Mutation Analysis of NGAC Policies

Erzhuo Chen, Vladislav Dubrovenski, Dianxiang Xu
Department of Computer Science Electrical Engineering
University of Missouri - Kansas City
Kansas City, MO 64110, USA
{lcrnd,vadpb7}@mail.umkc.edu,dxu@umkc.edu

ABSTRACT

The NGAC (Next Generation Access Control) standard for attribute-based access control (ABAC) allows for run-time changes of the permission and prohibition configurations through administrative obligations triggered by access events. It makes access control more fine-grained and dynamic. However, it raises challenges for assuring the correctness of NGAC policies. As policy testing is an important technique for quality assurance, this paper presents an approach to mutation analysis of NGAC policies. It can evaluate the effectiveness of a testing method and reveal potential faults in an inadequately tested policy. The mutation analysis covers various types of potential faults in the assignments, associations, prohibitions, and obligations of NGAC policies. This paper also proposes an incremental testing approach that first validates the initial configuration of a policy and then the policy as a whole. It helps determine whether faults appear in the configuration or the obligations. To evaluate the work, we have developed four working policies and their test suites based on the current NGAC reference implementation. The empirical studies show that the mutation analysis can shed light on the strengths and weaknesses of the test suites. They also demonstrate the need for developing more cost-effective testing methods.

CCS CONCEPTS

• **Security and privacy → Access control**; *Software security engineering*; Authorization.

KEYWORDS

Access control; Next Generation Access Control (NGAC); mutation testing; policy testing

ACM Reference Format:
Erzhuo Chen, Vladislav Dubrovenski, Dianxiang Xu. 2021. Mutation Analysis of NGAC Policies. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies (SACMAT '21), June 16–18, 2021, Virtual Event, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3450569.3463563

SACMAT '21, June 16–18, 2021, Virtual Event, Spain

ACM ISBN 978-1-4503-8365-3/21/06…$15.00
https://doi.org/10.1145/3450569.3463563

1 INTRODUCTION

Next Generation Access Control (NGAC) is a new standard created by the American National Standard Institute to meet the needs for attribute-based access control (ABAC) in modern distributed systems [2]. Its reference implementation is being developed and deployed by NIST [17]. In ABAC, authorization elements are defined in terms of attributes, i.e., characteristics of subjects, actions, resources, and environments predefined and pre-assigned by an authority [5]. By combining various attributes into access control decisions, ABAC enables fine-grained access control. The increased complexity of ABAC has also raised concerns about quality assurance of access control policies. Faults in ABAC policies may lead to unauthorized accesses, escalation of privileges, and denial of service [22].

To address quality assurance of access control policies, policy testing has gained attention in the past decade. It aims to find potential faults or vulnerabilities in a policy by running it against certain test inputs (e.g., access requests). The most recent research has focused on XACML (eXtensible Access Control Markup Language) [12] polices [8],[15],[22],[23]. Although NGAC and XACML are both ABAC standards, they are very different. XACML specifies access control policies and policy sets with logical expressions and rules defined over attributes, whereas NGAC policies build upon sets and relations of attributes.

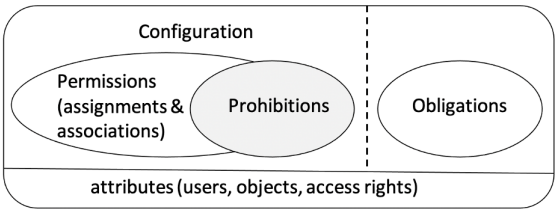

Figure 1: Abstract Structure of NGAC Policies

As illustrated in Figure 1, an NGAC policy specifies an initial access control configuration together with a set of obligations. The configuration consists of permission (assignment and association) and prohibition relations on the policy elements, such as user/object attributes, users, objects, and access rights. An obligation describes the response to an access event performed on behalf of the user. The response comprises administrative commands that dynamically change the configuration. Thus, NGAC obligations allow for run-time changes of permission and prohibition decisions. This offers unprecedented expressiveness for managing fine-grained access control of dynamic data and operations. In comparison, the notion

of obligation in XACML does not support run-time changes of access control rules.

Although the administrative obligations in NGAC make access control more dynamic and flexible, it can be difficult to assure correct access control. As stated in the NGAC standard specification, the application of NGAC has the potential of "grave harm to the authorization state through error or intent" [2]. With the NGAC reference implementation deployment, it is expected that more and more applications will adopt the NGAC standard. The US National Strategy for Information Sharing and Safeguarding recommended that the federal government should extend and implement the FICAM Roadmap across federal networks in all security domains [5], where the FICAM Roadmap has called out ABAC as a recommended access control model for promoting information sharing between diverse and disparate organizations [4]. Currently, no work has been done on the quality assurance of NGAC policies.

This paper presents an approach to mutation analysis of NGAC policies for evaluating the effectiveness of testing methods for revealing potential faults (errors) in NGAC policies. Given an NGAC policy, it automatically generates a set of policy mutants and executes each mutant against the given test suite representing a testing method under evaluation. A mutant is a modified version of the original policy where a policy element is mutated according to a fault model of representative types of errors in NGAC policies. A mutant is said to be killed if it fails one or more tests; otherwise, it is a live mutant. Testing effectiveness is indicated by mutation score, i.e., the ratio between the number of mutants killed and the total number of non-equivalent mutants.

The contributions of this paper are three-fold:

- This paper is the first work on mutation analysis of NGAC policies. It offers a comprehensive fault model that accounts for various potential errors in the configurations and obligations of NGAC policies. Based on the fault model, mutation operators are designed and implemented in an open-source tool. The tool can automatically generate mutants of a given policy, execute mutants with the policy's test suite, and report its effectiveness.
- This paper presents an incremental approach to testing NGAC policies. Given a policy, it applies combinatorial tests to the initial configuration and then coverage-based tests to the obligations in the policy (i.e., each obligation is tested at least once). The incremental testing facilitates fault localization. When an obligation test fails, it is likely caused by a faulty obligation because the initial configuration has passed all the tests. This paper is the first work to present and evaluate a specific method for testing NGAC policies.
- This paper presents a set of working policies of the NGAC reference implementation that can be reused by the research community. One of them is the first fully-fledged open-source application of the NGAC reference implementation with 30,249 lines of Java code. It was considered a very interesting case study by the NGAC implementation team, who helped with the policy design. We have created the test suites of the sample policies and evaluated their effectiveness. The experiment results indicate that the existing methods for testing NGAC policies are inadequate for high assurance. It motivates further research on new testing methods for NGAC policies.

The remainder of this paper is organized as follows. Section 2 reviews related work; Section 3 introduces NGAC policies; Section 4 presents the fault models and mutation operators for generating NGAC policy mutants; Section 5 describes the incremental approach to testing NGAC policies; Section 6 presents the empirical studies; Section 7 concludes this paper.

2 RELATED WORK

2.1 Mutation Testing

Mutation analysis is a widely applied approach to evaluating software testing methods [6]. It mutates a program by slightly changing the code without any syntax error. Each modified version, called a mutant, represents a potential error (fault). Given a program together with a passing test suite, mutation analysis generates a set of program mutants and runs each mutant against the test suite. A mutant is said to be killed if it fails one or more tests; otherwise, it is a live mutant. After all the mutants have been tested, the numbers of killed mutants and equivalent mutants are determined. An equivalent mutant has the same behavior as the original program for any input. The mutation score is defined by $\frac{K}{M-E}$, where K is the number of mutants killed, M is the total number of all mutants, and E is the number of equivalent mutants. $M - E$ is the number of non-equivalent mutants. Suppose the total number of mutants is 1,000, the number of equivalent mutants is 120, and the number of killed mutants is 748. The mutation score is 748/(1,000-120)=85%. Mutation score is an important indicator of testing effectiveness.

The essential hypotheses of mutation testing [13] include: (a) the mutants are representative of real faults, (b) developers write programs close to being correct, and (c) tests sufficient to detect simple faults can detect complex ones. Experiments have shown that mutants are indeed similar to real faults to evaluate testing techniques [7].

2.2 Policy Mutation

Mutation analysis of access control policies has also been the main approach to evaluating policy testing methods. MutaX is a tool for mutating Organization Based Access Control (OrBAC) policies [11][18]. It provides five types of mutation operators: replacing permission rule with prohibition, replacing prohibition rule with permission, changing role, changing context, and adding a rule. Xu et al. used these mutation operators for mutation analysis of Role-Based Access Control (RBAC) policies to evaluate a model-based testing method of RBAC policies [20][24].

Recent policy testing research has focused on XACML policies [8][9][15][22]. Several mutation analysis tools have been developed for different versions of the XACML standard, including [10] for v1.0, [1] for v2.0, and [22] for v3.0. They have been used to evaluate various testing methods for XACML policies. Xu et al. have also applied mutation analysis to generate test cases from XACML policies [23] and repair faulty XACML policies [21]. Elrakaiby et al. proposed a mutation-based approach to testing obligation policy enforcement, where obligations are independent of access control rules [3]. Obligations in this work are conceptually distinct from those in NGAC.

This paper presents the first mutation tool for NGAC policies based on the current NGAC reference implementation. As mentioned before, NGAC is significantly different from XACML even though both are ABAC standards. Mutation analysis of XACML policies deals with such policy elements as policy sets, policies, combining algorithms, and rules. This paper deals with the mutation of NGAC policy elements such as configuration (assignment, association, and prohibition relations) and obligations. The mutation tool of XACML v3.0 policies has 14 mutation operators [22] [23]. In this paper, the mutation analysis involves 40 mutation operators for NGAC policies.

3 NGAC POLICIES

An NGAC policy consists of an initial configuration of permissions and prohibitions and a set of obligations. Specifications of initial prohibitions and obligations are optional. In the reference implementation of the NGAC standard, the initial permissions (i.e., assignment and association relations), the initial prohibitions, and the set of obligations of a policy are specified in three separate files.

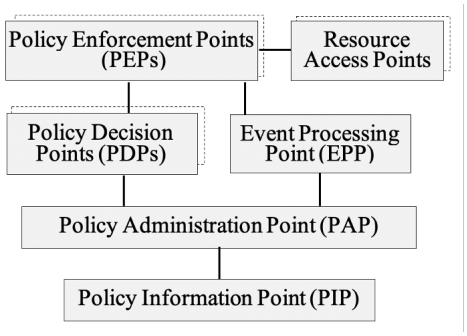

Figure 2: Architecture of NGAC Applications

NGAC allows the access control policy of an application to be separated from the business functionality. It reduces hard-coding of access control constraints and facilitates requirements changes. Figure 2 shows the general architecture of NGAC applications that use obligations to manage dynamic permissions and prohibitions. It includes one or more Policy Enforcement Points (PEPs) in the application's functional components and one or more Policy Decision Points (PDPs) of the NGAC standard. A PEP sends a user's request to a PDP, which renders an access decision according to the current permission and prohibition configuration and replies to the PEP. When an access event occurs in the PEP, it triggers the Event Process Point (EPP), which issues directives to the Policy Administrative Point (PAP) to find the matching obligations. The administrative commands in the matched obligations will be executed on behalf of the user who established the access event. The execution will change the current configuration.

To make this paper self-contained, we briefly introduce policy elements, configurations, and obligations of NGAC policies based on the standard specification [2].

3.1 Basic Elements

The basic elements of an NGAC policy comprise users, user attributes, objects, object attributes, policy classes, processes, and operations. Processes operate in a unique memory space of a system on behalf of a user. Authenticated users initiate access requests by creating processes. Operations denote modes of access to be performed on objects or policy information. Each operation requires one or more access rights. Processes are typically associated with the PEPs. To separate the analysis of NGAC policies from the specific PEP implementations, this paper will not elaborate on processes. Access requests and decisions related to processes are covered by those for users.

Let PE be the set of policy elements of an NGAC policy. $PE = U \cup UA \cup O \cup OA \cup PC$, where U is a finite set of users, UA is a finite set of user attributes, O is a finite set of objects, OA is a finite set of object attributes, and PC is a finite set of policy classes. U, UA, OA, and PC are disjoint, i.e., each of these elements is uniquely identified. $O \subseteq OA$ because objects are also treated as object attributes.

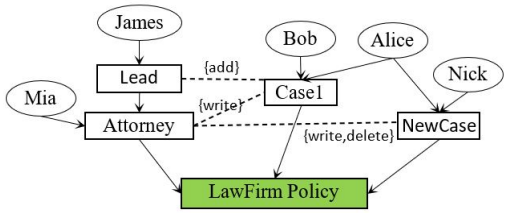

Figure 3: A Simple Policy Graph

A policy class is a container of certain users, user attributes, objects, and object attributes. The notion of policy classes helps modularize a complicated policy into multiple groups and avoid interference among access rights of different user attributes and object attributes. For example, one person can be enrolled as a student in one course and a teaching assistant in another course. All elements in PE except policy class names should be contained in one or more policy classes.

In Figure 3, the policy elements are organized in one policy class. $U = \{Mia, James\}, UA = \{Attorney, Lead\}, O = \{Nick, Alice, Bob\}$, $OA = \{NewCase, Case1, Nick, Alice, Bob\}, PC = \{LawFirmPolicy\}$.

3.2 Configurations

A configuration consists of three relations on the policy elements: assignment, association, and prohibition. The assignment and association relations specify what is permitted. However, the permissions can be overridden by the prohibition relation.

3.2.1 Assignments. The assignment relation, $ASSIGN$, is defined as follows:

$$ASSIGN \subseteq (U \times UA) \cup (UA \times UA) \cup (OA \times OA) \cup (UA \times PC) \cup (OA \times PC)$$

Given $(x, y) \in ASSIGN$, i.e., x is assigned to y, x is the direct ascendant of y and y is the direct descendant of x. A user can be assigned to a user attribute, and a user attribute can be assigned to

another user attribute or a policy class. An object can be assigned to an object attribute, and an object attribute can be assigned to another object attribute or policy class. A policy class cannot be assigned to any other policy element. The assignments of objects to object attributes are implied by $OA \times OA$ because of $O \subseteq OA$.

In Figure 3, the assignment relation is { (Mia, $Attorney$), ($James$, $Lead$), ($Alice$, $Case1$),($Alice$, $NewCase$), (Bob, $Case1$), ($Nick$, $NewCase$), ($NewCase$, $LawFirmPolicy$), ($Case1$, $LawFirmPolicy$), ($Lead$, $Attorney$),($Attorney$, $LawFirmPolicy$) }.

The assignment relation can be visualized as acyclic directed graphs (hierarchies), as illustrated in Figure 3. The root of each hierarchy is a policy class. It connects two sub-hierarchies: one for the users and user attributes, and the other for the objects and object attributes. Compared to RBAC, NGAC introduces object attribute hierarchies while generalizing role hierarchies of RBAC by user/user attribute hierarchies.

Let $contains(x, y)$ denote x contains y. It is defined as follows: (1) $contains(x, x)$ for any $x \in UA \cup OA \cup PC$. (2) $contains(x, y)$ if $(y, x) \in ASSIGN$. (3) $contains(x, z)$ if $contains(x, y)$ and $contains(y, z)$. In the directed graphs of the assignment relation, $contains(x, y)$ indicates that there is a path from node y to node x.

In Figure 3, $LawFirmPolicy$ contains all other elements. $James$ is contained by $Attorney$. $Alice$ is contained by $Case1$ and $NewCase$.

3.2.2 Associations. The association relation specifies access rights between a user attribute and an object (or user) attribute. Let AR be the set of all access rights and $AT = UA \cup OA$ be the set of user attributes and object attributes. The ternary association relation is defined as $ASSOCIATION \subseteq UA \times 2^{AR} \times AT$.

In Figure 3, the association relation is {($Attorney$, {$write$, $delete$}, $NewCase$), ($Attorney$, {$write$}, $Case1$), ($Lead$, {add}, $Case1$) }. ($Attorney$, {$write$, $delete$}, $NewCase$) specifies that any user directly assigned to user attribute $Attorney$ (e.g., Mia) has the access rights of $write$ and $delete$ on an object directly assigned to object attribute $NewCase$ (e.g., object $Nick$).

3.2.3 Prohibitions. The prohibition relation defines the negation of associations to constrain the access rights granted to policy elements, i.e., $PROHIBITION \subseteq (U \cup UA) \times 2^{AR} \times 2^{UA \cup U \cup OA}$. Each prohibition is a triple (s, ars, tps), where $s \in U \cup UA$ is a user or user attribute, $ars \subseteq AR$ is a set of access rights, and $tps \subseteq UA \cup U \cup OA$ is a target prohibition set of policy elements. tps is not specified directly. Instead, its syntax includes an intersection operator ($conjunctive$ or $disjunctive$) followed by one or more container specifications (α_i, c_i), where α_i is a complement operator (either $inclusion$ or $exclusion$) and $c_i \in UA \cup OA \cup PC$ is a container (user attribute, object attribute, or policy class). Let $\rho(\alpha_i, c_i)$ be the set of policy elements defined by container specification (α_i, c_i). $\rho(inclusion, c_i)$ of an inclusive container comprises all policy elements contained by container c_i. $\rho(exclusion, c_i)$ of an exclusive container consists of all the policy elements not contained by container c_i. $\rho(\alpha_i, c_i) \subset UA \cup U$ if $c_i \in UA$. $\rho(\alpha_i, c_i) \subset OA$ if $c_i \in OA$. $\rho(\alpha_i, c_i) \subset UA \cup U \cup OA$ if $c_i \in PC$.

Given target prohibition specification ($< intersection >$, (α_1, c_1), ..., (α_n, c_n)), the target prohibition set of policy elements tps is defined as follows:

$$tps = \begin{cases} \rho(\alpha_1, c_1), & n = 1 \\ \bigcap_{i=1}^{n} \rho(\alpha_i, c_i), & conjunctive\ intersection\ for\ n > 1 \\ \bigcup_{i=1}^{n} \rho(\alpha_i, c_i), & disjunctive\ intersection\ for\ n > 1 \end{cases}$$

If there is only one container specification, $tps = pe(\alpha_1, c_1)$; otherwise, the intersection operator applies to all container specifications.

3.2.4 Access Decisions. In the NGAC standard specification, an access request comprises a process, an operation (a list of access rights), and a list of enumerated arguments (policy elements). Considering that processes are typically created for users in the PEPs, this paper uses the basic form of access requests, i.e., (s, ar, pe), where s is a subject (user or user attribute), ar is an access right, and pe is a policy element (user, user attribute, object, or object attribute). The advantage is that we can analyze and test an NGAC policy without the PEPs. Each request in the standard specification can be converted into one or more basic requests.

Given an access request (s, ar, pe) and a configuration of assignments, associations, and prohibitions, the access decision is "permit" if the following conditions hold, otherwise "deny":

- (s, ar, pe) is permitted by the assignment and association relations. There exists association (sa, ars, at) in each policy class pc containing pe such that: (1) $contains(sa, s)$, (2) $ar \in ars$, (3) $contains(at, pe)$, and (4) $contains(pc, at)$.
- No user-based prohibition precludes request (s, ar, pe), i.e., no prohibition (u, ars, tps) exists such that: (1) $s = u$, (2) $ar \in ars$, and (3) $pe \in tps$.
- No user attribute-based prohibition precludes request (s, ar, pe), i.e., no prohibition (ua, ars, tps) exists such that: (1) $contains(ua, s)$, (2) $ar \in ars$, and (3) $pe \in tps$.

In Figure 3, the association relation directly specifies the following privileges: ($Attorney$, $write$, $NewCase$), ($Attorney$, $delete$, $NewCase$), ($Attorney$, $write$, $Case1$), ($Lead$, add, $Case1$). The user attribute hierarchy and object attribute hierarchy also imply the following privileges: ($Lead$, $write$, $NewCase$), ($Lead$, $delete$, $NewCase$), (Mia, $write$, Bob), ($James$,$write$, $Alice$), ($James$, add, Bob). However, access decisions also depend on the prohibitions.

Let us consider user "Mia", access right "$write$", and containers $Case1$ (c_1) and $NewCase$ (c_2) in Figure 3. (1) If the container specification is ("Mia", "$write$", conjunctive, (exclusion, c_1), (inclusion, c_2)), then $tps = \{Nick\}$. (2) If the container specification is ("Mia", "$write$", disjunctive, (exclusion, c_1), (inclusion, c_2)), then $tps = \{Nick, Alice\}$. (3) If the container specification is ("Mia", "$write$", conjunctive, (exclusion, c_1), (exclusion, c_2)), then $tps = \varnothing$. (4) If the container specification is ("Mia", "$write$", disjunctive, (inclusion, c_1), (inclusion, c_2)), then $tps = \{Nick, Alice, Bob\}$.

Prohibitions can be an effective tool to manage conflict of interest and achieve separation of duties. In Figure 3, $Alice$ is involved in both $Case1$ and $NewCase$. With the prohibitions described above, we can have fine-grained control of the exact permissions Mia will have on the nodes contained by $Case1$ and $NewCase$.

3.3 Obligations

The obligation relation specifies the responses triggered automatically when certain events of access requests have been completed. The responses comprise administrative commands that will update the assignment, association, and prohibition relations of the current configuration and change run-time authorization state and access decisions. The user for carrying out the response to a matching event needs to possess adequate authority at the time of matching. The use of administrative commands distinguishes NGAC's notation of obligation from that in XACML [12] and the Usage CONtrol Model (UCON) [14]. It is also different from that in Administrative RBAC (ARBAC) [16]. Administrative commands in ARBAC are performed by system administrators rather than triggered automatically by run-time access events.

Each obligation is specified by an event pattern and a response. The event pattern consists of the following components:

$$[< user\ spec >][< pc\ spec >] < op\ spec > [< pe\ spec >]$$

$< user\ spec >$ denotes a set of users and/or user attributes. If it is omitted, any event that matches the other components would be accepted by the event pattern. $< pc\ spec >$ represents a set of policy classes. Considering that one user could exist in multiple policy classes and initiate the same event in different policy classes, only the events in the policy classes listed in $< pc\ spec >$ will be processed. If $< pc\ spec >$ is omitted, events are not filtered by policy classes. $< op\ spec >$ specifies a set of event operations (access rights). $< pe\ spec >$ refers to a set of policy elements. An event on these policy elements and those contained by these policy elements is accepted as a match. One event may match the event patterns of multiple obligations.

A response consists of one or more conditional actions. Conditional action is specified as follows:

$$[if < condition > \ then] < action > \{, < action >\}$$

A condition includes one or more factors, i.e., $< factor > \{and\ < factor >\}$. An action is performed only when the specified condition holds, i.e., all the factors in the condition are true. There are five types of actions: *create*, *assign*, *grant*, *deny*, and *delete*. As defined below, all of them change the configuration:

$$< create >::= create < create\ what >< create\ where >$$

$$< assign >::= assign < assign\ what > [< assign\ where >]$$

$$< grant >::= grant < grant\ to >< grant\ what > [< grant\ on >]$$

$$< deny >::= deny < deny\ to >< deny\ what >< deny\ on >$$

$$< delete >::= delete < delete\ subaction >$$

A *create* action adds a new policy element or a new obligation. An *assign* action adds a new assignment to the assignment relation. A *grant* action adds a new association to the association relation. A *deny* action adds a new prohibition. A *delete* action removes the specified policy elements.

The following is a simple obligation from the *LawFirm* example:
$Event : \{anyUser : LeadAttorney, withdraw, NewCase\}$
$Response : \{delete, NewCase\}$

If any user contained by "*LeadAttorneys*" performs the operation "*withdraw*" on the policy element "*NewCase*", then the node "*NewCase*" will be deleted.

In essence, the obligation relation specifies the state transitions from one configuration (state) to another. Let $\psi = (UA \cup U) \times AR \times (UA \cup U \cup OA)$ be the set of all possible access requests. Each configuration ϕ defined by the assignment, association, and prohibition relations boils down to the access decisions for all possible access requests, i.e., $\phi \subseteq \psi$ (the interpretation is that only the requests in ϕ are permitted and the others are all denied). Thus, the obligation relation is a binary relation on the set of configurations, i.e., $OBLIGATION \subseteq 2^{\psi} \times 2^{\psi}$. For example, $|\psi|$=42,400 for the GPMS-NGAC case study in Section 6 (i.e., the size of the all-combinations test suite in Table 6). The number of possible configurations can be as large as $2^{42,400}$. Therefore, it can be hard to reason about the correctness of obligations.

4 MUTATION OF NGAC POLICIES

Policy mutation is the act of slightly modifying a given policy without introducing any syntax error. The modified versions are called policy mutants. In this paper, policy mutants are created automatically by mutation operators (programs) according to a fault model (i.e., a collection of fault types). Each non-equivalent mutant of a correct policy represents a possible fault in the policy during its development process. Policy mutation is meaningful even when we do not know whether a given policy is correct. If it is faulty, one of its mutants may be a correct version.

In the following, we discuss mutation of the initial configuration and obligations of an NGAC policy, respectively.

4.1 Mutation of Initial Configuration

The initial configuration of an NGAC policy consists of the assignment, association, and prohibition relations. Mutation applies to all these relations.

4.1.1 Mutation of Assignments. When a correct policy is supposed to have an assignment (x, y), the possible faults are incorrect assignments with wrong direction (y, x), wrong descendent (x, z), wrong ascendant (z, y), and missing the assignment. As shown in Table 1, these faults can be represented by the mutants created with mutation operators RAD, CAD, CAA, and RAG, respectively. Another possible fault is that there is an extra assignment in the incorrect policy.

When creating mutants, we need to ensure that the changes will not lead to syntax errors. Thus, the mutation operators of the assignment relation must observe the following restrictions of correct assignments.

- The assignment relation is irreflexive. We cannot create a mutant by adding a new assignment (x, x).
- We cannot assign a policy class to any other policy elements. For example, mutation operator RAD does not apply to any assignment whose descendent is a policy class.
- The directed graph representation of the assignment relation is acyclic. A mutant is invalid if it contains a cycle. Assume that ua_1 is assigned to ua_2 and ua_2 is assigned to ua_3 in the given policy. We cannot create a mutant by adding a new assignment from ua_3 to ua_1 because it would result in

Table 1: Mutation Operators for the Assignment Relation

No	Name	Meaning	Fault Type
1	RAD	Reverse Assignment Direction	Wrong assignment direction
2	CAD	Change Assignment Descendent	Wrong descendent node
3	CAA	Change Assignment Ascendent	Wrong ascendent node
4	RAG	Remove an AssiGnment	Missing assignment
5	AAG	Add an AssiGnment	Extra assignment

a cycle. In general, we cannot create mutants by introducing assignment (x, y) when y is contained by x. This applies to mutation operators RAD, CAD, CAA, and AAG in Table 1.

- The assignment relation is policy class connected, i.e., every element (except policy class) must be contained by some policy class. When applying mutation operators RAD, CAD, CAA, and RAG, a policy element could be disconnected from policy classes. When such disconnection happens to a policy element, we add a new assignment from the policy element to the same policy class as the assignment being mutated.
- The assignment relation precludes object attribute to user attribute assignments and user attribute to object attribute assignments. All mutation operators of the assignment relation avoid such assignments.

We also avoid generating mutants known to be equivalent to the original assignment relation. For example, if $(x, y) \notin ASSIGN$ and $contains(y, x)$, we will not add assignment (x, y) to create a mutant because it does not affect the privileges.

4.1.2 Mutation of Associations. When a correct policy is supposed to have an association (sa, ars, at), the possible faults are incorrect associations with wrong source user attribute (sa', ars, at), an incorrect target object/user attribute (sa, ars, at'), a missing access right (sa, ars', at) where $ars' \subset ars$, an extra access right $ars' \subset ars'$ where $ars \subset ars'$. In addition, an incorrect association relation may have a missing or extra association. Table 2. shows all fault types of the association relation and corresponding mutation operators.

To avoid generating invalid association mutants, we classify access rights into two groups as some access rights are only applicable to objects or object attributes while others are only applicable to users or user attributes. When applying mutation operator AARA (add one access right to association) to an association (sa, ars, at), if at is an object attribute, we only choose the access rights applicable to object attributes.

We also avoid generating duplicate mutants. When association (sa, ars, at) has only one access right (i.e., $|ars| = 1$), RARA for removing one access right and RACR for removing the association will result in the same mutant. To avoid duplication, we apply RARA only to the associations with two or more access rights.

When applying mutation operator AAC, which stands for adding one association with one access right, the new added association (sa, ar, at) will make no change to the initial configuration when there already exists an association (sa, ars, at) where $ar \in ars$. Adding an association $(sa, \{ar\}, at)$ when there already exists an association (sa, ars, at) such that $ar \notin ars$ is the same as adding an access right ar to association (sa, ars, at), which is covered by mutation operator AARA (adding one access right to an association). Similar cases are considered by mutation operator AAC (Add AssoCiation with one access right).

Table 2: Mutation Operators for the Association Relation

No	Name	Meaning	Fault Type
1	CUAA	Change User Attribute of an Association	Incorrect user attribute in association
2	COAA	Change Object Attribute of an Association	Incorrect object attribute in association
3	RARA	Remove one Access Right from an Association	Missing an access right in association
4	AARA	Add one Access Right to an Association	Extra access right in association
5	RAC	Remove an AssoCiation	Missing association
6	AAC	Add AssoCiation with one access right	Extra association
7	RARAA	Remove an Access Right from All Associations	Missing an access right

4.1.3 Mutation of Prohibitions. A prohibition consists of four syntactic components: subject, operation (access rights), intersection, and container specifications. The mutation operators of prohibitions are designed regarding the fault types of these four components. They are shown in Table 3. Operator 1 (CSS) represents the fault of the wrong subject. Operators 2-4 involve different forms of incorrect access right set. Operators 5-9 deal with incorrect target prohibition set, including incorrect containers, incorrect intersection operator, and incorrect complement operator. Operator 10 (ROP) represents the fault of missing one prohibition.

The following are the main considerations about generating prohibition mutants:

- Avoid creating a mutant by adding a container that contains no policy element. It imposes no change to the prohibition. This applies to Mutation operator AOC (Add One Container).
- Avoid creating a mutant by adding a duplicate container with the same complement operator. It will not change the target prohibition set. However, adding a duplicate container with the reversed complement operator can result in a meaningful mutant. This applies to mutation operator AOC.
- Adding a new container with different complement operators will cause different changes to the target prohibition set. Whenever a new container C_1 is added, for example, two mutants will be generated. One mutant adds $\{inclusion(T(C_1))\}$ while the other adds $\{exclusion(T(C_1))\}$ to the target prohibition set. This applies to mutation operator AOC.
- When a prohibition involves only one access right, ROAR for removing one access right and ROP for removing one

Table 3: Mutation Operators for the Prohibition Relation

No	Name	Meaning	Fault Type
1	CSS	Change Subject Spec	Wrong subject
2	AOAR	Add One Access Right	Extra access right
3	COAR	Change One Access Right	Wrong operation
4	ROAR	Remove One Access Right	Missing access right
5	RIS	Reverse InterSection	Wrong intersection
6	AOC	Add One Container	Extra container
7	COC	Change One Container	Wrong container
8	ROCT	Remove One ConTainer	Missing container
9	RCT	Reverse ComplemenT	Wrong complement
10	ROP	Remove One Prohibition	Missing prohibition

prohibition will result in the same mutant. To avoid duplication, we apply ROAR only to the prohibitions with more than one access right.

4.2 Mutation of Obligations

As described in Section 3.3, an obligation consists of an event pattern and a response. The mutation operators of obligations are designed based on the fault types of event patterns and responses. They are shown in Table 4. Operator #1 (ROB) represents the fault of missing an obligation. Operators #2-8 deal with incorrect user specification, operation specification, and target policy element specification in the event pattern of an obligation. Operators #9-12 involve different forms of incorrect conditions in the response pattern of an obligation. Operators #13-18 handle various incorrect actions in the response part of an obligation.

A major challenge of obligation mutation is to verify if a candidate mutant is valid. Unlike the validity checking of the configuration mutants, where invalid mutants can be detected once these mutants get loaded, an invalid obligation mutant cannot be detected unless the event, which matches the event pattern of the obligation with errors, has been executed. The following are the main considerations to avoid syntax errors when generating obligation mutants. When applying mutation operator CAC (change action), not all types of actions are considered to be a candidate of a new action because actions have specific structures. Constructing a new action is non-trivial and error-prone. We only change an action to another with the same structure. For example, both "create" and "assign", have three components: action name, policy element the action happens on, and another policy element where the former policy element to be assigned to. Actions "grant" and "deny" also share the same structure. The mutation on action "delete" is not considered because it has to be aware of the whole policy elements.

We also aim to avoid generating duplicate mutants and equivalent mutants. For example, mutation operator CEU (change event user) is to change user name in the event to another user name, and REU (remove event user) is to remove a user name in the list of users in the event. Assume the origin list contains John, Tom. Applying CEU to the list would introduce a new list L1= {John, John}, if changing Tom to John. Applying REU to the list results in list L2= {John}, if Tom is removed. L1 and L2 are the same even

Table 4: Mutation Operators for the Obligations

No	Name	Meaning	Fault Type
1	ROB	Remove one OBligation	Missing obligation
2	CEU	Change Event User	Wrong user name
3	REU	Remove Event User	Missing user name
4	CEO	Change Event Operation	Wrong operation
5	AEO	Add Event Operation	Extra operation
6	REO	Remove Event Operation	Missing operation
7	CEPE	Change Event Policy Element	Wrong target
8	REPE	Remove Event Policy Element	Missing target
9	ROC	Remove One Condition	Missing condition
10	NCD	Negate One Condition	Wrong condition
11	ROF	Remove One Factor	Wrong condition
12	NOF	Negate One Factor	Wrong condition
13	ROA	Remove One Action	Missing action
14	COA	Change One Action	Wrong action
15	ICA	Incorrect Create Action	Wrong create
16	IAA	Incorrect Assign Action	Wrong assignment
17	IGA	Incorrect Grant Action	Wrong grant
18	INA	Incorrect Deny Action	Wrong deny

though both are syntactically valid. Mutation operators avoid generating duplicate lists. REO (remove event operation) only applies to those with two or more operations. Otherwise, it will have the same effect as ROB (remove obligation). Similarly, ROC (remove one condition) and ROF (remove one factor) are also overlapping. When there is only one factor in a condition, removing the factor is the same as removing the condition.

5 INCREMENTAL POLICY TESTING

We propose an incremental approach to the testing of NGAC policies. Given an NGAC policy, we first test the initial configuration and then the policy as a whole. The advantage is that it helps localize faults. If the initial configuration has passed all the tests, but the entire policy (including the obligations) fails some tests, the failure would have to do with the obligations. Our approach takes the black-box strategy, i.e., it does not use the knowledge about the implementation details in the input files of the assignments, associations, prohibitions, and obligations. Instead, it is based on the specifications of user attributes, object attributes, and obligations.

It is worth mentioning that the policy testing approach aims at finding potential policy faults in the development process. It is performed under the development environment, which is often different from the operational environment. For example, an operational learning management system (e.g., Canvas) may involve 30,000 users and 1,000 courses. Its development environment does not need such large numbers of users and courses. In the context of policy testing, the complexity depends on the number of user attributes, access rights, and object attributes rather than users and objects. In our empirical studies, we only assign a small number of users (or objects) to each user attribute (or object attribute) adequate for testing purposes.

5.1 Testing the Initial Configuration

A test case for the initial configuration consists of an access request and an oracle value. The access request is a triple (s, ar, pe), where subject $s \in UA \cup U$ is a user or user attribute, $ar \in AR$ is an access right, and $pe \in UA \cup U \cup OA$ is a policy element. The oracle value is the expected access decision according to the access control requirements specification. It is either true (i.e., "permit") or false (i.e., "deny"). If it is different from the actual access decision rendered by the initial configuration, then the initial configuration is faulty. When we believe the policy under test is correct, its actual decision on a request is saved as its oracle value. It can then be used to test if a revised policy has resulted in an incorrect decision. This is very helpful because policies are often developed in an incremental fashion and revised from time to time.

In this paper, we have implemented the following methods for automatically generating access requests:

- All combinations: This test suite covers all possible access requests, i.e., all possible combinations of (s, ar, pe), where $s \in UA \cup U$, $ar \in AR$, and $pe \in UA \cup U \cup OA$. It can be inefficient for large configurations.
- Pairwise combinations: This test suite covers all possible pairs of (s, ar), (s, pe), and (s, pe). Note that each request implies three pairs. For example, $(Lead, add, Case)$ covers $(Lead, add)$, $(Lead, Case)$, and $(add, Case)$. For the same configuration, the pairwise test suite is typically much smaller than the all-combinations suite.

The test suite generated by each method is stored in a spreadsheet. To execute the tests, we developed a test harness to read and execute each test automatically.

5.2 Testing the Obligations

As discussed before, obligations are triggered by access events. To test an obligation, we need to issue an appropriate access request. However, the initial configuration may not grant this access request. In other words, testing an obligation may require a sequence of other requests to lead the configuration to a certain state that the target obligation can be triggered by permitted access. As such, an obligation test consists of a sequence of access requests and oracle values – each request triggers an obligation to change the configuration, and its oracle value consists of the expected changes to the configuration (i.e., certain requests should be permitted or denied).

Due to the complexity of obligations, we found it difficult to generate obligation tests automatically. In this paper, we designed obligation tests manually according to the obligation specifications and then converted each of them into a JUnit test for automated execution. Our test design aimed to cover each obligation at least once by considering the sequential constraints among the obligations (e.g., obligation A cannot be triggered before obligation B). One test may exercise multiple obligations. The GPMS-NGAC case study with 19 obligations in the experiments has eight JUnit tests. These tests have a total of 764 lines of code.

Appendix A provides a sample test from the GPMS-NGAC case study. The sequence of access events includes add-copi (lines 11-13), add-sp (lines 15-17), submit (lines 19-25), and chair-disapprove (lines 27-31). Line 11 assures the precondition of add-copi obligation

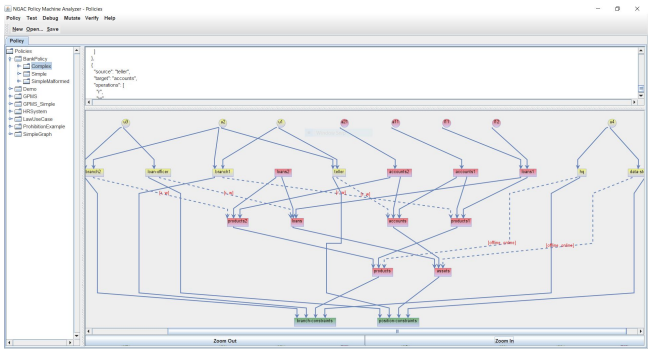

Figure 4: POMA GUI

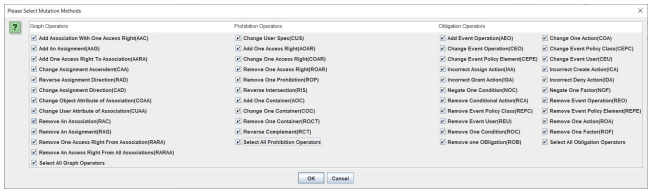

Figure 5: Mutation Operators

is satisfied (i.e.., the even pattern is defined correctly) before the add-copi event is issued in line 12. This is similar for other events.

6 EMPIRICAL STUDIES

6.1 Tool Implementation

We have implemented the mutation analysis approach in a tool called POMA (POlicy Machine Analyzer). The NGAC reference implementation is coded in Java. So is POMA.

As shown in Figure 4, POMA provides a GUI for visualizing policy graphs (i.e., the assignment and association relations in a policy). As illustrated in Figure 5, POMA also allows the user to choose certain mutation operators for mutation analysis. Given an NGAC policy and a test suite (in the form of a spreadsheet for the initial configuration or in the form of a JUnit class), POMA can automatically generate policy mutants with the chosen mutation operators, execute the test suite against each mutant, and report whether each mutant is killed.

6.2 Subject Policies

Table 5 shows the metrics of the subject policies in our empirical studies, including numbers of policy classes, user attributes, object attributes, assignments, associations, prohibitions, and obligations. As mentioned before, the initial configuration of each policy only assigns a small number of users (or objects) to user attributes (or object attributes) because the policies under test are meant for the development environments rather than the operational environments. Considering that prohibitions and obligations are optional, we have two policies with no obligation and two policies with no prohibition in the initial configuration.

In the following, we give a brief introduction to the policies.

6.2.1 GPMS-NGAC. GPMS-NGAC (or simply GPMS in all relevant tables) is the first fully-fledged open-source application of the NGAC reference implementation that utilizes NGAC's obligation mechanism. It includes not only the access control policy but also the functionality. The authors received much help from the NGAC implementation team during the development process; meanwhile, the NGAC reference implementation was developed and improved. NGAC's obligation mechanism was not fully implemented when the GPMS-NGAC project started.

GPMS-NGAC is a web-based application that aims to automate the grant proposal approval workflow at an academic institution. Such workflows apply to many application domains (e.g., healthcare and finance). GPMS-NGAC originated from GPMS, which used XACML as the access control language [19]. GPMS-NGAC has 30,249 lines of Java code. Its NGAC policy template consists of four policy classes and 19 obligations. The most important feature is that each proposal object carries its own NGAC policy instance, including the current configuration and obligations because it depends on the users involved, and its access control is very dynamic. As this paper focuses on testing, we use the development environment with a small number of users assigned to limited user attributes. An operational environment of GPMS-NGAC may have many users and data objects.

6.2.2 LawFirm Policy. Many law companies utilize case tracking systems. Large law companies utilize such systems for billing and file sharing, while smaller ones utilize simple emails for everything because case tracking systems get very expensive. Also, those systems are not very secure, as everyone in the company generally has access to every case file. That may lead to conflicts of interest if, for example, a company works for multiple clients that have such opposite interests. Therefore, we created this case study to address various issues in case tracking systems.

Since we intended to make this example small, we utilize only three policy classes, called "LawFirm Policy", "Case Policy", and "ValueType Policy". The first one contains two hierarchies of attributes. One hierarchy consists of user attributes for managing the employees, and the other hierarchy consists of object attributes for managing the cases.

Each office has attorneys of various ranks (Lead Attorneys, Attorneys, and Interns), each with various access privileges. As this policy is a mock case study intended to be scalable for performance testing, there can be an unlimited number of offices, cases, attorneys, and interns. "Case Policy" is the policy that is used for approving a new case. All of the dynamic changes to this policy are done via obligations.

6.2.3 Bank Policy. This policy is the management structure of a hierarchy in a bank system inspired by the example in the NGAC standard. It only contains one policy class. Similar to the LawFirm Policy, one hierarchy has $UA \cup U$ represented by various bank offices that contain managers and officers. Another one has $OA \cup O$, which is used to represent loans, accounts, and products.

6.2.4 Healthcare Policy. Healthcare, in general, is widely used in many articles about access control. Due to The Health Insurance Portability and Accountability Act (HIPAA), information integrity in clinical settings is critical because of the private nature of the documents stored. The healthcare policy demonstrates an access control system between the healthcare personnel and their patients. When creating this policy, we intentionally created every combination of prohibitions' logical relations described in Section 3.2.3.

Table 5: Subject Policies

	#PC	#UA	#OA	#ASM	#ASC	#PRO	#OBL
Healthcare	1	4	5	7	2	1	-
Bank	2	6	10	33	6	-	-
LawFirm	2	10	5	29	7	5	7
GPMS	4	34	27	91	8	-	19

#PC: number of policy classes; #UA: number of user attributes; #OA: number of object attributes; #ASM: number of assignments; #ASC: number of associations; #PRO: number of prohibitions; #OBL: number of obligations

Table 6: Test Suites of the Subject Policies

	Configuration		Obligations	
	Pairwise	All-Combinations	LOC	#JUnit Tests
Healthcare	36	72	-	-
Bank	240	960	-	-
LawFirm	1,066	3,500	252	7
GPMS	4,240	42,400	764	8

6.3 Policy Testing

We have applied the testing approach when developing the policies. Table 6 shows the test suite sizes. The pairwise test suite for an initial configuration is much smaller than the all-combinations counterpart. Their sizes range from 10% to 50% of the latter.

The testing has revealed various faults during the implementation of the subject policies. One major bug that one can produce during the design of a policy graph is the Detached Node Problem. In Figure 6 only the following privilege exists: (*Attorneys*, *read*, *NewCase*). Note, even though *Alice* is contained by *NewCase*, the following privilege does not exist: (*Attorneys*, *read*, *Alice*). That is because *Alice* is also contained by *Case*1, and *Case*1 belongs to another PC: *OrganizationPolicyClass*. In *OrganizationPolicyClass*, there is no container that contains *Alice* AND has the association with *read* access right. Therefore, *Alice* is considered a "Detached Node" with the respect to the association relation (*Attorneys*, *read*, *NewCase*). While this problem might not be useful for malicious purposes, it can definitely make the policy design and implementation a lot harder since the access rights will simply not be shown. Furthermore, until a recent fix, the policy analyzer function in NGAC policy-machine did show this access right in discrepancy with the policy decider function.

We also would like to discuss a few issues found in the obligation design of the Law Firm policy. A simple case tracker in a law firm can be very complex from the testing perspective. In our case study, there are two loops: one for when the case is sent back for review and another for disapproving of the newly added case by anyone.

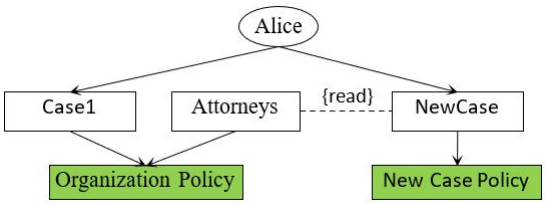

Figure 6: Detached Node Problem

First, we found that if the new case was accepted by Lead and then disapproved by Attorney, this would create a "zombie" state of such case, meaning that no one will perform any actions on it. Another test found that the state "Accepted" is not reachable if an Attorney initially refuses to take the case, even though the Lead can still choose to accept such a case.

6.4 Mutation Analysis

After having tested the subject policies without any failure (i.e., all known faults have been fixed), we continued to apply mutation analysis to evaluate the strengths and weaknesses of the test suites created by the incremental testing method.

For each subject policy and each test suite, we used the following protocol to conduct the mutation analysis:

- Select all mutation operators relevant to the test suite (i.e., the configuration mutation operators for the test suite of the initial configuration and the obligation mutation operators for the obligation tests).
- Generate and execute mutants against the test suite.
- Examine the live mutants to determine the number of equivalent mutants.
- Calculate the mutation score, which is the number of killed mutants divided by the total number of non-equivalent mutants. Mutation score is the main performance indicator of fault detection capability of the testing method.

A live mutant is a mutant not killed by the test suite, i.e., no failure was reported. It may or may not be equivalent to the original counterpart. A configuration mutant is equivalent to the original configuration if both render the same access decision for every possible request. As the all-combinations test suite covers all possible requests, any live configuration mutant of this test suite is equivalent to the initial configuration. Therefore, the equivalent mutants of each initial configuration can be determined automatically. The mutation score of an all-combinations test suite is always 100%.

An obligation mutant is equivalent to the original obligation if and only if both policy versions (including the initial configuration) lead to the same configuration for every possible sequence of access requests. The problem of determining equivalent obligation mutants is believed to be intractable due to the complexity of obligations and the infinite number of access request sequences. In this paper, we examined each live obligation mutant manually.

6.4.1 Mutation Analysis of the Initial Configurations. Table 8 shows the results of configuration mutation analysis. The all-combinations method yields 100% mutation score for all policies. For relatively simple policies, *healthcare*, *bank*, and *law firm*, the pairwise method kills most of the mutants. For the most complex policy *GPMS-NGAC*, its mutation score drops to 51.3%.

Table 7: Configuration Mutants

	#Total Mutants	#Equivalent Mutants	#Non-Equivalent Mutants
Healthcare	225	39	186
Bank	752	27	725
LawFirm	1,890	266	1,624
GPMS	25,483	2,496	22,987

Table 8: Results of Configuration Mutation Analysis

	Pairwise			All- Combinations		
	#KM	MS(%)	MKPT	#KM	MS(%)	MKPT
Healthcare	148	79.6	4.11	186	100	2.58
Bank	573	79.0	2.39	725	100	0.76
LawFirm	1,439	88.6	1.35	1,624	100	0.46
GPMS	11,784	51.3	2.78	22,987	100	0.54

#KM: number of killed mutants; MS(%): mutation score; MKPT: average number of mutants killed per test

Because the all-combinations method achieves 100% mutation score, it can reveal all possible faults in the initial configuration. However, it is not necessarily efficient or scalable for complex configurations. For example, it has 42,400 test cases for the GPMS-NGAC policy. It can be a daunting task to determine the oracle value for each access request according to the access control requirements. Although the mutation scores of the pairwise test suites range from 51.3% to 88.6%, they are more cost-effective than the all-combinations test suites in terms of the average number of mutants killed per test (MKPT). As shown in Table 8, the pairwise method has a better MKPT score than the all-combinations method. For the most complicated policy GPMS-NGAC, Pairwise's MKPT is five-fold of the all-combinations method. As such, neither method is satisfactory. It is desirable to develop a more cost-effective method to achieve the perfect mutation score with a minimum test suite.

6.4.2 Mutation Analysis of the Obligations. Table 9 presents the results of obligation mutation. For the Law Firm policy, the obligation tests killed 60% of the non-equivalent obligation mutants. For the GPMS-NGAC policy, the mutation score is 87.2%. Therefore, the test suites of obligation coverage are inadequate for high assurance of obligations. In particular, the mutation scores related to three mutation operators AEO (add event operation), ROF (remove one factor), and NOF (negate one factor) are very low. They are shown in Table 10. ROF and NOF did not apply to the GPMS-NGAC policy. The obligation tests of the LawFirm case study killed 30.4% of the ROF mutants and 51.9% of the NOF mutants. Even worse, the obligation tests for LawFirm and GPMS-NGAC only killed 2.7% and 4.7% of the AEO mutants. This indicates that the obligation testing is poor at finding the faults represented by these mutants.

6.4.3 Discussion. Before applying mutation analysis to the initial configuration of a subject policy, we were sure that the initial configuration was correct. The all-combinations test suite has covered

Table 9: Results of Obligation Mutation Analysis

	#Mutants	#Equivalent Mutants	#Killed Mutants	Mutation Score(%)
LawFirm	238	13	135	60.0
GPMS	2,790	15	2,420	87.2

Table 10: Obligation Mutants with Low Mutation Scores

	AEO		ROF		NOF	
	#M	MS(%)	#M	MS(%)	#M	MS(%)
LawFirm	149	2.7	23	30.4	54	51.9
GPMS	64	4.7	-	-	-	-

#M: number of mutants; MS(%): mutation score

all possible requests, and the access decision for each request was proven correct. However, mutation analysis of the obligations is somewhat different. Before conducting mutation analysis of the obligations, we were confident, but not sure, that the obligations were correct. Although the obligation testing had exercised every obligation without failure, it was not exhaustive. An obligation mutant may be correct, whereas its original obligation is faulty per the actual access control requirements (i.e., whether they have changed the access decisions for certain requests). Thus, an examination into the obligation mutants may reveal faults in the original obligations for mutation.

7 CONCLUSIONS

We have presented the approach to mutation analysis of NGAC policies. It deals with various types of faults that may occur in the initial configurations and obligations of NGAC policies. The results of the empirical studies demonstrate that mutation analysis is useful for evaluating the effectiveness of a testing method and finding potential faults in an inadequately tested policy. The all-combinations method for testing initial configurations is effective but inefficient and hardly scalable, whereas the pairwise method cannot reveal many configuration faults. Testing obligations by covering each obligation once is inadequate for detecting the majority of obligation faults.

As the first report on testing NGAC policies, this paper opens up several interesting topics for future research. First, we need more cost-effective methods for testing the initial configurations of NGAC policies. Second, it is inefficient to determine equivalent configuration mutants by dealing with all access requests. A possible solution is to formulate the semantic difference between a configuration mutant and its original counterpart. Equivalence may be determined by considering only those requests relevant to the difference. Third, we need effective methods for testing obligations. Obligation tests can be created from either obligation implementation (i.e., white-box testing with the yml file) or obligation specification (i.e., black-box testing without knowledge of the yml file). Automated generation of obligation tests is challenging because of complex state space (i.e., many reachable configurations caused by obligations of complex policies).

ACKNOWLEDGMENTS

This work was supported in part by US National Science Foundation (NSF) under grant CNS 1954327 and grant CNS 1618229. The authors were grateful to the NGAC implementation team for their help in the design of the NGAC policy of GPMS-NGAC.

REFERENCES

[1] A. Bertolino, S. Daoudagh, F. Lonetti, and E Marchetti. 2013. XACMUT: XACML 2.0 mutants generator. In *Proc. of 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 28–33.
[2] W. Jansen (Ed.). 2020. *Next Generation Access Control (NGAC), DPANS INCITS 565, Revision 1.00.*
[3] Y. Elrakaiby, T. Mouelhi, and Y. Le Traon. 2012. Testing obligation policy enforcement using mutation analysis. In *Proc. of the 7th International Workshop on Mutation Analysis*. 673–680.
[4] FEDCIO2:. 2011. *Federal Identity, Credential, and Access Management (FICAM) Roadmap and Implementation Guidance Version 2.0.*
[5] V. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. 2013. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations, NIST Special Publication 800-162.*
[6] Y. Jia and M Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE Trans. on Software Engineering* 37, 5 (2010), 649–678.
[7] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, and G. Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proc. of the Symposium on the Foundations of Software Engineering (FSE'14)*. 654–665.
[8] S. Khamaiseh, P. Chapman, and D. Xu. 2018. Model-based testing of obligatory ABAC systems. In *Proc. of the 18th International Conference on Software Quality, Reliability and Security (QRS'18)*. 405–413.
[9] Y. Li, Y. Li, L. Wang, and G. Chen. 2014. Automatic XACML requests generation for testing access control policies. In *Proc. of the 26th International Conf. on Software Engineering and Knowledge Engineering (SEKE'14)*.
[10] E. Martin and T. Xie. 2007. A fault model and mutation testing of access control policies. In *Proc. of the 16th International Conf. on World Wide Web (WWW'07)*. 667–676.
[11] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon. 2008. A model-based framework for security policy specification, deployment and testing. In *Proc. of the ACM/IEEE 11th International Conf. on Model Driven Engineering Languages and Systems (MODELS'08)*.
[12] OASIS. 2013. *eXtensible Access Control Markup Language (XACML) Version 3.0.* http://www.oasisopen.org/committees/xacml/.
[13] A. J. Offut. 2011. A mutation carol: Past, present and future. *Information and Software Technology* 53, 10 (2011), 1098–1107.
[14] J. Park and R. Sandhu. 2002. Towards usage control models: beyond traditional access control. In *Proc. of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT'02)*. 57–64.
[15] M. Safarzadeh, M. Taghizadeh, B. Zamani, and B.T. Ladani. 2017. An automatic test case generator for evaluating implementation of access control policies. *The ISC International Journal of Information Security* 9, 1 (2017), 73–91.
[16] R.S. Sandhu, V. Bhamidipati, and Q. Munawer. 1999. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and Systems Security* 2, 1 (1999), 105–135.
[17] NGAC Implementation Team. 2019. *NGAC Reference Implementation.* Retrieved January 2021 from https://github.com/PM-Master/
[18] Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry. 2008. Test-driven assessment of access control in legacy applications. In *Proc. of the First IEEE International Conference on Software, Testing, Verification and Validation (ICST'08)*. 238–247.
[19] D. Xu. 2021. *Modern Software Engineering: Principles and Practices – Writing Clean, Dependable Code.* Independently Published.
[20] D. Xu, M. Kent, L. Thomas, T. Mouelhi, and Y. Le Traon. 2015. Automated model-based testing of role-based access control using predicate/transition nets. *IEEE Trans. Comput.* 64, 9 (2015), 2490–2505.
[21] D. Xu and S. Peng. 2016. Towards automatic repair of access control policies. In *Proc. of the 14th IEEE Conference on Privacy, Security and Trust (PST'16)*. 485–492.
[22] D. Xu, R. Shrestha, and N Shen. 2018. Automated coverage-based testing of XACML policies. In *Proc. of the 23rd ACM Symposium on Access Control Models and Technologies (SACMAT'18)*. 3–14.
[23] D. Xu, R. Shrestha, and N Shen. 2020. Automated strong mutation testing of XACML policies. In *Proc. of the 25th ACM Symposium on Access Control Models and Technologies (SACMAT'20)*. 105–116.
[24] D. Xu, L. Thomas, M. Kent, T. Mouelhi, and Y. Le Traon. 2012. A model-based approach to automated testing of access control policies. In *Proc. of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT'12)*. 209–218.

A A SAMPLE OBLIGATION TEST

```
@Test
public void chairDisapprove() throws Exception {
    PReviewDecider decider = new PReviewDecider(graph);
    //get PDP loaded
    PDP pdp = getGPMSpdp(graph, obligation);
    //create proposal
    assertTrue(decider.check("Bob", "", "PDSWhole", "create"));
    pdp.getEPP().processEvent(new CreateEvent(graph.getNode(PDSWhole)), "Bob", "
        ");
    assertTrue(graph.getChildren("PI").contains("Bob"));
    //add CoPi
    assertTrue(decider.check("Bob", "", "CoPI", "add-copi"));
    pdp.getEPP().processEvent(new AddCoPIEvent(graph.getNode("CoPI"), graph.
        getNode("Michael")), "Bob", "");
    assertTrue(graph.getChildren("CoPI").contains("Michael"));
    //add SP
    assertTrue(decider.check("Michael", "", "SP", "add-sp"));
    pdp.getEPP().processEvent(new AddSPEvent(graph.getNode("SP"), graph.getNode(
        "Alice")), "Michael", "");
    assertTrue(graph.getChildren("SP").contains("Alice"));
    //submit proposal
    assertTrue(decider.check("Bob", "", "PDSWhole", "submit"));
    pdp.getEPP().processEvent(new SubmitEvent(graph.getNode("PDSWhole"), true),"
        Bob", "");
    assertFalse(decider.check("Bob", "", "PDSWhole", "submit"));
    assertTrue(decider.check("ChairCSUser", "", "PDSWhole", "approve", "
        disapprove"));
    assertTrue(decider.check("ChairCSUser", "", "ChairApproval", "write"));
    assertTrue(decider.check("ChairChemUser", "", "PDSWhole", "approve", "
        disapprove"));
    assertTrue(decider.check("ChairChemUser", "", "ChairApproval", "write"));

    //chair disapproval
    pdp.getEPP().processEvent(new DisapproveEvent(graph.getNode(ChairApproval)),
        "ChairChemUser", "");
    assertFalse(decider.check("ChairCSUser", "", "PDSWhole", "approve", "
        disapprove"));
    assertFalse(decider.check("ChairCSUser", "", "ChairApproval", "write"));
    assertFalse(decider.check("ChairChemUser", "", "PDSWhole", "approve", "
        disapprove"));
    assertFalse(decider.check("ChairChemUser", "", "ChairApproval", "write"));
    //check the return to the initial state
    assertTrue(decider.check("Bob", "", "PDSWhole", "submit","delete"));
    assertTrue(decider.check("Bob", "", "PIEditable", "write"));
    assertTrue(decider.check("Bob", "", "CoPI", "add-copi", "delete-copi"));
    assertTrue(decider.check("Michael", "", "CoPIEditable", "write"));
    assertTrue(decider.check("Michael", "", "CoPI", "add-sp", "delete-sp"));
    assertFalse(decider.check("ChairCSUser", "", "PDSSections", "read"));
    assertFalse(decider.check("ChairChemUser", "", "PDSSections", "read"));
}
```