Linear Time Algorithms to Restrict Insider Access using Multi-Policy Access Control Systems

Peter Mell[1], James Shook[1], Richard Harang[2], and Serban Gavrila[1]

[1]National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899
[1]{*peter.mell, james.shook, serban.gavrila*}*@nist.gov*
[2]U.S. Army Research Laboratory, Adelphi, MD United States
[2]*rich.harang@gmail.com*

Abstract

An important way to limit malicious insiders from distributing sensitive information is to as tightly as possible limit their access to information. This has always been the goal of access control mechanisms, but individual approaches have been shown to be inadequate. Ensemble approaches of multiple methods instantiated simultaneously have been shown to more tightly restrict access, but approaches to do so have had limited scalability (resulting in exponential calculations in some cases). In this work, we take the Next Generation Access Control (NGAC) approach standardized by the American National Standards Institute (ANSI) and demonstrate its scalability. The existing publicly available reference implementations all use cubic algorithms and thus NGAC was widely viewed as not scalable. The primary NGAC reference implementation took, for example, several minutes to simply display the set of files accessible to a user on a moderately sized system. In our approach, we take these cubic algorithms and make them linear. We do this by reformulating the set theoretic approach of the NGAC standard into a graph theoretic approach and then apply standard graph algorithms. We thus can answer important access control decision questions (e.g., which files are available to a user and which users can access a file) using linear time graph algorithms. We also provide a default linear time mechanism to visualize and review user access rights for an ensemble of access control mechanisms. Our visualization 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. It also provide an implicit mechanism for symbolic linking that provides a powerful access capability. Our work thus provides the first efficient implementation of NGAC while enabling user privilege review through a novel visualization approach. This may help transition from concept to reality the idea of using ensembles of simultaneously instantiated access control methodologies, thereby limiting insider threat.

Keywords: ABAC; access control; algorithms; complexity; computer security; graph theory; insider; NIST; NGAC; Policy Machine; simultaneous instantiation; XaCML

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) [19]). However, many other access control approaches exist that provide enhanced features, especially for enterprise environments. This includes capabilities relevant to particular paradigms (e.g., Chinese Wall for conflict of interest [9] and Mandatory Access Control (MAC) for the handling of classified data [28]). Other policies provide greater simplicity in administering access control at scale (e.g., Role Based Access Control (RBAC) [1]). However, methods for protecting information with multiple models under one mechanism have been lacking, and this can result in enterprises settling for using a single simple model (e.g., DAC).

The result can be restrictions on insider access being defined very loosely; this increases the risk of insiders having unnecessary access to sensitive information and sharing that information outside of the organization. To ensure that users do not inappropriately share data, enterprises may then resort to the costly and inefficient approaches of separating different data types (e.g., military classification levels) into totally distinct and isolated networks or administering multiple independent access control systems over the same data sets. They may try to implement multiple independent access control systems on each host to enforce multiple policies, but this technique has management and scalability issues (although it is used for high security operating systems such as SELinux [16]). Alternately, they may accept the risk of data being leaked, which can have disastrous results (e.g., classified documents being made public).

What is needed then is a single access control mechanism that enables an enterprise to simultaneously instantiate multiple policies (e.g., DAC and MAC) to limit user access to unneeded resources. Furthermore, this mechanism must be able to efficiently provide access control decisions and also enable administrators/auditors to review the access privileges on a per user/attribute basis. This ability to both make decisions and provide review are necessarily features that must be scalable to accommodate the access control policies of large enterprises.

The American National Standards Institute (ANSI) has addressed this need by standardizing an access control approach, Next Generation Access Control (NGAC) [2, 3]. The NGAC stems from and is in alignment with the Policy Machine (PM) [11], a research effort by the National Institute of Standards and Technology (NIST) to develop a general purpose Attribute Based Access Control (ABAC) framework [13]. The NGAC is designed to enable simultaneous instantiation of multiple access control policies within a single access control mechanism, enabling both unified access control decisions and also review of user access capabilities. Examples for how to use NGAC to enforce the DAC, MAC, RBAC, and Chinese Wall policies are available from [10]. The NGAC 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. While appropriate, this leaves open the important question as to whether or not NGAC is scalable. In this work, we find that the existing reference implementations are inefficient (using cubic algorithms) which indicates that NGAC may not be scalable. The primary NGAC reference model version 1.5 [20] could only scale to a test model of a couple of hundred nodes, at which point it took several minutes to visualize the set of objects available to just one user. Answering this scalability question for NGAC is critical because NGAC is the only access control methodology available promising both efficient decision and review for simultaneous instantiation of multiple access control policies.

The only other multi-policy access control methodology available is the current market leader, the eXtensible Access Control Markup Language (XACML) standard [21] from OASIS [22]. Other related logic-based policy ABAC models (that either have no reference implementation or are not multi-policy) include ABAC$_\alpha$ [13], HGABAC [26], and ABAC for Web Services [29]. XACML has been shown empirically to lack scalability in [27] where 3 different XACML implementations all experienced performance problems in making access control decisions where the performance decreased as the number of policies increased (each policy in XACML contains the access rules for a set of target objects). In addition, all of these logic-based policy ABAC models have been shown to be NP-complete with respect to simply determining the access attributes needed by a user to access a particular resource [7] (mapping to the satisfiability problem). Thus, these methodologies do not meet our stated need for a multi-policy system that provides for both efficient decision and review. They are then undesirable for large enterprise systems with respect to ensuring the restrictions on insider access to sensitive data to avoid information leakage by insiders.

In this work we prove that NGAC is scalable by providing linear time algorithms for both access control decisions and review of user access rights. To provide efficient decision capabilities, 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 solution in general was to use breadth first search (BFS) and depth first search (DFS) variants that perform 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 amortized cost of the multiple searches can be shown to be linear, resulting in a linear time complexity algorithm. 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.

With respect to review of user privileges, the NGAC standard does not provide any guidance on visualizing access control results. 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 is not ideal in a system that simultaneously implements multiple policies (which is the whole point of NGAC). For example, the current NIST NGAC implementation [20] 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 should be efficient and would ideally be automatically generated from the existing access control graph to avoid additional and excessive administrative burden.

To meet this need for review capabilities, 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. Our algorithm for review is based off of our algorithm for decision capabilities and thus is of linear time complexity.

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 novel visualization approach to enable review of user object access on NGAC systems, and
3. linear time algorithms for performing both access control decisions and review of user access rights.

These contributions make great strides towards enabling large scale deployments of NGAC, thus offering enterprises a powerful new capability by which to efficiently limit insider access to information (and thereby limit information leakage).

This paper is an extended version of a shorter paper [18] published in the 8th Association for Computing Machinery (ACM) Internation Workshop on Managing Insider Security Threats. This workshop was held as part of the ACM Conference on Computer and Communications Security in October of 2016 in Vienna, Austria. This extended version provides the following enhancements:

1. a new access control decision algorithm to determine which users have access to particular objects
2. memory profiling results for the primary access control algorithm as well as for the size of the access control graphs themselves
3. performance results on graphs of up to 2 000 000 nodes (formerly our graphs were at most 700 000 nodes due to processor and memory limitations)
4. example NGAC reference implementation visualization with a comparison against our new visualization approach
5. discussion of the complementary nature of popular access control policies
6. discussion of needed future work and possible research pitfalls

The remainder of this paper is structured as follows. Section 2 discusses background material while 3 discusses related work. Section 4 provides an overview of access control graphs within NGAC and provides a definition of when a user is allowed to access an object. Section 5 presents our access control algorithms and section 6 presents our visualization approach. Section 7 discusses future research ideas and their related challenges. Section 8 concludes.

2 Background

In this section, we provide a short description of some of the most popular access control policies. We focus on demonstrating their different strengths to support our argument that an ensemble of methods can most tightly restrict insider access to information. We then discuss the need to simultaneously instantiate multiple policies within a single access control system as opposed to having multiple independent systems running simultaneously.

2.1 Common Access Control Policies

Begining with Butler Lampson's protection access matrix in 1969 [14, 15] basic research in access control models began to build steam. Researchers Bell, Lapuda and Biba's pioneering work can be found in [4, 6]. Since then many access control models were created and examined [24, 25]. Each model was designed to either improve existing models or tailored to a specific use case. Some well know examples of access control models are The Chinese Wall, Discretionary Access Control, Mandatory Access Control, and Role-Based Access Control. We briefly present some of them below.

Rule based access control is one of the basic types. Simply, there is a list of rules that are checked when deciding if a user has access. For instance a firewall contains a list of rules that an incoming data packet would be checked against.

Often when someone says they own an object it is interpreted that they have discretionary access to an object [19]. Discretionary Access Control says that a user with certain access rights to an object can grant that access to others. Although this model is convenient, it unfortunately can result in leakage of access. For example, if Bob gives Todd read access to an object then Todd may allow Rob to read the object.

One remediation to access leakage within DAC models is to introduce mandatory controls. Within a Mandatory Access Control model every user and object receives a label. Those labels form a partial order that sets mandatory access rights. Assignment of these labels are mandatory and ordinary users cannot change them.

An example of a model that can implement both DAC and MAC policies (but not at the same time) is the Bell-Lapadula model [4]. It focused on confidentiality and does not let lower labels “read-up” and higher levels “write-down.” This model was designed for defense systems that try to balance the need of confidentiality with accessibility.

The Chinese Wall was designed to address conflict-of-interest issues with respect to information flow. For example, an investment firm with two competing companies as clients should not let their analysts work with both clients.

Many of access control models mentioned in this section could be called lattice based [25]. This is where every user and object has a security label such that those labels are assigned a partial-order that indicates allowed information flow. Clearly, MAC and The Chinese wall are examples of this, and there are many others. Due to the relationship based nature of NGAC, it can be shown that most latticed based models can be efficiently modeled using NGAC. For instance, with the aid of prohibitions, MAC can be modeled within NGAC. Furthermore, it was highlighted within the NGAC standard [3] that The Chinese Wall can be implemented with NGAC.

In the mid 1990s Role Based Access Control started to gain steam. RBAC became one the most widely used models since MAC and DAC. In RBAC users take or are assigned roles. The users then inherit from those roles privileges. RBAC allows role definitions to be easily modified. Thus, policies become easier to manage. RBAC has all the advantages of MAC and DAC since those models can be implemented within RBAC. After quickly becoming a well supported standard, RBAC has saved corporations and other institions large sums of money. A study in 2002 [23] showed that RBAC could save U.S. organizations millions of dollars per year.

Content based access control (CBAC) is a more recent model that focuses on the content of objects to make access decisions. Introduced in [30], CBAC is expected to be deployed on top of RBAC or ABAC as a complement to traditional access control models. CBAC could potentially be used to address online privacy concerns as well. For example, it is possible to use natural language processing to scan a social media posts for private data such as phone numbers and location to prevent public postings. It can also be used for more advanced scanning such as notifying the people in posted photos. However, this type of CBAC implementation does not prohibit disallowed behavior and should not be used this way if strong security is needed.

One of the many advantages of ABAC, in particular NGAC, is that they can handle multiple access control policies. That is RBAC, rule based models, lattice based models, parts of content based models, and other access control models can be simultaneously implemented within a NGAC instantiation. As just discussed, each access control model has different strengths. Thus we argue that an ensemble of approaches is best to most tightly restrict user access.

2.2 The Need for Simultaneous Instantiation

Unfortunately, no efficient mechanisms has existed to simultaneously instantiate multiple policies. Such a system would create an ensemble approach that takes advantage of each policies unique contribution. As a result, organizations typically use just a single access control policy (usually DAC).

Some more secure operating systems such as SELinux [16] enable both DAC and MAC simultaneously. Inside of the SELinux operating system their can be two independent access control mechanisms operating at the same time. This not only consumes excess resources on the local machine but requires administrators to maintain two separate sets of access control rules. What is desired is to enable administrators to administer a single set of rules that cover all instantiated policies simultaneously.

Furthermore, there is a desire for a single such ensemble system to work at the network level, controlling resources at an enterprise scale. Maintaining multiple separate system, like with SELinux, would be even more difficult to maintain at an enterprise scale thus furthering the argument for a single system to maintain multiple access control policies.

Finally, to our knowledge it is extremely uncommon for an organization to simultaneously enforce multiple access control policies (even in military settings). This provides further evidence that such approaches are too costly to maintain. However, recent major U.S. based leaks have demonstrated the need to more tightly limit insider access.

3 Related Work

Of the two existing multi-policy access control methodologies, XACML is the oldest with the first version having been published in 2003. Compared to the relatively young NGAC standard (published in 2013), there exist many more implementations and it has achieved much greater adoption. Perhaps this is because XACML was available first and, up to this point, there has been a lack of compelling evidence to convince the community to use NGAC. In addition, there has been the unanswered but critical question of whether or not NGAC can scale.

Likely for these reasons, there exist only two publicly available NGAC/PM reference implementations; both are available on GitHub [12]. NIST provides a reference implementation in Java that was the primary reference used in the development of the NGAC [20]. The company Medidata provides an implementation in Ruby that they use for their software products in the medical field [17]. A third GitHub policy machine implementation is available from Colorado State University, 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 [5].

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 linear time algorithms to NIST PM development team and they upgraded to using our PReview access control algorithm in their version 1.6 (released on Github in November of 2016). They also implemented our visualization approach in this version as well.

For the Medidata code base version 1.1.0, we evaluated their default implementation[1] in the file '\lib\policy_machine.rm'. In this, they have a method 'accessible_objects' that determines which files a user can access. Our analysis shows this algorithm to have an $O(nm^2)$ cubic execution time. Their method 'is_privilege', to determine if a user has a specific privilege for a particular object, is also quadratic. Our algorithm is linear for both these operations. We provided them our algorithm and they plan to use it 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.

4 Access Control Graph Overview

Using the NGAC specification [3] 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 attribute (ua), object attribute (oa), and policy class (pc). All edges are directed. Per specification all object nodes are object attribute nodes, but there could be an object attribute that is not an object. Object attribute nodes may have edges to oa and pc nodes. However, no object attribute node may point to object nodes. User nodes are sources with edges to ua nodes. User attribute nodes may have edges to ua, oa or pc nodes [2]. Policy class nodes are sinks. Cycles and self-loops are prohibited. User attribute to Object attributes edges are labeled with a

[1]They have other implementations that make calls out to various databases.

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

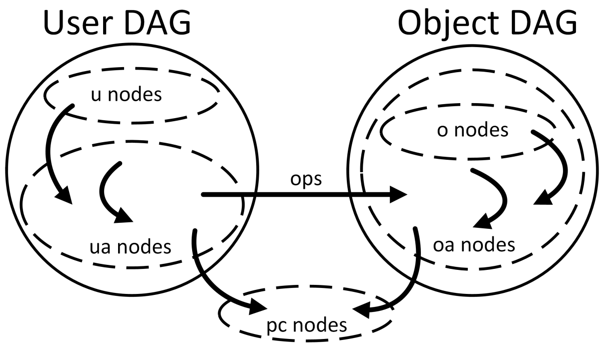

Figure 1: Diagram showing allowed edge relationships between the five different sets of NGAC node types.

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 cardinality of the ops set 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[3]:

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.*

This definition has three data collection components:

1. Identify the 'active' ua to oa bridge edges. This is the set of bridge edges that have label op_1 and where the tail is reachable from u_1 and the head is reachable from o_1. These active edges are the ones that enable u_1 to possibly have privilege op_1 on on o_1.

[3]We do not 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.

2. Determine the set of pc nodes reachable from the head of each active bridge edge. The union of such pc nodes is the set of ‘covered’ policy classes for op_1.

3. Determine the set of pc nodes reachable from o_1. These are the policy classes that are ‘required’ to be covered.

For u_1 to perform op_1 on o_1, the set of covered policy classes must be a, not necessarily proper, superset of the set of required policy classes.

Figure 2 shows an example NGAC access control graph which we will evaluate using Definition 1[4]. 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. We now discuss this in detail:

- o_1 requires pc_2 because there is a path connecting the two. This requirement is covered 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 covered 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.

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 interpret the nodes as shown next to Figure 2. Note how the user attribute nodes represent ‘teams’ to which Bob belongs (his own team[5] and the death star personnel team). The object attribute nodes provide a kind of hierarchy for different

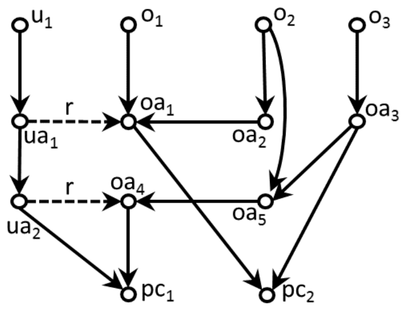

Figure 2: Example NGAC access control graph.

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’

[4]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.

[5]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.

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).

5 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?
2. What is the set of accessible objects for a user, u_1, and what operations can u_1 perform on each object?
3. What is the set of users that can access an object, o_1, and what operations can those users perform on o_1?

The first two determinations can be made through a slight variation on the same algorithm, which we refer to as PReview. Furthermore, PReview forms a basis for performing policy review in general and, to be discussed later, for our visualization approach. This generalization enables us, for the third determination, to write an algorithm similiar to PReview but that flows information in the opposite direction (from the object DAG to the user DAG). With the 'backwards' PReview algorithm we can identify all users that can access a given object using a particular operation in linear time complexity (discussed further below).

5.1 The PReview Algorithm

The PReview 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 from reachable border ua nodes). 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 with cardinality 0 or 1).

The core of the algorithm is then to iterate over each object of interest and use the object DAG to determine which are accessible and what ops are available to the user. Each time we process an object, we will perform a topological sort DFS to collect data for the object. However, we will reuse information such that the amortized cost of all the DFSs is linear. Each DFS from an object of interest determines three data sets for that object: 1) a set of candidate ops, 2) the set of 'covered' policy classes for each distinct op, and 3) the set of 'required' policy classes for the object. With these three sets, Definition 1 can be applied to determine which of the candidate ops can be used by the user on the object. In short, an op is available to the user on the object if the set of covered policy classes for that op is a superset of the set of required policy classes for that object.

In more detail, the algorithm is as follows:

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

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 recursive topological sort DFS to find the reachable pc nodes. However, when performing each DFS, label all processed nodes with the information found (the set of reachable pc nodes) 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 the 'required' pc nodes for each object.

7. While performing the 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. For example, use a dictionary where the key is the op and the value is the set of reachable pc nodes. Then use the normal functionality of the DFS to propagate these dictionaries up to the root of the tree (one of the objects of interest). When a node is being processed where multiple of its successors have these dictionaries, then union the dictionaries by taking the union of all keys; for the values take the union of the values for each key[6].

8. For each object of interest, compare the set of required policy classes against the covered policy classed for each key from the dictionary propagated to the object in the preceding step. If for some op_1, the set of covered policy classes is a superset of the set of required policy classes for that object, then the user can use privilege op_1 on the object.

We now look at the algorithm complexity. Steps 1 and 2 can be implemented to at most perform a single traversal of each ua node, edge in the user DAG, and each bridge edge. Steps 3 and 4 traverse each object DAG edge at most once. Step 5 takes constant processes each o node at most once. 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 (and touch each oa node at most three times). In summation, each nod and edge in the graph is then guaranteed to be touched/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($n+m$).

5.2 Empirical Algorithm Results

In this section, we evaluate the scalability of our PReview algorithm versus the two available reference implementations (NIST PM and Medidata). For our experimental platform we used an Oracle VirtualBox Ubuntu virtual machine with two cores and 10GB 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 was 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.

[6]For example, if one successor has key/value pair $op_1 \rightarrow (pc_1)$ and another successor has key/value pairs $op_1 \rightarrow (pc_2)$ and $op_2 \rightarrow (pc_3)$, then the union of the dictionaries would be key/value pairs $op_1 \rightarrow (pc_1, pc_2)$ and $op_2 \rightarrow (pc_3)$.

For our empirical scalability study, we used the PReview variant that computes the set of accessible objects for a particular user and the set of operations available for each accessible object. For comparitive purposes, we coded up the analogous algorithms from both NGAC reference implementations (the NIST PM [20] and Medidata [17]) using the same language and libraries. These implementations are discussed in section 3. 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 2 000 000 nodes. We used the number of nodes, n, as the independent variable and then scaled all other graph features relative to n. For each node type (u, ua o, ou, and pc) we use the following node proportions.

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

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 numbered labels (edges within a group were not allowed). A similar operation was performed for the object DAG.

There does 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 20s 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 target 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.

In the figures, we refer to our algorithm as PReview and the other two as 'Medidata' and 'NIST PM'. We also provide results for an algorithm 'PReview-nonrc' which is simply a non-recursive version of PReview. For each data point, we took the mean of 300 trials. We limited each algorithm to taking no more than 60s, 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 2s.

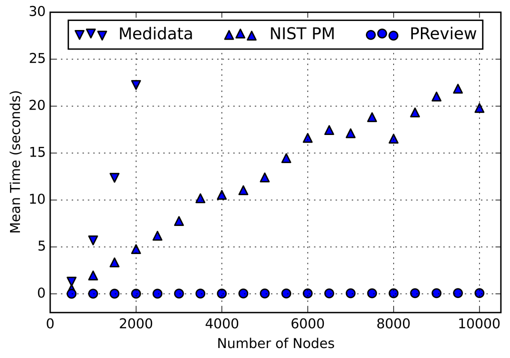

Figure 3: Execution time on graphs up to 10 000 nodes.

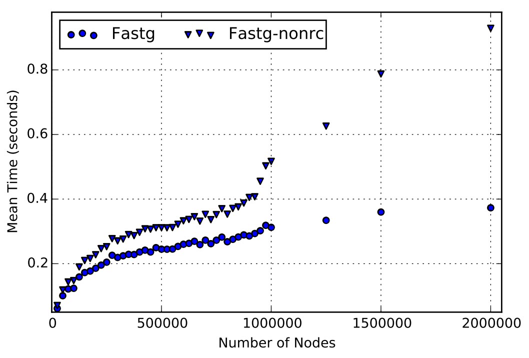

Figure 4: Execution time on graphs up to 2 000 000 nodes.

Figure 3 shows the timing for all three primary algorithms for graphs up to 10 000 nodes. At 10 000, the PReview algorithm took a mean of 0.077s to retrieve the set of objects available to a particular user. The NIST PM algorithm was 285 times slower, taking 22s. The Medidata algorithm exceeded the 60s 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 with less 2000 (that conform to our parameters).

Figure 4 shows the performance of the PReview-nonrc and PReview algorithms on graphs up to 2 000 000 nodes. The PReview algorithm takes on average 0.373s at 2 000 000 nodes and appears to have sub-linear curve. The non-recursive version takes 0.929s at 2 000 000 nodes which is 2.5 times longer. This difference may simply be due to the fact that we found better code optimization for the recursive code (theoretically the two should have similar runtimes). The non-recursive version can be used in situations where recursion is not desired or for graphs where the recursion depth might be an issue (some programming language limit recursion depth).

Given our assumed operation requirements of less than 2s, this makes PReview 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 PReview 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 affect 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 PReview should make it suitable for use on any realistic NGAC graph.

5.3 Algorithm to Determine Which Users Can Access a Particular Object

We now discuss how to modify the PReview algorithm to execute in ‘reverse’, although the actual algorithm is more complicated than a clean inverse. It is more accurate to say that this new algorithm recomposes and slightly modifies pieces of the PReview algorithm. This additional algorithm enables us to answer our third determination from section 5: ‘What is the set of users that can access an object, o_1, and what operations can those users perform on o_1?’.

To perform PReview in reverse we start at an object, o_1 and label it our 'object of interest'. We perform a topological sort DFS from o_1 to find the set of required pc nodes. At the same time, we identify the set of reachable oa border nodes. Each reachable oa border node is labeled with the pc nodes reachable from that node. This is very similar to steps 6 and 7 of the PReview algorithm. However, our goal this time is to label all oa border nodes (in linear time).

We now know the set of required pc nodes for o_1. What we need to determine is, for each user, the set of ops the have on o_1 paired with the covered pc nodes for those ops. With that information, we can use the same logic in PReview step 8 to determine if the user can access o_1 and with what ops. To determine the set of ops paired with the covered pc nodes, we could simply BFS from each user to find the reachable oa border nodes that have been labeled with this information. However, that would create a quadratic algorithm. Thus, we once again use a topological sort DFS starting from the user nodes to propagate up this information from the oa border nodes. During execution of each topological sort DFS, all visited ua nodes are labeled with this information so that it can be reused in subsequent topological sort DFSs starting at different users (similiar to step 7 of PReview). Note that since we label ua nodes only once and then reuse that information between topological sort DFSs (one per user), the amortized cost for the time complexity is linear as with the PReview algorithm.

In more detail, the algorithm is as follows:

1. Perform a topological sort DFS from o_1 to identify all oa border nodes (those with a back edge to at least one ua node) and label them with the pc nodes reachable from the labeled node.

2. Using the same topological sort DFS from the previous step, find the set of required pc nodes for o_1 (the pc nodes reachable from o_1).

3. For each user, perform a recursive topological sort DFS to find the reachable oa border nodes. However, when performing each DFS, label all processed ua nodes with the information found. However, we need to discuss how to create the information that is then propagated through the graph. When traversing an ops edge (those from a ua to oa node) they will be labeled with a set of ops. The associated oa node will be labeled (from our previous steps) with a set of reachable pc nodes. Together this forms a privilege pairing where each operation in ops is 'covered' by the set of pc nodes from the label on the oa node.

4. Each user node is now labeled with a set of operations and the covered policy classes for each operation. Since we know the from step 2 the required pc nodes for o_1, we can use PReview step 8 individually for each user to determine the privileges each user has on o_1.

5.4 Memory Analysis of the PReview Algorithm

We now evaluate the memory usage of the core PReview algorithm. This is necessary as in algorithm design, it is often possible to lower execution time through 'tricks' that consume large amounts of memory. At an extreme, an algorithm that achieved a linear execution time through using exponential amounts of memory would not be useful. Below we provide results showing that PReview uses a modest amount of memory. We also show that access control graphs on which PReview was executed also consume a modest amount of memory (although much more than the PReview algorithm). Figure 5 shows the memory usage of the PReview algorithm as the number of nodes in the generated access control graph varied from 25 000 to 2 000 000. The growth rate empirically appears to be almost sub-linear.

Figure 6 shows the maximum memory usage out of the 50 trials for each data point for the same experiment. A linear trend is observable here although the algorithm will randomly use far less memory for certain data points, creating the periodic deep troughs. This is reflective of the fact that some users had simpler relevant subgraphs enabling the PReview algorithm to use less memory.

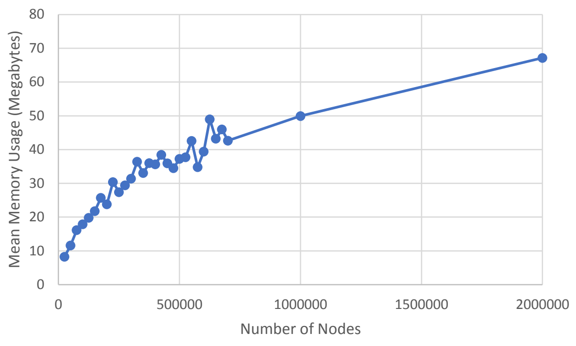

Figure 5: Mean memory usage of the PReview algorithm on graphs up to 2 000 000 nodes with 50 trials per data point.

Lastly, we show the memory consumed by the access control graphs themselves in Figure 7. Note that the graphs themselves consume much more memory than the PReview algorithm run on those graphs. At 25 000 nodes, the graphs use 88% of the total memory consumed by PReview operating on the graphs. At 700 000 nodes, the graphs use 98% of the total memory consumed. This indicates that as the graph size grows, the relative consumption of memory by PReview becomes negligible, which is remarkable given that PReview is also a linear execution time algorithm.

The modest linear growth rate of the memory consumption of the access control graphs means that a commodity laptop with 16GB of memory could handle graphs up to around 6.7×10^6 nodes. PReview is more likely to be run on large servers with much more memory and thus could handle appreciably larger graphs.

6 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 and enable administrator review of a user's privileges. To this end we have designed an access control visualization approach that meets the following goals:

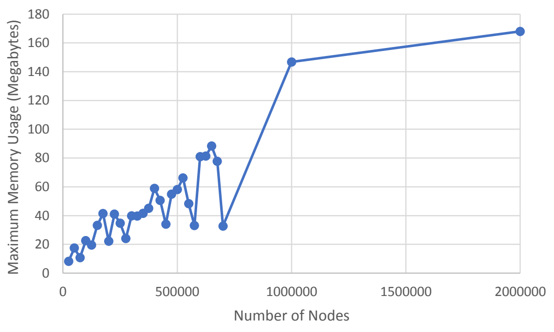

Figure 6: Maximum memory usage of the PReview algorithm on graphs up to 2 000 000 nodes with 50 trials per data point.

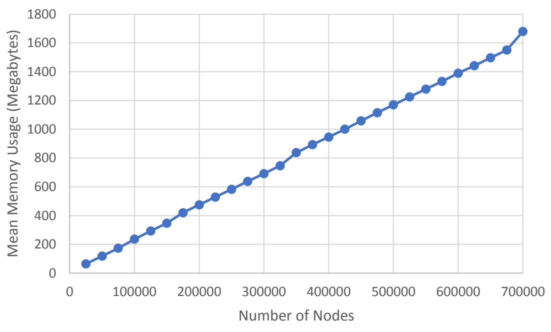

Figure 7: Mean memory consumed by the generated access control graphs up to 700 000 nodes with 50 trials per data point.

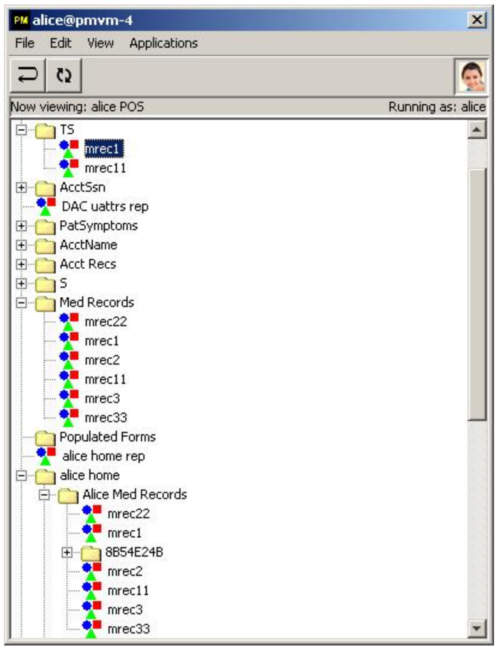

Figure 8: Example hierarchical visualization of a user access rights directed acyclic graph.

1. Leverage the access control graph to create a default visualization method for review of user file access (this avoids administrators having to separately maintain some sort of file hierarchy).

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.

A visualization approach meeting these goals will provides a default mechanism to enable efficient review of user privileges.

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 one to navigate to the user's 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 when the user reaches object 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 8 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: $root \rightarrow TS$, $root \rightarrow MedRecords$, and $root \rightarrow alicehome \rightarrow AliceMedRecords$. In fact, all files shown in this visualization depict this multi-path behavior except for the files 'DAC uattrs rep' and 'alice home rep'.

6.1 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 (which the user can always view by definition). 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 node that u_1 has the ability to view).

2. Execute a variant of PReview where in step 4, the algorithm stores all visited nodes instead of just objects when performing the backwards BFS. Thus, we have identified a set of nodes of 'interest' that includes both o and oa nodes.

3. In step 5 of the PReview algorithm, use the set of predecessors of x as the input nodes and intersect this set with the set of nodes of interest (from the previous step in PReview) to create a new set of nodes of interest. This is analogous to what already occurs in step 4 except that instead of using a single object as input we are using x's set of predecessor nodes.

4. PReview will return whether or not each predecessor of x is visible to u_1. Display those predecessors that are visible.

This algorithm is a simple variant on PReview and thus is linear, $O(n+m)$ (assuming as usual that the number of distinct access right types and policy classes are a small constant).

6.2 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 corresponding node labels.

The fully available hierarchical tree for user Bob is shown in Figure 9. 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 9. 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 be able to access the 'Technical Designs' sub-folder and 'Energy Shield' file, there would have to exist an edge $oa5 \rightarrow oa2$,

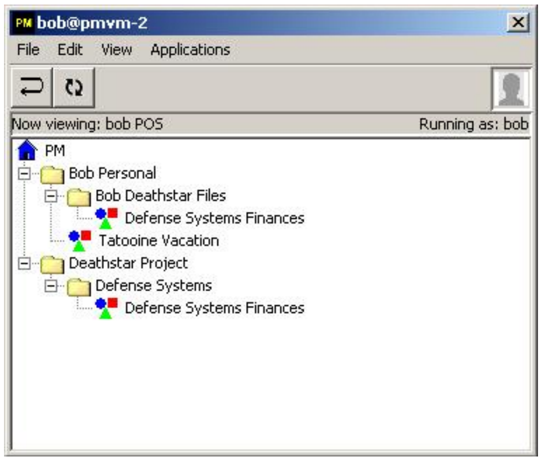

Figure 9: Example File Hierarchy for Access Control Graph in Figure 2.

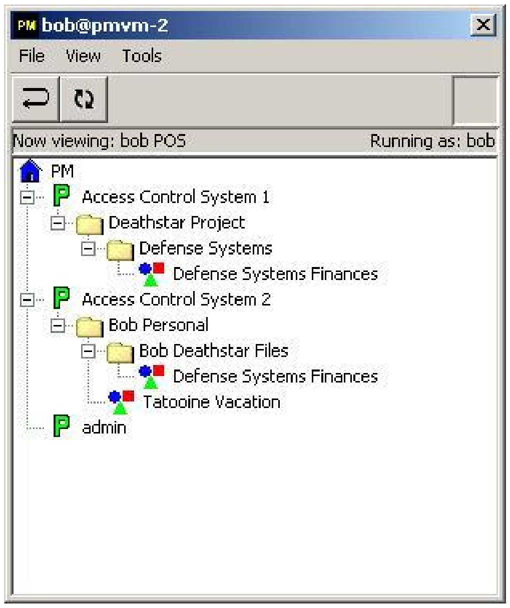

Figure 10: NIST PM File Hierarchy for Access Control Graph in Figure 2.

$oa3 \rightarrow oa2$, $oa5 \rightarrow oa1$, 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 cannot 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.

Figure 10 shows how the same example appears using the NIST PM visualization interface. The NIST PM also creates a hierarchical tree structure for the user to access their files or for an administrator to perform a policy review (shown in Figure /refbob-old). However, they root each user’s tree at a node called “PM”. The second layer of the hierarchy is the set of policy classes available as represented by the set of pc nodes. Then the subsequent “folders” are the oa nodes reachable from each respective pc node found through performing a backwards BFS. The leaf nodes then are the objects.

This approach has the same implicit symbolic link type advantages as our approach. Thus, we cannot claim to have invented this feature but, to our knowledge, this is the first time it has been identified as a useful feature. However, it a distinct disadvantages relative to our approach. The user must have some knowledge of the access control policies being implemented and must know which of their files are covered by which access control policies. Otherwise, the user might search down the hierarchical tree of one access control policy only to discover that the file is only reachable through a different hierarchy.

Furthermore, the implementation of the NIST PM approach version 1.5 used cubic algorithms and pre-computed the entire tree from the PM root not as opposed to our on demand approach. They did not computer whether or not the user can access a particular oa node, only the final leaf nodes (the files). This can cause the user to explore paths in their tree that would not lead to the files they are allowed to access. The resulting implementation takes excessive execution times on only small example graphs (e.g., several minutes).

Because of this, the NIST PM implementation team has adopted our approach in version 1.6 of their reference implementation. The example hierarchy of our visualization approach in Figure 9 was generated from a development version of the revised NIST PM version 1.6. Since this code was still

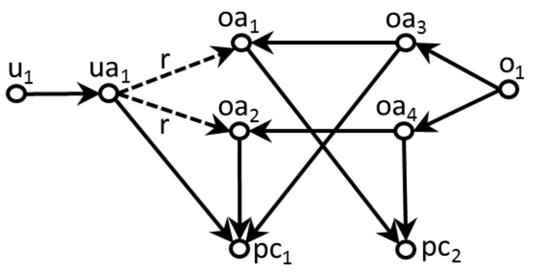

Figure 11: Minimal Access Control Graph Containing an Orphaned File.

under development by the implementation team, this explains why the root node in Figure 9 is still labeled “PM” when according to our approach it should be labeled “Bob”.

6.3 Orphan Files

One issue that we must discuss is that there does exist the possibility that a user may not be able to traverse the our 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 data sets we have never experienced an orphan file. Nevertheless, the possibility exists and so we discuss approaches to address 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 (or any other path from that node to a reachable border oa node). Note that an intermediate oa node requires a policy class when it has a path to the corresponding pc node (see Definition 1). While the intermediate nodes on each path are not accessible, each path still provides the user privileges to the object such that the union of the received privileges enables the object to be accessible.

Figure 11 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:

1. Enable the user to perform a search through all accessible files as a method to have access to any orphaned files. Our PReview algorithm from section 5.1 can 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 linear time 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. This can be accomplished through using our linear time algorithm to find orphans (below) in combination with a reverse BFS from the non-accessible folder.

Each of these three approaches can be used independently or together simultaneously.

6.4 Orphan Node Detection Algorithm

This algorithm, based on PReview, enables one to detect orphan nodes for a user node u_1 in linear time. The algorithm is as follows:

1. Execute a variant of PReview where in step 4, the algorithm stores all visited nodes instead of just objects when performing the backwards BFS. Thus, we have identified a set of nodes of 'interest' (including both o and oa nodes).
2. PReview will return the o and oa nodes that are accessible.
3. Analogous to steps 1-4 in the PReview algorithm, perform a backwards BFS from the set of reachable border oa nodes. However, this time only visit a node if it is accessible (determined in the previous step). Record this set of visible nodes.
4. The set of accessible nodes (from step 3) that are not visible (from step 4) is the set of orphan nodes.

7 Future Work and Challenges

One major challenge to NGAC is to keep the size of the graph manageable. There could be a large number of attributes assigned to many different users, and without adequate checks it is possible for local decisions to create attributes that are equivalent, but semantically the same. Thus, causing a node or attribute "blow-up." For instance, one administrator could create a formal attribute for a project name while the other could use an abbreviation or alias. Equally likely the NGAC graph could have an edge "blow-up." There could be both a directed path $ua_1 \to ua_2 \to ua_3$ and the assignment $ua_1 \to ua_3$ within the graph. However, the assignment $ua_1 \to ua_3$ is redundant and can be safely removed. With good design and clever logical reductions, our initial research indicates that the size of a NGAC graph can be kept minimal.

One of the advantages of ABAC is that it allows denial or prohibition of a privilege. Currently, in existing NGAC implementations a proposed privilege is checked against a list of prohibitions before a decision is made. Although not immediately clear it is possible to model prohibitions within NGAC in an efficient manner so that a list of prohibitions is not needed. For instance, user Smith, an IRS auditor, maybe able to alter tax returns however he should be prohibited from altering his own tax return. We will explore how to efficiently implement prohibitions with NGAC in future work.

Even if the size of the graph becomes large, we presented work in this paper that some basic policy review can be done in linear time. In its current form, PReview focuses on what objects a user has access to or what users have access to an object. However, a fully implemented NGAC model has more complex policy review needs. Such as incorporating prohibitions in the access decisions. With careful NGAC design we can adapt PReview, without increasing algorithmic complexity, to consider prohibitions along with other complex policy review.

In future work we will discuss NGAC graph management, a treatment of prohibitions, and a more general PReview algorithm.

8 Conclusion

It is vital to limit insider access to only the data necessary to perform their work in order to limit possible leakage of sensitive data to outside entities. To best constrain this access, we need a scalable access control methodology that supports protection of objects under the simultaneous instantiation of multiple access control policies (e.g., DAC and MAC).

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 the serious question as to whether or not NGAC is scalable. Furthermore, NGAC did not provide guidance on how to visualize the results of the systems, making it unclear how perform review of user access privileges.

This work addresses both of these issues. In [7] the authors pointed out that performing policy review on ABAC models, such as NGAC, is polynomial time. We provide the first implementation of NGAC using an efficient linear time algorithm (bounded to the parts of the graph relevant to the user). With minimal modifications the methods used in PReview can be adapted for other types of policy review as well, such as identifying the users that can access an object. Furthermore, we provide a novel visualization approach that works by default with multiple access control policies and that enables efficient linear time review of user access rights. Given that the only other multi-policy approach with available reference implementations has been shown to not be scalable, our work thus enables for the first time a scalable methodology for limiting user access through simultaneous instantiation of multiple policies.

9 Acknowledgments

The authors would like to thank Aaron Zinger from Medidata for his help in understanding their implementation. We would also like to thank David Ferraiolo from NIST for his expert advice on how to construct realistic access control graphs, guidance on learning the nuances of the NGAC standard, and the associated NIST Policy Machine implementation. Lastly, we would like to thank Wayne Jansen from the NGAC ANSI committee for his help in understanding the standard.

References

[1] ANSI. American national standard for information technology, role-based access control (RBAC), 2004.

[2] ANSI. American national standard for information technology - next generation access control - functional architecture (NGAC-FA), 2013.

[3] ANSI. American National Standard for Information Technology - Next Generation Access Control - Generic Operations And Data Structures (NGAC-GOADS), 2016.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[5] K. Belyaev. tinyPM Prototype. www.github.com/kirillbelyaev/tinypm, 2015.

[6] K. J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[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. F. Brewer and M. J. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214. IEEE, 1989.

[10] 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*, 57(4):412–424, 2011.

[11] 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.

[12] GitHub. Github code repository. www.github.com.

[13] 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.

[14] B. W. Lampson. Dynamic protection structures. In*Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, AFIPS '69 (Fall), pages 27–38, New York, NY, USA, 1969. ACM.

[15] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, Jan. 1974.

[16] F. Mayer, D. Caplan, and K. MacMillan. *SELinux by example: using security enhanced Linux*. Pearson Education, 2006.

[17] Medidata Solutions Worldwide. Medidata Policy Machine code on github, version 1.1.0. www.github.com/mdsol/the_policy_machine, 2016.

[18] P. Mell, J. M. Shook, and S. Gavrila. Restricting insider access through efficient implementation of multi-policy access control systems. In *Proceedings of the 2016 International Workshop on Managing Insider Security Threats*, pages 13–22. ACM, 2016.

[19] 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.

[20] NIST. NIST Policy Machine code on github, version 1.5. www.github.com/PM-Master/PM, 2016.

[21] OASIS. eXtensible access control markup language (XACML) Version 3.0., OASIS Standard, Jan. 2013.

[22] OASIS. Organization for the advancement of structured information standards (OASIS). www.oasis-open.org, 2016.

[23] A. C. O'Connor and R. J. Loomis. 2010 Economic Analysis of Role-Based Access Control. Technical report, 2010.

[24] E. Sahafizadeh and S. Parsa. Survey on access control models. In*2010 2nd International Conference on Future Computer and Communication*, 2010.

[25] R. S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[26] 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.

[27] 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.

[28] U.S. Department of Defense. Trusted computer system evaluation criteria DoD 5200.28-STD, 1985.

[29] 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.

[30] W. Zeng, Y. Yang, and B. Luo. Content-based access control: Use data content to assist access control for large-scale content-centric databases. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 701–710, Oct 2014.

Author Biography