be the XACML standard [14] from OASIS [4]. Others include ABAC_α [12], HGABAC [16], and ABAC for Web Services [18]. The market leader XACML has been shown empirically to lack scalability in [17] where 3 different XACML implementations all experienced performance problems as the number of policies was increased past 100 (each policy in XACML contains the access rules for a set of target objects). This is not surprising as all of these logic-based formula approaches have been shown to be NP-complete with respect to enabling an administrator to review who can perform what actions. This is because policy review for such systems maps to the satisfiability problem [7]. This makes them undesirable for large enterprise systems with respect to ensuring the restrictions on insider access to sensitive data to avoid information leakage by insiders.

In summary, the contributions of this paper include:

1. the first ever study demonstrating the scalability of the NGAC multi-policy access control system,

2. a linear time algorithm for determining the objects and associated operations available to a user,

3. a novel visualization approach to enable review of user object access on NGAC systems,

4. and the ability for enterprises to efficiently implement multiple access control policies (which can limit insider access to information and thereby limit information leakage).

The remainder of this paper is structured as follows. Section 2 provides an overview of access control graphs within NGAC and provides a definition of when a user is allowed to access an object. Section 3 presents our access control algorithms and section 4 presents our visualization approach. Section 5 discusses related work and section 6 concludes.

2. ACCESS CONTROL GRAPH OVERVIEW

Using the NGAC specification [6] set theoretic definitions, we can form access control graphs as follows. There are 5 types of nodes to be created: user (u), object (o), user

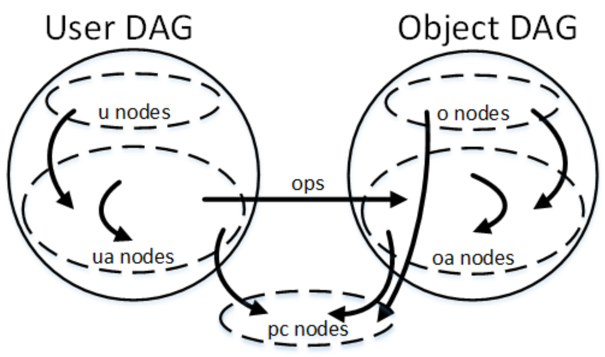

Figure 1: Diagram showing allowed edge relationships between the five different sets of NGAC node types

attribute (ua), object attribute (oa), and policy class (pc). All edges are directed. u nodes are sources with edges to ua nodes. ua nodes may have edges to ua, oa, and pc nodes[1]. oa nodes may have edges to oa and pc nodes. o nodes may have edges to oa and pc nodes. pc nodes are sinks. Cycles and self-loops are prohibited. ua to oa edges are labeled with a set of one or more allowed operations (ops) (e.g., read or write). All other edges are unlabeled. All nodes must have a path to at least one pc node (without using any ua→oa edges). For complexity evaluation purposes, the number of u, o, ua, and oa nodes are unbounded. However, the number of pc nodes and the number of distinct ops are assumed to be small constants.

These connectivity restrictions result in several features that we can leverage. The overall graph is a directed acyclic graph (DAG) that can be divided into two DAGs: a user DAG (with u and ua nodes) and an object DAG (with o and oa nodes). The set of u nodes act as sources for the user DAG and the set of o nodes act as sources for the object DAG. The set of ua to oa edges bridge the two DAGs and this bridge is the only place where edges are labeled, with operations (ops). We refer to the set of nodes on either side of these bridging edges as border nodes. The set of pc nodes act as sinks for both DAGs. The resulting overall graph is weakly connected.

An arbitrary access control graph can now be represented as shown in figure 1. Arrows within a set represent that nodes of that type can have edges to other nodes of that type, with no cycles allowed. This means that there are no edges between nodes within the set of pc nodes (the same is true for the set of u nodes and the set of o nodes). The arrows from the set of ua nodes to the oa nodes nodes represent the bridge edges (they contain the ops labels and connect the user and object DAGs). The bridge edges are the focal point in determining user privileges (see definition 1 below).

We now discuss how to determine user privileges. The ANSI NGAC standard provides set theoretic notation to enable computation of privileges abstracted away from any particular implementation. In this work, we describe the methodology using a graph oriented approach. Our graph theoretic derivation of the ANSI NGAC set theoretic definition of how to calculate access control is as follows[2]:

[1]In the NGAC specification, ua→pc edges are allowed but are not used for access control decisions.

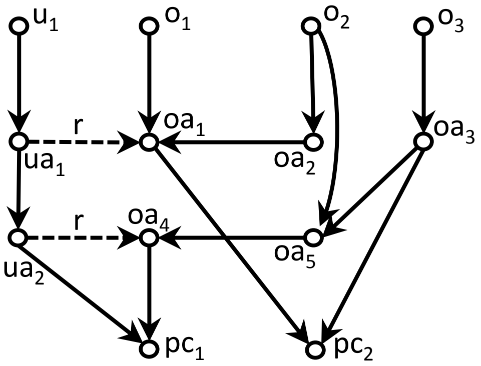

Figure 2: Example NGAC access control graph.

Definition 1. *For a user, u_1, to perform an operation, op_1, on some object, o_1, there must exist a set of ua to oa edges with label op_1 such that the tail of each edge is reachable from u_1 and the head of each edge is reachable from o_1 and where the set of pc nodes reachable from the set of head nodes is a superset of the set of pc nodes reachable from o_1.*

Figure 2 shows an example NGAC access control graph which we will evaluate using definition 1[3]. Note that the dashed edges represent the bridge edges that connect the user DAG to the object DAG. The edge label 'r' represents read access. In this figure, user u_1 can read o_1 and o_2 but not o_3:

- o_1 requires pc_2 because there is a path connecting the two. This requirement is fulfilled by the edge $ua_1 \rightarrow oa_1$ providing 'read' access(because ua_1 is reachable from u_1, oa_1 is reachable from o_1, and pc_2 is reachable from oa_1). Thus by definition 1, u_1 can read o_1.

- o_2 requires pc_1 and pc_2 because there is a path connecting o_2 with both pc nodes. This requirement is fulfilled by a combination of the edges $ua_2 \rightarrow oa_4$ (which covers the pc_1 requirement) and $ua_1 \rightarrow oa_1$ (which covers the pc_2 requirement). Note that these two bridge edges would not have fulfilled the requirements had they different labels. Thus by definition 1, u_1 can read o_2.

- o_3 requires pc_1 and pc_2 because there is a path connecting o_3 with both pc nodes. Edge $ua_2 \rightarrow oa_4$ covers the pc_1 requirement for o_3. However, there does not exist a ua to oa edge that will satisfy o_3's requirement to cover pc_2. Edge $ua_1 \rightarrow oa_1$ does not work because oa_1 is not reachable from o_3 (which is required in definition 1). Thus by definition 1, u_1 cannot read o_3.

[2]We don't include the NGAC definitions here because they use completely different set theoretic notation that would require extensive explanation and that is available in the NGAC standard).

[3]The edge $ua_2 \rightarrow pc_1$ fulfills the requirement in the specification that all u and ua nodes have a path to a pc node (without using bridge edges). However, the edge is not used for determining user privileges and will not be discussed further.

u_1 =	'Bob'
ua_1=	'Bob Privileges'
ua_2=	'Death Star Personnel'
o_1 =	'Tatooine Vacation'
o_2 =	'Defense Systems Finances'
o_3 =	'Energy Shield'
oa_1 =	'Bob Personal'
oa_2 =	'Bob Deathstar Files'
oa_3 =	'Technical Designs'
oa_4 =	'Deathstar Project'
oa_5 =	'Defense Systems'
pc_1 =	'Access Control System 1'
pc_2 =	'Access Control System 2'

Table 1: Node Labels for Access Control Graph in Figure 2

Throughout this paper, we will use Figure 2 as an example where an accountant, Bob, is working on the defense systems finances for a deathstar. In this context, we can interpret the nodes as shown in Table 1. Note how the user attribute nodes represent 'teams' to which Bob belongs (his own team[4] and the death star personnel team) and the object attribute nodes provide a kind of hierarchy for different projects. In this example Bob has access to his own data (oa_1, oa_2, and o_1) as well as the overall project and financial material (oa_4, oa_5, and o_2). However, Bob does not have access to any of the technical designs (oa_3 and o_3).

3. ACCESS CONTROL ALGORITHMS

We now provide a linear time complexity graph algorithm to answer two of the most common types of access control requests: 1) is user, u_1, allowed to perform operation, op_1, on object, o_1 and 2) what is the set of accessible objects for a user, u_1, and what operations can u_1 perform on each object. Both of these determination can be made through a slight variation on the same algorithm, which we refer to as 'Fastg'.

3.1 The Fastg Algorithm

The Fastg algorithm first isolates the problem to just the object DAG through labeling each border oa node reachable from u_1 with a set of operations (from the ops labels on the bridge edges). Then, the set of objects of 'interest' are found by performing a reverse BFS from the set of reachable border oa nodes (without traversing any bridge edges). If we are simply trying to determine if u_1 can access a particular object, we intersect this object with the set of objects of interest (forming a new set of objects of interest). Finally, we perform a DFS from each object of interest and percolate up through the graph the set of reachable pc nodes and the set of reachable operations. Finally, for each reachable object, we compare the set of operations associated with each reachable pc node to determine if any operations are valid. For an operation to be valid it must be associated with all the pc nodes reachable from the o node.

In more detail, the algorithm is as follows:

[4]We had to create ua_1 to represent Bob's access rights because the NGAC specification does not allow creation of u to oa edges.

1. BFS from u_1 to identify the set reachable ua border nodes (do not traverse oa nodes). For this set of ua border nodes, let the set of 'active' edges be the ua$\rightarrow$oa outedges.

2. For each 'active' edge, label the oa head node with the ops edge label (eliminating duplicates). At this point, each reachable border oa node is labeled with a set of access rights.

3. Create a temporary node that is a successor of each reachable border oa node

4. Perform a backwards BFS from the temporary node (traversing edges in reverse) to find the set of objects of 'interest'. Do not traverse any bridge edges. Once done, delete the temporary node.

5. If the goal is to determine if u_1 can access a specific object, then intersect this object with the set of objects of interest to form a new set of objects of interest (this set will contain either a single node or be the empty set).

6. For each object of interest, perform a DFS to find the reachable pc nodes. However, when performing a DFS, label all nodes with the information found such that subsequent DFSs can take advantage of the previously computed information. Each object of interest then is labeled with its set of reachable pc nodes. These represent 'required' pc nodes for each object. Note that to record on each node which pc nodes are reachable, we need to modify the traditional DFS such that we only process a node (record the set of reachable pc nodes) if all of its successors have been processed. If a node with unprocessed children is pulled off the stack, we must put it back on the stack. The second time it is pulled off, it will be guaranteed that its children will all be processed.

7. While performing the modified DFSs from the previous step, perform an additional data propagation. When a reachable border oa node is labeled with its reachable pc nodes, associate those pc nodes with the operation labels from step 2. Then use the normal operations of the DFS to propagate these pc/operation pairings up to the root of the tree (one of the objects of interest). We use the same 'trick' from the previous step to reuse information between DFSs.

8. For each object of interest, compare the set of required pc nodes against the pc/operation pairings. If for some object, o_1, an operation, ops_1, exists that is associated with each required pc node, then u_1 is allowed to perform ops_1 on o_1 per definition 1.

The algorithm is apparently quadratic because we may perform a DFS from each object node. However, in steps 6 and 7 we store DFS results at each processed node such that the information can be reused by other DFSs. As a result, the set of executed DFSs is guaranteed to traverse each edge in the object DAG at most twice. The BFS from step 1 traverses each edge in the user DAG once and each bridge edge once. Step 2 traverses each bridge edge once. And step 4 traverses each object DAG edge at most once. In summation, each edge in the graph is then guaranteed to

Number of user nodes $= .1 * n$
Number of user attribute nodes $= .1 * n$
Number of object nodes $= .5 * n$
Number of object attribute nodes $= .3 * n$
Number of pc nodes $= 3$

Table 2: Proportion of Nodes of Each Type

be traversed at most 3 times (most much less and some not at all). This makes the algorithm linear with respect to the number of edges, O(m).

3.2 Empirical Algorithm Results

In this section, we evaluate the scalability of our Fastg algorithm versus the two available reference implementations (NIST PM and Medidata). For our experimental platform we used an Ubuntu virtual machine with two cores and 10 Gb of memory running on a commodity laptop. For software to encode the algorithms, we used Python 2.7 and NetworkX (a graph algorithms library). Faster execution times can be achieved through use of more efficient programming languages (e.g., C) but our goal is to evaluate relative performance of the algorithms. These results then are an upper bound on what can be achieved relative to execution time. With respect to memory, none of the algorithms used even a majority of the available memory and thus we do not report memory usage statistics.

For our empirical scalability study, we used the Fastg variant that computes the set of accessible objects for a particular user and the set of operations available for each object. For comparitive purposes, we coded up the analogous algorithms from both NGAC reference implementations (the NIST PM [3] and Medidata [2]) using the same language and libraries. These implementations are discussed in 1 and then further in 5. The Medidata algorithm was perfectly analogous (identical inputs and outputs), however the NIST PM algorithm performed additional work not required to obtain our desired output. For example, the NIST PM algorithm outputs the oa nodes accessible to a user, not just the o nodes. To avoid unfairly penalizing the NIST PM algorithm, we included in our implementation only those parts relevant to obtaining the desired output.

To test the scalability of the algorithms, we generated access control graphs that varied in size from 1000 to 700,000 nodes. We used the number of nodes,n, as the independent variable and then scaled all other graph features relative to n. The proportion of nodes of each type (u, ua o, ou, and pc) are shown in table 2. For edges, we calculated an Erdos-Renyi edge probability, p, used to create random graphs [8] such that the mean number of edges per node would be no more than 5. Then, for each candidate edge allowed by the NGAC specification, we used p to determine whether or not to place the candidate edge in the graph. The only exception is that we limited the length of the u to pc paths in the user DAG and o to pc paths in the object DAG to be at most 5. We did this for the user DAG by dividing the ua nodes into 4 groups labeled with consecutive integers. Edges leaving a node were only allowed to go to nodes in groups with higher labels (edges within a group were not allowed). A similar operation was performed for the object DAG.

There do not exist any references that one can leverage for creating random NGAC graphs. Thus, we assigned the above parameters according to qualitative expert domain knowledge to create as realistic NGAC graphs as possible. To make sure that any particular parameter choice did not unfairly hamper one of the algorithms, we ran numerous experiments (not shown) where for a graph size of 2000 nodes, we varied the following parameters: proportion of nodes of a particular type (u, ua, o, oa, and pc), number of layers for the user and object DAG (to include turning off this feature), and the mean number of edges per node. We chose graphs of 2000 nodes for this experiment because that was the maximum size at which all three algorithms had a less than 20 second execution time. Some of these parameter changes produced no significant effect on execution time (e.g., number of pc nodes) while others produced significant changes (e.g., those related to the number of edges in the graph). The number of edges in the graph was affected by two factors: the number of candidate edges and the p variable. The number of candidate edges was changed by varying the proportion of ua and oa nodes and the number of layers. The p variable used to calculate whether or not to instantiate a candidate edge was changed by varying the parameter for the mean number of edges per node. While we were able to change the execution times through parameter manipulation, the relative execution times between the three algorithms remained the same.

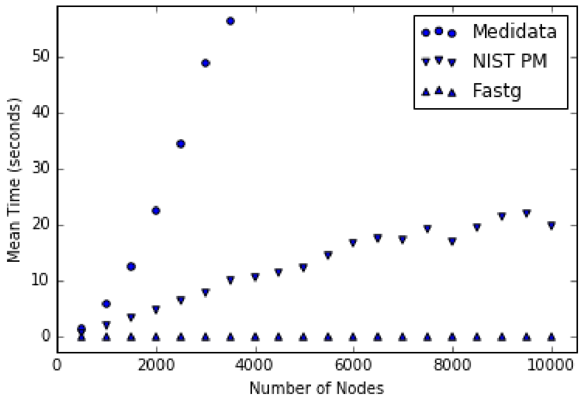

Figure 3: Execution time on graphs up to 10,000 nodes

In the figures, we refer to our algorithm as ‘Fastg’ and the other two as ‘Medidata’ and ‘NIST PM’. For each data point, we took the mean of 300 trials. We limited each algorithm to taking no more than 60 seconds, at which point we terminated further use of that algorithm. In an actual NGAC deployment, the required response time to show a user their accessible objects is more likely to be less than 2 seconds.

Figure 3 shows the timing for all three algorithms for graphs up to 10,000 nodes. At 10,000, the Fastg algorithm took a mean of .077 seconds to retrieve the set of objects available to a particular user. The NIST PM algorithm was 285 times slower, taking 22 seconds. The Medidata algorithm exceeded the 60 second limit at just 4000 nodes. Given that the required response time in an actual deployment is likely just a couple of seconds, the Medidata and NIST PM algorithms are limited to being used on graphs

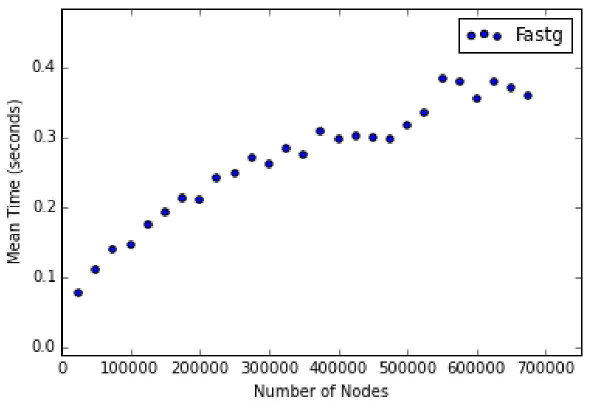

Figure 4: Execution time on graphs up to 700,000 nodes

with less than a couple of thousand nodes (that conform to our parameters).

Figure 4 shows the performance of the Fastg algorithm on graphs up to 700,000 nodes. The Fastg algorithm takes less than .4 seconds at 700,000 nodes. Given our assumed operation requirements of less than 2 seconds, this makes Fastg scalable up to the largest graphs that we produced (that conform to our parameters). We did not generate larger graphs due to execution time and memory limitations on our code used to produce the NGAC graphs.

Note, great care must be taken in interpreting these results. Our intention was to create as realistic graphs as possible and then show that the relative performance of the Fastg greatly outperformed that of the Medidata and NIST PM solutions. In this work, we have done that both theoretically and, in this section, empirically. However, NGAC graphs from operational deployments may have different parameter values or properties not modeled by our graph simulator. Such differences can greatly effect the absolute timing values (as we saw in our experiments on graph of 2000 nodes in changing the parameter values). Thus, we caution the reader to avoid using this work to calculate a precise upper bound on the size of graph that can be processed by any of the three algorithms. That said, the linear nature of Fastg should make it suitable for use on most any realistic NGAC graph.

4. ACCESS CONTROL VISUALIZATION

These graph algorithms enable access control decisions to be made while simultaneously instantiating multiple access control policies. However, a major question remaining is how to effectively communicate this set of privileges to the users. To this end we have designed an access control visualization approach that meets the following goals:

1. Leverage the access control graph to create a default visualization method for review of user file access

2. Abstract away the access control policy details such that the users (or administrators) do not need to understand the policies nor need to know which of the files are covered by which policies

Meeting these goals will enable efficient review of user privileges to best limit insider access to information (and thereby limit information leakage).

The NIST PM implementation, version 1.5, meets the first goal by leveraging the access control graph. This approach uses the PM itself as a root node in a file hierarchy and then the instantiated access control policies as the second level folders. Clicking on the access control policies enables the user to traverse the object DAG backwards (displaying only oa nodes pertaining to the chosen policy) until reaching the desired files. Unfortunately, this approach does not meet our second goal because their system requires users to navigate to their files by knowing which files are covered by which policies.

Our solution is to use the user's name as the root in a hierarchical file structure. The second level 'folders' are the labels for the border oa nodes reachable from the user's u node. Given the importance of the border edges in the NGAC access control definition 1, it is natural to use the border oa nodes as the first layer of file organization for the user. When a user clicks on an oa node name, the next level folders that appear are the oa node predecessors in the object DAG for which the user has some privilege. This graph traversal stops whenever the user reaches the object leaf nodes.

In our approach, we abstract away the complexity of the access control graph to make it appear to the user as if they are traversing the usual hierarchical directory structure used by default in all major operating system. In reality, the user is traversing possibly overlapping paths of the graph. The number of such paths is exponential and so we perform calculations only on the path actually being traversed by the user. Furthermore, there may be multiple ways for a user to access a particular file. This enables built in flexibility that previously had to be provided explicitly with artifacts such as symbolic links.

Figure 5 provides an example view of a user's accessible objects taken from one of our testing datasets covering a medical scenario. While it appears to be a typical file hierarchy, note how there are multiple paths by which to traverse to particular files (demonstrating that we are actually traversing a graph). For example, file 'mrec1' is available via three different paths in the graph: $root \rightarrow TS$, $root \rightarrow MedRecords$, $root \rightarrow alicehome \rightarrow AliceMedRecords$. In fact, all files shown in this visualization depict this multi-path behavior except for the files 'DAC uattrs rep' and 'alice home rep'.

4.1 Predecessor Node Visualization Algorithm

We now provide an efficient algorithm to determine what files and folders to show when a user clicks on some 'folder'. Initially, this will be one of the labels for the border oa nodes reachable from the user node, u_1. The algorithm is as follows:

1. Let the 'folder' on which the user clicks correspond to an oa node, x (note that this algorithm assumes that x is a folder that u_1 has the ability to view). Find the 'covered' pc nodes for x by performing a BFS and including all reachable pc nodes.

2. Find the set of predecessors of x and for each predecessor node, y, find the required pc nodes by performing a BFS and including all reachable pc nodes.

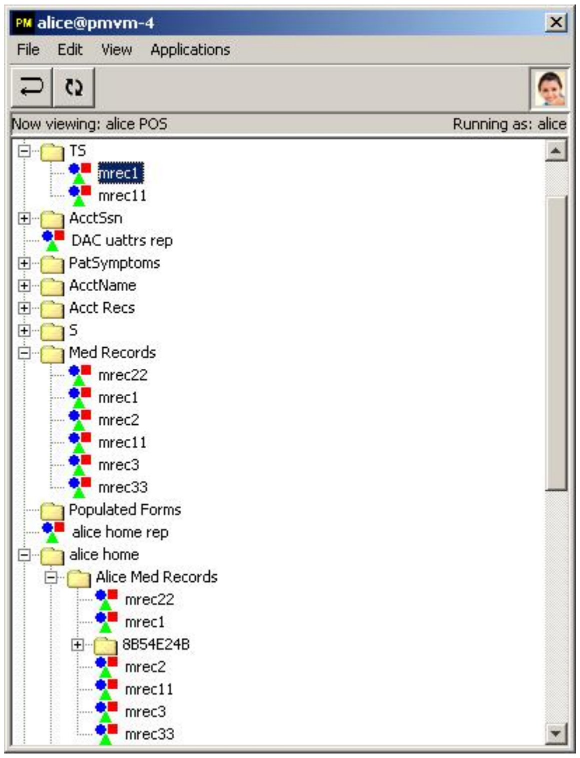

Figure 5: Example hierarchical visualization of a user access rights directed acyclic graph

3. For each predecessor node, y, if the set of required pc nodes is equal to the covered pc nodes for x then add it to a list of nodes available for display. If a node doesn't make it on this list in this step it doesn't meant that u_1 can't access it (simply we currently don't have enough evidence).

4. If there are predecessor nodes not on the list of nodes available for display, execute the 'Border oa Labeling' algorithm described below for u_1.

5. For each predecessor node, y, not on the available node list, perform a BFS from y to find all labeled border oa nodes (from the previous step). Let the 'available rights' for y be the union of the access right/pc node pairings from these reachable labeled border oa nodes. From this set of pairings, create a hash table where the keys are the access rights and the values the set of pc nodes. If there exists any key for which the values are a superset of the required pc nodes for y, then add y to the list of nodes available for display.

The 'Border oa Labeling' algorithm used above in step 4 is as follows:

1. BFS from u_1 to identify the set reachable ua border nodes. For this set of ua border nodes, let the set of 'active' edges be the ua$\rightarrow$oa outedges.

2. For each 'active' edge, label the oa head node with the ops edge label (eliminating duplicates). At this point, each reachable border oa node is labeled with a set of access rights.

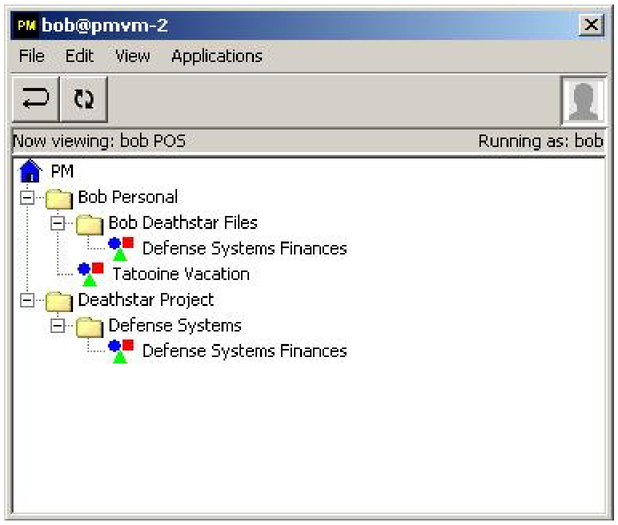

Figure 6: Example File Hierarchy for Access Control Graph in Figure 2

3. From each pc node, perform a backwards BFS (traversing edges backwards) to find labeled oa border nodes. For each such node, label it with the set of reachable pc nodes.

4. Each processed oa border node is then labeled with the cross product of the union of the access right labels with the set of reachable pc nodes (forming the access right/pc node pairings).

The combination of these two algorithms is linear, $O(n+m)$ (assuming as usual that the number of distinct access right types and policy classes are a small constant).

4.2 Visualization Examples

We now return to our example of the accountant Bob who is working on the defense systems finances for a deathstar. We will use our visualization approach to show the files available to Bob in Figure 2 using the node labels from Table 1.

The fully available hierarchical tree for user Bob is shown in Figure 6. This assumes that Bob has clicked on the 'Bob Personal' folder followed by a click on the 'Bob Deathstar Files' subfolder. It also assumes that Bob has clicked on the 'Deathstar Project' folder followed by a click on the 'Defense Systems' folder. These four clicks expand out visually all of Bob's available folders and files as shown in Figure 6. Note that the 'Technical Designs' folder and the 'Energy Shield' file are not visible because they are not accessible to user Bob.

A feature of this approach is that user Bob has access to the 'Deathstar Finances' file through both his own documents folder as well as the 'Deathstar Project' folder (logically this is because Bob is the owner/maintainer of that file). This again demonstrates the power of the approach where the user visually sees a hierarchy but can access the same files through multiple paths (without the need to explicitly create such linkages).

Note that while Bob has access to the 'Deathstar Project' folder, he is unable to see anything regarding the 'Technical Designs' folder including the 'Energy Shield' file. For Bob to

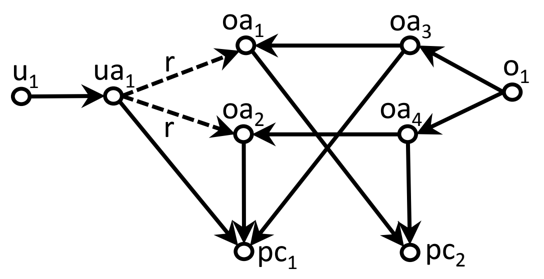

Figure 7: Minimal Access Control Graph Containing an Orphaned File

be able to access the 'Technical Designs' folder and 'Energy Shield' file, there would have to exist an edge $oa5 \rightarrow oa2$, $oa3 \rightarrow oa2$, $oa5 \rightarrow oa1$, or $oa3 \rightarrow oa1$ (see Figure 2). Alternately, the existence of an edge $o3 \rightarrow oa1$ or $o3 \rightarrow oa2$ would be sufficient to allow Bob access to the 'Energy Shield' file per Definition 1. However, for this our visualization approach would not allow Bob to use the 'Technical Designs' folder because it would still not be accessible to Bob. In this case, Bob could access the 'Energy Shield' file through the folder 'Bob Personal'. Thus, when a user can't get to one of their files through some particular oa node, there generally exists another oa node that will permit access through the visualization approach.

4.3 Orphan Files

However, there does exist the possibility that a user may not be able to traverse the visualization to reach a file that by Definition 1 is accessible. We call such files 'orphan' objects. None of the examples in the NGAC or the PM specification will generate orphans. Likewise, in our own test datasets we have never experienced an orphan file. Nevertheless, the possibility exists and so we discuss approaches to allowing for this eventuality.

For an orphan file to exist for a particular user, there must be an object node that is accessible but each path from the object to the set of reachable border oa nodes has a node that is not accessible. This happens when for each path, there exists an intermediate node that 'requires' a policy class not provided by the path's border oa node. An intermediate oa node requires a policy class when it has a path to that pc node (see Definition 1). Note that while the intermediate nodes on each path are not accessible, each path provides user privileges to the object such that the union of the received privileges enables the object to be accessible.

Figure 7 shows the simplest possible access control graph with an orphan file. o_1 is accessible because it receives pc_1 read privileges from oa_2 and pc_2 read privileges from oa_1. However, oa_3 is not accessible because it requires pc_1 privileges but only receives pc_2 privileges from oa_1. Likewise, oa_4 is not accessible because it requires pc_2 privileges but only receives pc_1 privileges from oa_2.

We have identified three different approaches to handling the possibility of orphan files such that the user can still find and access them (in order of increasing cost of computation time):

1. Enable the user to perform a search through all accessible files as a method to have access to any orphaned files. Our algorithm to find accessible objects (section 3.1) provides a list of all accessible files, both orphaned and available through the visualization approach. The user can simply perform a regular expression search on that list.

2. In the user's visualization of their file hierarchy, provide a folder at the second tier (alongside the reachable border oa node labels) that is labeled 'Orphan Files'. The orphan files can be detected when first launching the visualization and then listed in that directory. Our quadratic algorithm for finding orphan files is provided below.

3. Show the user orphaned files while they are traversing their hierarchical file structure. Whenever a non-accessible folder is encountered, perform a search for orphaned nodes only above the non-accessible folder. If orphans are encountered then show them in the current directory with a special designation to indicate that they are orphans. To the best of our knowledge, this approach requires executing our full quadratic 'find orphans' algorithm (below) followed by a reverse BFS to determine which orphans should map to the non-accessible folder.

4.4 Orphan Node Detection Algorithm

This algorithms enables one to detect orphan nodes for a user node u_1. The algorithm is as follows:

1. Execute the 'Border oa Labeling' algorithm to label the border oa nodes, reachable from u_1, with access right/pc node pairings (see section **??**).

2. Create a hash table where the keys will be node names and each value will be a set of access right/pc node pairings.

3. From each visible border oa node, x, BFS up (traversing the edges backwards) over the object DAG (i.e., don't traverse any ua$\rightarrow$oa edges). For each visited node, y, add it to the hash table (if it isn't already there). Add x's access right/pc node pairings to the value set for y.

4. For each key in the hash table, x, perform a BFS down (traversing edges forwards) to find the required pc nodes. If the value set for x does not contain some privilege for which all required pc nodes are covered, then delete this key from the hash table. In more detail, the value set must have access right/pc node pairings with some privilege, p, where the associated pc nodes in the pairings with p must be a superset of the required pc nodes. The resulting hash table will contain only nodes that are accessible to u.

5. For each key, x, in the reduced hash table that references an o node (not an oa node), BFS down (traversing edges forwards) attempting to reach a visible border oa node (as identified previously). However, modify the BFS to only traverse nodes that are referenced as keys in the reduced hash table. Nodes not in the hash table are either not accessible to u or will not provide a path to one of the visible border oa nodes. If the BFS terminates without reaching any visible border oa node, add x to a list of orphaned objects.

Steps 3 and 5 cause the algorithm to be quadratic. Steps 4 is described as being quadratic (for clarity) but can be made linear if one initiates the required BFSs from the pc nodes. The overall algorithm is quadratic.

5. RELATED WORK

In this section, we discuss the NGAC reference implementations and algorithms that preceded this work. There are two public NGAC/PM reference implementations; both available on GitHub [10]. NIST provides a reference implementation in Java that was the primary reference used in the development of the NGAC [3]. The company Medidata provides an implementation in Ruby that they use for their software products in the medical field [2]. A third GitHub policy machine implementation is available from Colorado State, but we will not reference it further as it focuses on using the NIST PM implementation to manage application-level operating system resources in Linux environments [1].

For the NIST implementation, we evaluated version 1.5. Their code related to determining which resources are available to a particular user is cubic, which explains the slow execution time even on small test sets. We provided our algorithms to NIST PM development team and they plan to use a variant on our access control algorithm as well as our visualization approach in an upcoming software release.

For the Medidata code base, we evaluated their default implementation in the file '\lib\policy_machine.rm' of version 1.1.0. Note that they have alternate implementations that we did not analyze that use a graph database and relational database (they do not recommend use of the graph database as they claim the 'interface is slow' and we didn't have access to their relational database to test it). For their default 'in memory' implementation, they have an $O(nm^2)$ cubic execution time method 'accessible_objects' that determines which files a user can access. Their method 'is_privilege', to determine if a user has a specific privilege on a particular object, is also quadratic while ours is linear. We provided them our algorithms and they plan to use them to improve their default implementation.

It appears that both implementations are inefficient due to a direct translation of the set theoretic NGAC notation into computer code.

6. CONCLUSION

The lack of an efficient system to simultaneously instantiate security policies has resulted in the use of blunt mechanisms to restrict user access to data (e.g., reliance on just DAC and isolated networks for differing levels of data sensitivity). This has resulted in insiders having access to more data than is necessary to perform their job function, exacerbating the impact of insiders leaking sensitive information. The NGAC provides a solution to this important problem by enabling the instantiation of multiple security policies within a single access control system. It quite appropriately provides requirements without specifying implementation details, allowing for competing approaches. However, the existing reference implementations use cubic algorithms, which raised into question whether or not NGAC can be implemented efficiently. Furthermore, NGAC did not provide guidance on how to visualize the results of the systems, making it unclear how perform reviews and audits of user access.

This work addressed both of these issues. We provide the first implementation of NGAC using an efficient linear time algorithm (bounded to the parts of the graph relevant to the user). Furthermore, we provide a novel visualization approach that works by default with multiple access control policies and that enables efficient review of user access rights.

7. REFERENCES

[1] Colorado state 'tinypm' implementation on github.
[2] Medidata policy machine code on github, version 1.1.0.
[3] Nist policy machine code on github, version 1.5.
[4] Organization for the advancement of structured information standards (OASIS).
[5] ANSI. American national standard for information technology, role-based access control (RBAC). Technical Report ANSI INCITS 359-2004, American National Standards Institute, 2004.
[6] ANSI. Information technology - next generation access control - functional architecture (NGAC-FA). Technical Report ANSI-INCITS 499-2013, American National Standard Institute, 2013.
[7] P. Biswas, R. Sandhu, and R. 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, pages 1–12, New York, NY, USA, 2016. ACM.
[8] B. Bollobas. *Random graphs.* Cambridge studies in advanced mathematics. Cambridge university press, Cambridge, New York (N. Y.), Melbourne, 2001.
[9] D. Ferraiolo, S. Gavrila, and W. Jansen. Policy machine: Features, architecture, and specification. Technical Report NISTIR 7987 Revision 1, National Institute of Standards and Technology, Oct. 2015.
[10] GitHub. Github code repository.
[11] V. Hu, D. Ferraiolo, and D. Kuhn. Assessment of Access Control Systems. Interagency report, National Institute of Standards and Technology (NIST), 2006.
[12] X. Jin, R. Krishnan, and R. Sandhu. *A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
[13] NCSC. *A Guide to Understanding Discretionary Access Control in Trusted Systems.* Number NCSC-TG-003. National Computer Security Center, Fort George G. Meade, Maryland, USA, 1 edition, Sept. 1987.
[14] OASIS. eXtensible access control markup language (XACML) Version 3.0., OASIS Standard, Jan. 2013.
[15] A. C. O'Connor and R. J. Loomis. 2010 Economic Analysis of Role-Based Access Control. Technical Report RTI Project Number 0211876, RTI International, 3040 Cornwallis Road Research Triangle Park, NC 27709, Dec. 2010.
[16] D. Servos and S. L. Osborn. *HGABAC: Towards a Formal Model of Hierarchical Attribute-Based Access Control*, pages 187–204. Springer International Publishing, Cham, 2015.
[17] F. Turkmen and B. Crispo. Performance evaluation of XACML PDP implementations. In *Proceedings of the 2008 ACM Workshop on Secure Web Services*, SWS '08, pages 37–44, New York, NY, USA, 2008. ACM.

[18] E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. In *IEEE International Conference on Web Services (ICWS'05)*, page 569, July 2005.