ProfessorX: Detecting Silent Vulnerabilities in Policy Engine Implementations

Ben Weintraub[*]
Northeastern University
Boston, MA, USA
weintraub.b@northeastern.edu

Chanyuan Liu[*]
Northeastern University
Vancouver, BC, Canada
liu.chany@northeastern.edu

William Enck
North Carolina State University
Raleigh, NC, USA
whenck@ncsu.edu

Cristina Nita-Rotaru
Northeastern University
Boston, MA, USA
c.nitarotaru@northeastern.edu

Abstract

Enterprises that own or handle sensitive resources rely on access control models to protect those resources. NIST has recently standardized a new access control model called Next Generation Access Control (NGAC) and provided a reference implementation. Despite the importance of properly functioning access control systems, little work has been done to verify that the software implementing NGAC properly conforms to the NGAC standard. Prior approaches for finding bugs are either designed to detect fail-stop faults, or model the protocol or software itself, which does not identify discrepancies between software and standards.

In this paper, we solve this problem with a methodology we call *policy engine differential mutation analysis*, which we implement in a system called ProfessorX. ProfessorX detects access decision discrepancies between policy engine implementations—specifically, the NIST reference implementation, and our own implementation of the standard. If there are no discrepancies, we mutate the policy slightly and then try again. Using this technique, we identified two novel vulnerabilities, and show that our system is fast enough to be practically useful to developers.

CCS Concepts

• **Security and privacy** → **Authorization**; • **Software and its engineering** → *Software verification and validation.*

Keywords

Differential Testing; NGAC; Mutation Analysis; Security

ACM Reference Format:
Ben Weintraub, Chanyuan Liu, William Enck, and Cristina Nita-Rotaru. 2025. ProfessorX: Detecting Silent Vulnerabilities in Policy Engine Implementations. In *Proceedings of the 30th ACM Symposium on Access Control Models and Technologies (SACMAT '25), July 8–10, 2025, Stony Brook, NY, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3734436.3734446

[*]Both authors contributed equally to the paper

This work is licensed under a Creative Commons Attribution International 4.0 License.

SACMAT '25, Stony Brook, NY, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1503-7/2025/07
https://doi.org/10.1145/3734436.3734446

1 Introduction

Access control systems are designed to restrict access of sensitive resources to only people who need such access to perform their jobs, and over the years, numerous access control models have been created and developed to help achieve these goals [6, 20, 21]. Of particular utility are attribute-based access control (ABAC) [25] and role-based access control (RBAC) [18] due to how attributes and roles relate to job functions within organizations. However, even these models constrain policy administrators—forcing them to structure their policies in ways that are representable in their chosen access control system [17].

The US government has a history of interest in access control models dating back at least to 1976 [5]. So it is no surprise that in response to these problems, the US National Institute for Standards and Technologies (NIST) developed and standardized a new enterprise access control framework in 2015 called Next Generation Access Control (NGAC) [17]. NGAC's key benefit is its ability to model and maintain compatibility with numerous alternative policy frameworks including ABAC, RBAC, and ACLs [16], and permit implementation of these models in centralized and distributed settings. As a result, NGAC is recommended by NIST as an authorization framework for deploying microservice architectures [11] and cloud-native applications in distributed environments [10].

Core to NGAC's specification is a mathematical model that defines how access decisions should be made in the presence of various types of rules. The model is a description of a finite state machine whose access decision function takes as input a request for access and outputs a decision to grant or deny access. NGAC consists of a number of policy elements and relations between them, and thus can be used to model ABAC, RBAC, and other types of policies.

NIST has provided a reference implementation of an NGAC-compatible *policy engine* [36] that provides access decisions for arbitrary NGAC policies. We call this implementation $\texttt{NGAC}_{\texttt{NIST}}$. However, the flexibility that is key to NGAC's utility also makes it complex to implement. Deviation from the specification could constitute a security-critical bug. For example, a deviation could be exploited to allow access to a resource in violation of a policy's intent. Prior work has considered the correctness of specific NGAC policies [12] and has used $\texttt{NGAC}_{\texttt{NIST}}$ for runtime enforcement [2, 3]. However, these all assume $\texttt{NGAC}_{\texttt{NIST}}$ is correct. While $\texttt{NGAC}_{\texttt{NIST}}$ contains unit tests, these tests are limited by the developers' creativity for building (or simulating) diverse program states.

Software bug finding is a robust field that includes methods such as fuzzing, formal methods, and program analysis. Fuzzing works best at detecting fail-stop faults but is limited when faults in the implementation are *silently* incorrect. Another approach, formal methods, are generally applied to protocols or specifications in the abstract [7, 24, 30], or used to directly develop verified programs using theorem provers [1, 26]. In cases where they are used to verify if code conforms to specification, they must be tightly coupled with the development workflow, a process that is both difficult and time consuming [29]. Static analysis techniques equally fall short. Approaches like symbolic execution can confirm the satisfiability of specific execution traces [4], but beyond finding unreachable code, these need to approaches require properties designed by experts to connect low-level inferences about program execution to the semantics of the specification. Another technique, points-to analysis, can be used to track the flow of specific variables and search for patterns in executed statements [27, 35], but again, these require expert-designed properties to compare the statement-level inferences to high-level expectations. The above pitfalls make these types of static analyses unsuitable for a context in which the precise execution of the program is less important than the values it returns.

In this paper, we propose a method of *differential mutation analysis* to investigate if $\texttt{NGAC}_{\texttt{NIST}}$ is operating correctly, and we implement it in the form of ProfessorX. At a high level, ProfessorX accepts a seed policy, and iteratively performs a series of semantically-valid mutations; after each mutation, it compares the access decisions made by $\texttt{NGAC}_{\texttt{NIST}}$ to the access decisions of a simplified policy engine model we implemented using logic programming. We call this simplified model $\texttt{NGAC}_{\texttt{Tiny}}$. Due to the structure of logic programming languages, we can implement NGAC's complex set theory-based mathematical model in more direct syntax. Our $\texttt{NGAC}_{\texttt{Tiny}}$ implementation is less than 100 lines of liberally spaced and heavily commented code—in comparison to the 33,949 lines of code in the NIST implementation. This much smaller implementation size gives us a great deal of confidence in its correctness.

The key insight of our approach is that policy engine implementations that conform to the NGAC specification should agree on all access decisions, and though there is only one well-known implementation of an NGAC-compatible policy engine (i.e., $\texttt{NGAC}_{\texttt{NIST}}$), it can still be tested by comparing it against a more trusted implementation (i.e., $\texttt{NGAC}_{\texttt{Tiny}}$).

Using ProfessorX, we discovered two discrepancies between $\texttt{NGAC}_{\texttt{NIST}}$ and $\texttt{NGAC}_{\texttt{Tiny}}$. We call these *association overwrite* and *scope leakage*. Both are novel, to the best of our knowledge. Association overwrite allows certain new policy changes to overwrite previously committed policy elements. This can allow an adversary to deny access to resources explicitly allowed in a policy. Scope leakage, on the other hand, causes access rights to be granted to peers outside the scope defined in the NGAC specification [17]. Our detection of both of these novel vulnerabilities suggests that ProfessorX can be useful for discovering silent failures in implementations of policy engine specifications.

Contributions. We make the following contributions:

(1) We model a simplified NGAC policy engine as a logic program, which can be more easily reasoned about than the full-scale reference implementation.
(2) We built ProfessorX, a differential mutation analysis framework to test the NIST implementation against our simplified model. Using this framework, we were able to uncover two vulnerabilities. We make our implementation open source for the benefit of the community[1].
(3) We show that ProfessorX can be used to quickly detect implementation deviations from the NGAC specification, and could be a useful addition to a developer workflow.

Ethics. We disclosed our findings to NIST and were assigned CVE-2025-31507.

2 NGAC Background

NIST developed the NGAC standard to meet the demands of globally distributed and interconnected business enterprises [19]. The structure of NGAC is based on both the *attribute-based access control* (ABAC) and *role-based access control* (RBAC) models. NGAC's structure allows it to use node characteristics and properties to define and manage access control policies [11].

2.1 NGAC Model

Access control is essential for all enterprises that wish to protect sensitive information or hardware. An enterprise may be a business, university, government, or other organization interested in protecting resources it owns. The details of resource protection are governed by the security *policies* (P) that make explicit who is able to access what resource. We call the person who defines the policy a *policy administrator*. NGAC demarcates a number of different policy elements and relationships that can be used as building blocks to formulate arbitrarily complex policies. We discuss the policy elements and relationships in turn.

2.1.1 Policy Elements.

The basic data elements of an NGAC policy include users, processes, user attributes, objects, object attributes, policy classes, operations, and access rights. The combined set of users, user attributes, objects, object attributes, and policy classes form the set of *policy elements* (PE). In NGAC, the sensitive resources are called *objects* (O), while the entities who want to access those resources are called *users* (U). However, only processes ($Proc$) can actually access resources; the system is designed such that authenticated users must initiate access requests through a process that accesses the object on the user's behalf. Both users and objects can be tagged with any number of attributes—these are called *user attributes* (UA) and *object attributes* (OA), respectively. The semantics of an attribute can be decided at the discretion of the policy administrator; either type of attribute could represent, e.g., clearance levels, geographic locations, and organization departments. Notably, NGAC regards the set of objects as a subset of the object attributes, $O \subseteq OA$, though the same does not apply to users and user attributes, $U \nsubseteq UA$. The affiliation of certain users, user attributes, and object attributes with a policy is denoted as a *policy class* (PC). There can be multiple policy classes in an NGAC policy and policy elements can be part of multiple policy classes. A policy administrator can define a set of access *access rights* (AR) in a given policy and one or more access rights are required to execute an *operation* (Op).

[1]https://doi.org/10.5281/zenodo.15388891

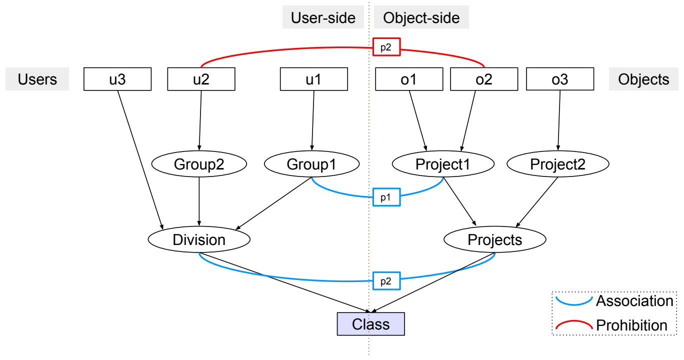


Figure 1: An example policy graph DAG. This example includes some illustrative groupings of both users (in the form of divisions and groups), and objects (in the form of hierarchical projects). In this example, the user attribute "Division" is granted access right $p2$ to object attribute "Projects".

2.1.2 *Relations.* NGAC defines the types of relationships that can exist between policy elements. The types of relations are assignments, associations, prohibitions, and obligations (see Appendix C).

Policy Graph. In NGAC, policy administrators create representations of organizational structures including personnel, sensitive resources, and groupings of both. The codified organizational structure comprises a *policy graph*, which is a directed acyclic graph (DAG) connecting policy elements. An example policy graph can be seen in Figure 1, and is generally visualized as an upside-down n-ary tree, by convention. All policy elements constitute the policy element set PE, i.e., $PE = U \cup UA \cup OA \cup PC$, where each of U, UA, O, OA, and PC is a finite set of its respective type of policy element. Additionally, AT is a finite set of all attributes, i.e., all user attributes and all object attributes, $AT = UA \cup OA$.

Assignments. Directed edges between pairs of nodes in the policy DAG are called *assignments*. An assignment relation between two policy elements x and y, denoted as $(x, y) \in ASSIGN$, represents a directed edge that originates at x and terminates at y. When there is a path through the DAG from policy element x to y, we say (in alignment with the NGAC specification) that x is "contained by" y, and denote this notationally as $x \rightsquigarrow y$. Specific constraints define which policy elements types can be assigned to each other. The specific constraints are represented formally as: $ASSIGN \subseteq (U \times UA) \cup (UA \times UA) \cup (OA \times OA) \cup (UA \times PC) \cup (OA \times PC)$.

Associations. Policy administrators grant users privileges to access objects through *associations*, which are the downward blue arcs in Figure 1. Each policy class has a set of ternary relations $(ua, ars, at) \in ASSOCIATION$, for $ua \in UA$, $ars \subseteq AR$, and $at \in AT$. The association applies to all users and user attributes contained by ua, $u \rightsquigarrow ua \cup ua' \rightsquigarrow ua$, as well as all attributes contained by at, $ua' \rightsquigarrow at \cup oa \rightsquigarrow at \cup o \rightsquigarrow at$, for $o \in O$.

Prohibitions. Policy administrators explicitly deny users access to objects through *prohibitions*; these are represented as the upward red arcs in Figure 1. Prohibitions must be provided as input both an inclusion set of policy elements, $ati \in PE \backslash PC$, and an exclusion set of policy elements, $ate \in PE \backslash PC$. Additionally, they can be both disjunctive or conjunctive. A disjunctive *in*clusion set defines the attributes for which a request will be denied if *any* are the target of the access request, $at \in ati$. A disjunctive *ex*clusion set defines the attributes for which an access request will be denied if the attribute is not contained by at least one element in the exclusion set. Conversely, a prohibition can be *conjunctive*. In this case, a conjunctive *in*clusion set defines the attributes for which a request will be denied if the target is contained by *all* elements of the set, $at \in ati$. A conjunctive *ex*clusion set defines the the attributes for which an access request will be denied if the attribute is not contained by all attributes the exclusion set.

Decisions. All of the above policy components play a role in the outcome of the access request mediation. That is, ultimately, any access request must be sent to some endpoint that is capable of boiling the above components down into a single decision. We model this endpoint as the function $\texttt{decision}(u, ar, at)$ for user u, access right ar, and attribute being accessed at.

2.1.3 *NIST Reference Implementation* $\texttt{NGAC}_{\texttt{NIST}}$. The reference implementation $\texttt{NGAC}_{\texttt{NIST}}$ is written in Java and has 33,949 lines of code, with four main packages and 455 unit tests, as of the latest version. The policy information stored internally is compiled from policy statements written in Policy Machine Language (*PML*) [36]. PML is a domain-specific language for defining policies accepted by $\texttt{NGAC}_{\texttt{NIST}}$. It includes provisions for defining access rights, creating policy elements, and building relations between policy elements. Each of these has a regular syntax. Sample policies are included in our open sourced implementation. Since the first release in 2019, $\texttt{NGAC}_{\texttt{NIST}}$ has been under active development, with the latest version released in 2024.

2.2 Related Work

Differential testing is widely used in discovering semantic bugs that cause deviations from high-level specifications in software implementations. They have been used for discovering evasion vulnerabilities in email systems [38], testing the fork-handling behavior in blockchain implementations [28], and SSL/TLS certificate validation [14]. However, the above approaches are limited to specific domains, none of which include NGAC policy mediation or policy engines, in general.

Petsios et al. [32] designed Nezha as a domain-independent, input format-agnostic, differential testing framework. While this could, in theory, be applied to NGAC, it would not be effective in practice. This is because they require either instrumentation into the codebase—a difficult hurdle for large projects—or they diverse program outputs to guide their inputs—which NGAC does not have (it only outputs a binary allow or deny).

In addition, none of the above projects consider inputs with more complex semantics. An NGAC policy may look like a DAG, but it has constraints that mean it cannot be arbitrarily modified and still be assumed to be valid input. This semantically valid input format is something that Petsios et al. [32] cannot detect.

ProfessorX uses mutation analysis to generate valid inputs for testing. Chen et al. [12] also used mutation analysis in a project on NGAC policies, however they used mutations to detect errors in the policy. Mutation analysis, however, has been around since at least 1980 [9], so our main novelty with regards to mutation analysis is to apply it to NGAC policies. Further novelty in ProfessorX comes from the application of mutation analysis to differential testing—especially in the context of NGAC policy engines.

3 NGAC Modeling

The flexibility of NGAC is a double-edged sword; it simultaneously makes NGAC suitable as a model for many organizational structures and arrangements, while at the same time demanding a lengthy specification to (119 pages) to cover all edge cases, and commensurately complex implementations.

Our goal in modeling NGAC is to establish a more limited scope for what types of policies and access requests a policy engine needs to provide mediation for. The reason for this is that due to the potential complexity allowed in NGAC policies, the proper functioning of a policy engine implementation can be difficult to reason about. The NIST reference implementation $\texttt{NGAC}_{\texttt{NIST}}$ is a large project and contains many program constructs that allow for modular design and efficient decision making, but come at the cost of a system whose behaviors may be difficult to understand. Our simplified policy engine implementation $\texttt{NGAC}_{\texttt{Tiny}}$ that we describe in Section 3.4, is much less likely to contain bugs or deviations from the specification, because it is much smaller in scope (less than 100 lines of code), and therefore easier to reason about. Note that performance of $\texttt{NGAC}_{\texttt{Tiny}}$ is a non-goal, and as $\texttt{NGAC}_{\texttt{Tiny}}$ is written in Prolog, it has significant overhead that is not strictly necessary for a system trying to optimize for speed.

3.1 NGAC Scope

We start from the premise that $\texttt{NGAC}_{\texttt{Tiny}}$'s interface for mediating access requests must be similar to the interface for $\texttt{NGAC}_{\texttt{NIST}}$. Namely, $\texttt{NGAC}_{\texttt{Tiny}}$ must accept access requests in the form of a triple containing the user, access right, and object. It should then process the request in accordance with the rules of NGAC and the installed policy, and output an access decision, i.e., grant or deny.

$\texttt{NGAC}_{\texttt{Tiny}}$ reduces the scope of the policy engine by limiting four NGAC constructs. First, in $\texttt{NGAC}_{\texttt{Tiny}}$ we do not include the process and operation policy components; this is consistent with prior work [19], which states that the core computation of the NGAC framework is to adjudicate the relationships between users, objects, and access rights. Second, we greatly simplify the prohibition functionality in our model. In particular, we model only disjunctive prohibitions, because modeling conjunctive prohibitions quickly increases the complexity of our modeling—thus sacrificing the simplicity that makes our model valuable. We also assume a default exclusion set which includes all policy classes. The implications of this are that when an object attribute is specified as the target of the prohibition, that element (and anything contained by it) is the only element to be prohibited. Again, this is useful for reasoning about results and outputs. That said, there is no technical reason why it would not be possible to model prohibitions in their entirety.

3.2 Threat Model

The NGAC-compatible policy engine is at the core of the functional security security architecture, as shown in Figure 7. Any and all access requests to resources are mediated by the this policy engine. Any bugs in the implementation of the policy engine can lead to unauthorized access to resources or denial of access to legitimate users.

The goal of our adversary is to access sensitive resources that they have not been granted privilege to access, or to block another user from accessing resources that they should have access to. The adversary could be anyone with access to a device within the perimeter of the organization. This could include a member of the organization who is trying to access resources beyond their mandate, or an external adversary who has gained control of a system via social engineering or the exploitation of vulnerabilities.

We assume the adversary can only use the authenticated identity corresponding to the device they are operating from; they cannot spoof any other identities and they cannot move laterally to any device that has a different identity. All of the adversary's access requests must be done using this authenticated identity. The adversary can change or influence the organization's NGAC policy, but any attempt to do so may be detectable if it is too blatant. Therefore, the adversary will want to act surreptitiously.

3.3 Modeling NGAC Policies

NGAC has a long and complex specification. In addition to textual descriptions of the model, the specification also includes detailed set syntax describing the elements of a policy and how decisions should be made. From this perspective, the access request mediation process can be viewed as a sequence of set unions and intersections on both the information defined in the policy, as well as the information provided in the request.

This structure lends itself to logic programming, which includes natural syntax for reasoning about set logic. We select the logic programming language Prolog as it has been used in many prior works to model access control [13, 15, 22, 23, 34]. In this logic programming model, we define a policy as a set of Prolog *facts*. For facts representing policy elements, we use "name" to represent the label of each respective policy element. These policy elements are: users `u(name)`, user attributes `ua(name)`, objects `o(name)`, object attributes `oa(name)`, policy classes `pc(name)`, and access rights `ar(name)`.

For the NGAC relations, we model these as *n*-arity facts. In the following we show interface of these facts for assignments, associations, and disjunctive prohibitions, respectively.

```
assign(username, user_attribute).
association(user_attribute, object_attribute,
                            [permission_type]).
disjunctiveProhibition(user, [object_attribute],
                             [permission_type]).
```

3.4 Implementing $\texttt{NGAC}_{\texttt{Tiny}}$

We build $\texttt{NGAC}_{\texttt{Tiny}}$ out of a series of Prolog *rules*. A rule is a Horn Clause that represents a logical relationship between facts. Ultimately, our Prolog modeling results in a *decision* rule that we can use to evaluate access decisions in ProfessorX. However, this decision rule is built from smaller components. Below we present snippets of $\texttt{NGAC}_{\texttt{Tiny}}$, the full implementation can be viewed in Appendix D.

Assignments. To check that an assignment exists and is legal, according to the NGAC specification, we use the following Prolog rule. This rule both checks that an assignment exists from X to Y and that the assignment is between one of the allowed pairs of policy element types.

```
    legalAssignment(X, Y) :-
        assign(X, Y),
        ((u(X), ua(Y)); (ua(X), ua(Y));
         (o(X), oa(Y)); (oa(X), oa(Y))).
```

Containment. For implementing the "contained by" relation, we wrote a rule to perform a depth-first search. This rule says that X is contained by Y if (1) X and Y are the same element, (2) there is an assignment from X to Y, or (3) there is an assignment from X to Z with Z being contained by Y.

Associations. Our association rule checks that both an association fact exists in the policy between policy elements that can legally be associated to each other, and that the access rights being granted have been defined for the policy.

```
legalAssociation(UA, OA, ARS) :-
    association(UA, OA, ARS),
    ((ua(UA), ua(OA), legalAccessRights(ARS));
     (ua(UA), oa(OA), legalAccessRights(ARS));
     (ua(UA), o(OA), legalAccessRights(ARS))).
```

Prohibitions. Our implementation only includes disjunctive prohibitions. For this reason, we need to check that (1) a disjunctive prohibition fact is defined in the policy along with the reference access rights, (2) the target of the prohibition is one of the policy elements allowed to be prohibited, (3) the user who is accessing the target attribute is contained by the user attribute defined in the prohibition, and (4) the target attribute is in the inclusion set—the set of object attributes whose access is prohibited—or that the attribute is contained by an attribute in the inclusion set.

```
disjProhibited(U, AT, AR) :-
    disjunctiveProhibition(U_or_UA, ATI, ARS),
    legalAccessRights(ARS), member(AR, ARS),
    (ua(AT); oa(AT); o(AT)),
    isContained(U, U_or_UA),
    inInclusionSet(AT, ATI).
```

Decisions. We define a decision rule that checks if a user has explicitly been granted the right to access the target policy element, *and* that no prohibition has been defined precluding such access. The negation operator in Prolog is `\+`.

```
decideAll(U, PE, AR) :-
    legalAssociation(UA, PE_Parent, ARS),
    member(AR, ARS),
    isContained(U, UA),
    isContained(PE, PE_Parent),
    \+ (isContained(PE, PE_Prohib),
        disjProhibited(U, PE_Prohib, AR)).
```

4 Policy Engine Validation

We describe ProfessorX, our system for finding access decision differences between the NIST implementation $\mathtt{NGAC}_{\mathtt{NIST}}$ of the NGAC-compatible policy engine and our simplified policy engine $\mathtt{NGAC}_{\mathtt{Tiny}}$.

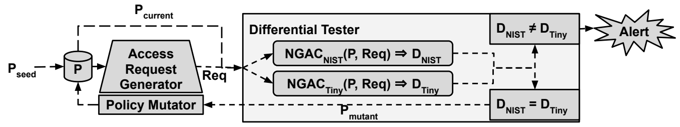


Figure 2: ProfessorX data flow pipeline. Policies are fed into the Access Request Generator module that sends generated requests to the policy engines. The Differential Tester compares D_{NIST} and D_{Tiny} which represent the set of decisions made by $\mathtt{NGAC}_{\mathtt{NIST}}$ and $\mathtt{NGAC}_{\mathtt{Tiny}}$, respectively. If an alert is not generated, the policy is mutated and the process restarts.

4.1 Overview

ProfessorX is designed to assess how closely $\mathtt{NGAC}_{\mathtt{NIST}}$ follows the NGAC specification. In particular, we aim to detect any behavioral deviations from the specification. The process starts by feeding a manually created *seed policy* P_{seed} into the Access Request Generator (ARG) and the Differential Tester module which will forward the policy to the $\mathtt{NGAC}_{\mathtt{Tiny}}$ and $\mathtt{NGAC}_{\mathtt{NIST}}$ policy engines. The ARG builds a set of access requests *Req* based on the users, access rights, and accessible objects defined in the policy. Like the seed policy, the access requests are fed into the Differential Tester, in which the policy engines output their decisions based on their respective $\mathtt{decision}_{\mathtt{Tiny}}(u, ar, at)$ and $\mathtt{decision}_{\mathtt{NIST}}(u, ar, at)$ procedures. The Differential Tester then determines if there is a disagreement about whether the access request should be allowed or denied. If there is a discrepancy, then the Differential Tester generates an alert. Otherwise, the policy is fed into the Policy Mutator module. This module creates a single change in the policy by adding exactly one policy element or relation; this results in a new policy P_{mutant}. The process then begins again in a new round with P_{mutant} being fed into the policy engines and the ARG module in the place of P_{seed} and continues until either a discrepancy is found or the predetermined maximum number of rounds R_{max} is reached. A diagram of ProfessorX's operation can be viewed in Figure 2.

4.2 ProfessorX

At its core, ProfessorX compares the outcome of access requests in $\mathtt{NGAC}_{\mathtt{NIST}}$ to the same access requests in $\mathtt{NGAC}_{\mathtt{Tiny}}$. In this context, there are two major considerations to make. First, what policy should be used when comparing the two models? And second, which access requests should be tested?

4.2.1 Input Policies. The policies that are fed into the respective policy engines are essential to the proper functioning of ProfessorX. A naïve approach might input randomly generated policies. However, this approach is subject to two pitfalls. The first is that if a policy is structurally invalid then we are unlikely to find discrepancies between the two models, because simple software development safeguards like type-checking are likely to catch these. The second reason is that if policies are unrealistic, it is not clear how meaningful detected discrepancies are.

For these reasons we take a two-pronged approach to generating input policies. We start by manually creating a seed policy. We can ensure this structure is well-formed, meaning it has a top-level policy class with object and user attributes above it, and objects and users, respectively, above the attributes. In addition, it can contain some associations and prohibitions.

The second prong in our approach answers the question of how to leverage a valid seed policy to generate more valid test policies. Our solution uses *mutation analysis*. We start with a valid policy, then create one atomic, semantically valid mutation, then pass that back to the respective policy engines to restart our differential testing. We discuss the details of mutations more in Section 4.2.4.

4.2.2 Differential Testing. Testing a policy involves generating a series of access requests, and sending those to the respective $\texttt{NGAC}_{\texttt{Tiny}}$ and $\texttt{NGAC}_{\texttt{NIST}}$ models. The composition of the set of possible access requests depends on the policy being tested. It only makes sense to check if a user u can perform access right ar on object o, if all three of u, ar, and o are defined within the policy being tested. However, even given this, it is not obvious which of the possible access requests is most likely to uncover a discrepancy between the models. For this reason, we opt to execute all possible access requests that can be made for a given policy. This means that for a policy P that includes users U, access rights AR, and accessible attributes AT, we perform the following: $decision_{NIST}(u, ar, at) \wedge decision_{Tiny}(u, ar, at)$ where $\forall u \in U, \forall ar \in AR, \forall at \in AT$.

The implication of this is that we are able to ignore any bugs that may exist in $\texttt{NGAC}_{\texttt{NIST}}$, unless it has a direct, observable impact on an access request mediation outcome. This again makes our testing procedure more practical than other analysis techniques.

4.2.3 Cyclic Testing and Alerts. While performing differential testing on a single policy has value, it does not take full advantage of the exploration made possible by our policy mutations. Additionally, for a single policy, we are testing all possible access requests, so there is nothing to be gained by performing a differential test more than once. For these reasons, we perform our tests cyclically. Every time we evaluate $\texttt{decision}_{\texttt{Tiny}}(u, ar, at)$ vs. $\texttt{decision}_{\texttt{NIST}}(u, ar, at)$, we can expect one of two outcomes. Either the two decisions will be the same (i.e., both allow or both deny), or they will be different. If they are different, we generate an alert and halt the program. If they are the same, then make a mutation to the policy, load the mutated policy into both the policy engines, and restart the differential testing. The reason we stop after a discrepancy occurs is that not only will we continue to see that same issue flagged repeatedly, but also we will see new discrepancies flagged that have the same root cause but are simply built on the policy elements involved in the previously detected discrepancy. We bound the number of rounds we will perform before stopping at R_{max}.

4.2.4 Mutations. Two key principles guide our mutations: (1) the mutated policy must be syntactically valid, and (2) the mutated policy must be semantically different from the pre-mutation policy.

Syntactically-valid Mutations. ProfessorX is designed to explicitly avoid detecting fail-stop faults. Our goal is instead to generate well-formed inputs that result in well-formed outputs. Therefore, it is not suitable to add or remove nodes and edges at random. This could result in policies that are syntactically invalid and would not be processed by either policy engine. For this reason, we place particular emphasis on generating NGAC policies that are syntactically valid, i.e., the guiding constraints of NGAC policies are observed.

We find that enforcing a single constraint can be used to ensure that generated policies are semantically-valid. The constraint is that the mutation must not create a cycle in the DAG. If we detect that a cycle would be created, we discard that mutation and select another. Concretely, this constraint is enforced by checking if a node x is already contained by another node y, and if it is, we do not add an assignment $(x, y) \in ASSIGN$, which would create a cycle. Since the "contained by" relation is transitive, this naturally avoids transitive cycles, e.g., $x \rightsquigarrow y \wedge z \rightsquigarrow x \wedge y \rightsquigarrow z$.

Semantically-aware Mutations. We also require mutations to be semantically aware to ensure we do not test the same policy multiple times. For example, assigning a user or an attribute element to its affiliated policy class does not change the policy engine's behavior—we avoid such changes.

Our mutations are informed by the NGAC specification, which defines administrative commands describing ways in which a policy can be modified [19]. We can use these administrative commands to guide our mutation process. Table 1 lists the valid types of mutations implemented in our approach. The mutations listed in Table 1 are a subset of the administrative commands defined in the NGAC specification. For the mutations that add new policy elements, we also must add a corresponding assignment otherwise the new element will be disconnected from the graph and not impact the effective policy at all. Incremental mutations also help us guarantee the mutated policy is semantically different from the previous policies. A policy, as described in Section 2.1, consists of policy elements and relations. If we mutate the set of policy elements or relations following the pre-conditions required by the mutation, we can be sure that the post-mutation policy is valid.

Note that we model only additive mutations. This is because removing nodes from the policy graph could disconnect the graph, and while reconnecting the graph is an option, it would invite new questions as to where to reconnect.

Selecting Mutations. In Table 1, we established the corpus of mutations that we consider. However, each round we only perform one mutation from the list, and so we must define a decision process for selecting mutations. This is a problem similar to *guidance* in fuzzing, where program inputs are guided by developer-chosen metrics that might lead towards more interesting program states. However, it is not obvious what types of program states might lead to a policy mediation being incorrect, so the lessons of fuzzing guidance do not prove useful here. As a result, we opt to select mutations randomly from the list. Our selection process works as follows. First, we select one of the four classes of mutations: adding a node, adding an association, adding an assignment, and adding a prohibition. If the random selection results in adding a node, we further randomly select the type of node, such as user, object, or attribute. We also need to reconnect the node with the graph with an assignment. To do this, we randomly select a viable end point, i.e., only elements that can legally contain the newly created element. If an assignment, association or prohibition is to be added, then we select, in a similar fashion, viable endpoints. We also randomly select an access right from the set defined for the policy; this is the access right that will be allowed or denied.

Table 1: Valid types of mutations implemented in our policy engine fuzzing process. ID is the item identifier in the policy.

Mutation	Description	
`CreateUinUA(x:ID,y:ID)`	Add policy element	Add user `x` to the policy representation and assign it to user attribute `y`
`CreateUAinUA(x:ID,y:ID)`	Add policy element	Add user attribute `x` and assign it to user attribute `y`
`CreateUAinPC(x:ID,y:ID)`	Add policy element	Add user attribute `x` and assign it to policy class `y`
`CreateOinOA(x:ID,y:ID)`	Add policy element	Add object `x` and assign it to object attribute `y`
`CreateOAinOA(x:ID,y:ID)`	Add policy element	Add object attribute `x` and assign it to object attribute `y`
`CreateOAinPC(x:ID,y:ID)`	Add policy element	Add object attribute `x` and assign it to policy class `y`
`CreateUserProhib(w:ID,x:ID,y:ID)`	Add relation	Add prohibition restricting user `x` from resource `z` with access right `y`
`CreateAssoc(x:ID;y:ID;z:ID)`	Add relation	Add association allowing user `x` access to resource `z` with access right `y`

Algorithm 1: Differential mutation analysis algorithm

```
Input: P: seed policy, $R_{max}$: max rounds
Output: true ∪ false
1  $Req \leftarrow GenReq(P)$  // generate exhaustive access requests
2  $i \leftarrow 0$ // iteration counter
3  while $i < R_{max}$ do
4      while All $r \in Req$ not processed do
5          $D_{NIST} \leftarrow NGAC_{NIST}(P, r)$
6          $D_{Tiny} \leftarrow NGAC_{Tiny}(P, r)$
7          $Log(P, D_{NIST}, D_{Tiny}, r)$
8          if $D_{NIST} \neq D_{Tiny}$ then
9              $GenerateAlert(r)$
10             return true
11         end
12     end
13     $P \leftarrow Mutate(P)$
14     Increment $i$
15 end
16 return false
```

4.2.5 Seed Policies. The first step in a run of ProfessorX is to initialize it with a seed policy. The main factor we consider when creating a seed policy is the total number of policy elements. We use the total number of policy elements as a proxy for the complexity of the policy. A more complex policy indicates more reliance on the policy engine's ability to mediate access decisions. By starting with a complex policy, we can guarantee that all runs will start with a minimum complexity. In addition, since we manually created the policy, we can ensure that it has a structure such as could be found in an enterprise setting. Conversely, a simple policy with fewer elements can be useful as a trusted starting point from which we can let the Policy Mutator build a variety of different structures.

4.3 Differential Mutation Analysis Algorithm

We present the algorithm used in ProfessorX for comparing the decisions of $\text{NGAC}_{\text{NIST}}$ and $\text{NGAC}_{\text{Tiny}}$ starting with a seed policy (see Section 4.2.5) and executing a series of mutations on the seed policy to find discrepancies between the two policy engines.

Algorithm 1 shows the pseudocode of our approach, where the input P is the seed policy and R_{max} is the maximum number of rounds. First, the algorithm generates an exhaustive set of access requests Req based on P using $GenReq$ (line 1). $GenReq$ calculates the cross-product of all possible users (and user attributes), objects (and object attributes), and access rights. The intuition behind this is that each access request $r \in Req$ may be impacted differently by a mutation—something we cannot easily predict. By evaluating every possible r, we cover the entire space of decisions that could be impacted by any given mutation. For evaluating an access request r on policy P, we write $NGAC_x(P, r)$, where x indicates the policy engine model used. The algorithm iterates over all access requests $r \in Req$ (line 4), querying each policy engine for access decisions (lines 5-6), and comparing the decisions D_{NIST} and D_{Tiny} of $\text{NGAC}_{\text{NIST}}$ and $\text{NGAC}_{\text{Tiny}}$, respectively (line 7). If $D_{NIST} \neq D_{Tiny}$, an alert is generated by $GenerateAlert$ function, the discrepancy is logged, and the algorithm returns `true` (lines 8-10). If there is no discrepancy, we mutate the policy P according to the rules in Table 1 (line 14) and then start back at the beginning of the loop (line 3) unless the maximum number of rounds R_{max} is surpassed.

4.4 Implementation

As discussed in Section 3.4, our policy engine model $\text{NGAC}_{\text{Tiny}}$ was implemented in Prolog. The rest of ProfessorX, however, is implemented in Java. Java was selected because it is the language that $\text{NGAC}_{\text{NIST}}$ is written in, so using the same language made interfacing between the two much easier. However, to interface with $\text{NGAC}_{\text{Tiny}}$, we relied on a library called JPL [37], which provides bindings for Java programs to use and reference Prolog facts and rules.

Policy Representation. Not only are the two policy engines $\text{NGAC}_{\text{Tiny}}$ and $\text{NGAC}_{\text{NIST}}$ written in different languages, but the policies that they can accept as inputs are also written in different languages. For $\text{NGAC}_{\text{Tiny}}$, the policies must be written in Prolog; for $\text{NGAC}_{\text{NIST}}$, they are written in PML. To ease implementation, we created a unified representation of the each policy in the form of a graph using JGraphT [31], a Java library that provides utilities for graph manipulation and analysis. This graph representation is critical for our ability to perform principled mutations on the policies.

Mapping Graph-based Policies to Prolog and PML Policies. While a graph representation is useful for mutations, eventually the mutated policies need to be reloaded into the respective policy engines. To satisfy the input requirements of the policy engines, we perform a breadth-first search of the policy graph, and for each policy element and relation we encounter, we convert that into both Prolog facts and PML statements. This ensures that the two policy engines receive the same policy in their native languages. Similarly, when we modify the graph, by remapping the new graph into PML and Prolog facts, the two updated files also represent the same policy design. As we do not model obligations or the more expansive definition of prohibition, the transpilation process can use simple string interpolation on standard policy element templates.

There is one challenge associated with this strategy, however. Namely, in PML, policy elements are processed sequentially, and therefore the order policy elements are declared in matters. Specifically, any policy element referenced in a policy relation must already have been defined. For instance, $\texttt{NGAC}_{\texttt{NIST}}$ will not accept an assignment between non-existing attributes—even if they are defined later in the same policy. As a result of this, when transpiling the policy graph to PML, we adopt a topological sort algorithm to ensure the order of creating policy elements is correct.

4.4.1 Seed Policies. We manually created two types of seed policies, simple and complex—each with their own benefits.

Simple Seed Policy. This policy contains very few policy elements, with only two users, two user attributes, one object, one object attribute, two associations, and one prohibition as shown in Figure 3a. The purpose of this policy is to serve as a starting point to add mutations to and to simplify reasoning in case a discrepancy arises.

Complex Seed Policy. This policy increases the number of policy elements versus the simple seed policy, and makes the number of objects (six) much greater than the number of users (one). We present a graphical representation of this policy in Appendix B (Appendix B). The reason for the disparity between users an objects is performance—it reduces the number of access requests that need to be checked. We are generating all possible resource accesses, which have a size equal to the size of the cross product between the set of users and the set of resources that can be accessed, $U \times AT$. If both of these quantities increase linearly, then $U \times AT$ grows quadratically, but if only one increases, then the growth of the total is only linear. By keeping the number of users at only one, we are drastically reducing the number of access requests we need to test. Policies like this are realistic, however. For example, Ferraiolo et al. [16] suggest this type of policy can be used to model Brewer and Nash's "Chinese Wall" policy [8].

4.4.2 $\texttt{NGAC}_{\texttt{NIST}}$ *Version.* Parts of this project started in 2022 stemming from a previous work Anjum et al. [3]), and as a result, evaluations began on a 2022 release of $\texttt{NGAC}_{\texttt{NIST}}$. We eventually migrated our evaluations to the latest version of $\texttt{NGAC}_{\texttt{NIST}}$, from 2024. We differentiate them by writing $\texttt{NGAC}^{22}_{\texttt{NIST}}$ and $\texttt{NGAC}^{24}_{\texttt{NIST}}$ for the old and new versions, respectively. We present evaluations primarily for $\texttt{NGAC}^{24}_{\texttt{NIST}}$, however we discuss in Section 5.4 discrepancies we discovered using $\texttt{NGAC}^{22}_{\texttt{NIST}}$.

5 Evaluation

We evaluate ProfessorX on its effectiveness of detecting $\texttt{NGAC}_{\texttt{NIST}}$ deviations from the NGAC specification as well as on performance metrics that have implications on the practicality of ProfessorX for deployment in real development workflows. Concretely, we aim to answer the following questions:

Q1 How effective is ProfessorX at detecting discrepancies between the $\texttt{NGAC}_{\texttt{Tiny}}$ and $\texttt{NGAC}_{\texttt{NIST}}$?

Q2 Is ProfessorX efficient enough to be used in a developer workflow?

Q3 Do discrepancies uncovered by ProfessorX have meaningful security implications?

We conducted our experiments on an Apple 2021 MacBook Pro with an M1 Pro processor and 16GB RAM. ProfessorX was run within a Ubuntu 20.04 Docker container on the host machine. For each seed policy, we ran ProfessorX 1,000 times, and in each run, ProfessorX mutated the seed policy 100 times—testing for discrepancies between each mutation—until a discrepancy was found or 100 rounds was reached.

5.1 Finding Discrepancies (Q1)

Out of 1,000 runs for each of the two seed policies for $\texttt{NGAC}^{24}_{\texttt{NIST}}$, ProfessorX found discrepancies in 796 runs that started with the simple seed policy and 567 runs for the complex seed policy. For $\texttt{NGAC}^{22}_{\texttt{NIST}}$, out of 1,000 runs, 998 and 856 runs ended with discrepancies for the simple seed policy and complex policy, respectively. Underlying each discrepancy is a misunderstanding or incorrect implementation of the NGAC standard. Upon triggering a discrepancy, we cannot say as a matter of course which policy model, $\texttt{NGAC}_{\texttt{Tiny}}$ or $\texttt{NGAC}_{\texttt{NIST}}$, is correct, and we perform manual analysis. Our manual analyses of all these discrepancies could be traced to two root causes—*association overwrite* and *scope leakage*.

For $\texttt{NGAC}^{24}_{\texttt{NIST}}$, all 796 runs that resulted in inconsistencies starting from the simple seed policy triggered association overwrite—for the complex seed policy, all 567 inconsistent runs also triggered the association overwrite. For $\texttt{NGAC}^{22}_{\texttt{NIST}}$, among the 998 inconsistent runs starting with simple seed policy, 235 runs triggered association overwrite, and 763 runs triggered scope leakage. The complex seed policy yielded 856 inconsistent runs with 755 runs triggering association overwrite, and 101 runs triggering scope leakage.

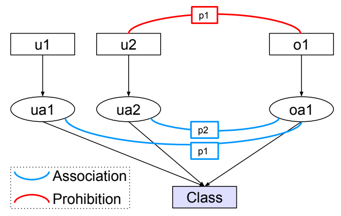
Additionally, choosing a proper seed policy appears to be important for finding discrepancies. Of our 1,000 runs starting with each of the simple and complex seed policies on $\texttt{NGAC}^{24}_{\texttt{NIST}}$, we detected discrepancies in 79.6% and 56.7% of the runs, respectively.

We cannot say for certain what factors make the best seed policies, but some notable differences between our two seed policies that could have impacted the detection rate are: policy size or policy structure. In any case, if we assume a likelihood of detecting a discrepancy to be as low as 50%, then there is a 99.9% probability of detecting a discrepancy after only ten runs.

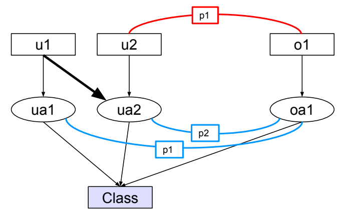
5.2 ProfessorX Performance (Q2)

We consider two relevant benchmarks: how long it takes to execute a single mutation-decision round, and how long it takes to execute enough rounds to make finding a discrepancy highly likely. An understanding of these will give a notion of how much time a user running ProfessorX can expect to experience.

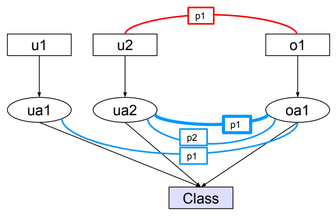
The time it takes for ProfessorX to execute a single mutation-execution round increases with the size of the policy. In Figure 4, we see that the runtimes are increasing superlinearly with the number of mutations. The likely cause is that every time we add a policy element, we increase by n the number of access requests that need to be checked, where n is the number of policy elements either being accessed or doing the accessing.



(a) Simple seed policy.



(b) Example mutation: new assignment.



(c) Example mutation: new association.

Figure 3: Graphical representation of the simple seed policy and two mutated policies. New elements and relations are bolded. (a) The simple seed policy. (b) A new assignment is added between $u1$ and $ua2$. Results in scope leakage vulnerability. (c) A new association relation is added from $ua2$ to $oa1$ with access right $p1$. Results in association overwrite vulnerability.

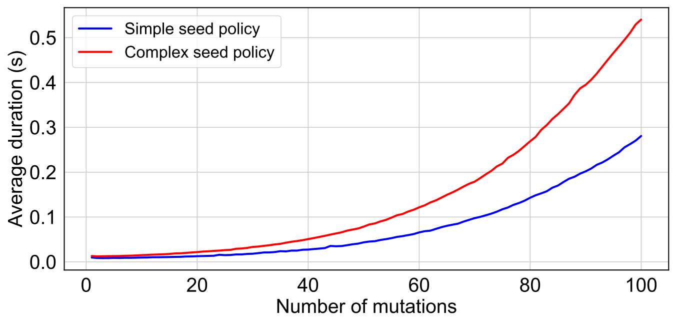


Figure 4: Average duration per mutation round across 1,000 independent runs (100 mutations per run).

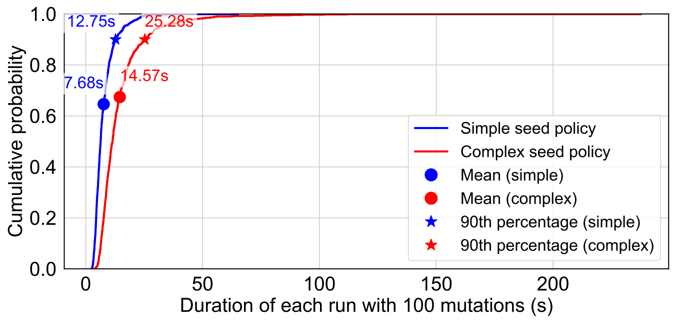


Figure 5: CDF of the cumulative duration of all mutations for each run.

For the purpose of analyzing runtimes, we force ProfessorX to execute all 100 rounds regardless of if it found a discrepancy or not. From Figure 5, we see the average duration of these 1,000 runs starting with the simple seed policy was 7.68 seconds, and the 90th percentile runtime was 12.75 seconds. The maximum duration taken by a single run is 65.19 seconds. For the complex seed policy, the average was 14.57 seconds, the 90th percentile runtime was 25.28 seconds, and the maximum duration was 237.7 seconds.

These runtime numbers are well within reasonable bounds for what a developer might expect in their pre-commit workflow. Running these types of tests alongside unit tests, or even alongside traditional fuzzing frameworks could be an effective way to reduce the presence of hard-to-detect vulnerabilities.

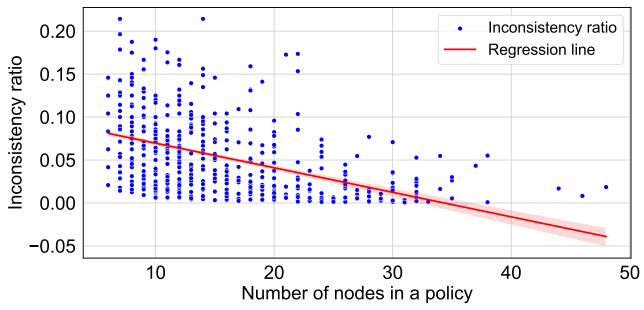


Figure 6: Ratio between access decision discrepancies and total access requests as a function of the total number of policy elements in our simple policy. Each point represents a mutated policy that caused a discrepancy.

5.3 Security Implications (Q3)

It is not immediately obvious what the scope of impact of a single discrepancy might be. For instance, it could indicate that the underlying vulnerability makes the entire set of decisions liable to be incorrect. In Figure 6 we plot the ratio between resource accesses that cause discrepancies and total resource accesses for policies of varying sizes. We see a general trend towards lower inconsistency ratios as policy sizes increase. This declining trend suggests that the final mutation that causes an inconsistency is creating a number of discrepancies that is increasing more slowly than the total number of accesses, which is proportional to the total number of nodes. From this we can infer that the created discrepancies are usually localized within the policy graph and usually do not create wide ranging effects. This suggests that while a single incorrect policy decision is still problematic, the existence of a mistaken vulnerability is not likely to have broad impact within a single policy.

5.4 Root Cause Analysis

Using ProfessorX, we discovered that all of the discrepancies we detected between $\mathrm{NGAC}_{\mathrm{Tiny}}$ and $\mathrm{NGAC}_{\mathrm{NIST}}$ were the result of two root causes. We analyze each in this section.

Association Overwrite. The first vulnerability we discovered happens when adding a new association $(ua, newAR, at)$ to a policy containing an existing association $(ua, oldAR, at)$, where $newAR \neq oldAR$. Because the access rights are not equal, the new access right $newAR$ should be added *in addition* to the old access rights $oldAR$ [19, p. 46]. However, we see in $\texttt{NGAC}^{24}_{\texttt{NIST}}$ that the new association overwrites the existing one. We confirmed this in the code where we observe that, they explicitly delete all associations between policy elements when adding a new association between those elements [36]. This is a direct violation of the NGAC specification. The correct behavior would be to define an association with the union of the access rights in $newAR$ and $oldAR$.

Figure 3c shows the added association $(ua2, p1, oa1)$ that triggers the discrepancy between $\texttt{NGAC}_{\texttt{NIST}}$ and $\texttt{NGAC}_{\texttt{Tiny}}$. The two policy engines make different decisions on four access requests. The implications of this vulnerability is that it is possible for an adversary to create associations that block other associations from being honored—this constitutes a denial of service attack. This is especially counterintuitive because adding an association should never reduce the scope of access for any user.

Scope Leakage. The second vulnerability we discovered occurs when adding an assignment between two existing nodes, where one of the nodes becomes contained by two unconnected nodes. This vulnerability only applies to $\texttt{NGAC}^{22}_{\texttt{NIST}}$, and appears to have been fixed in $\texttt{NGAC}^{24}_{\texttt{NIST}}$. We saw this, when the seed policy in Figure 3a had the assignment relation $(u1, ua2)$ added, which is highlighted in bold in Figure 3b. This caused two access request discrepancies.

Manual analysis of the policy revealed that the user attribute $ua1$ should not have access $p2$ access on object $o1$, however $\texttt{NGAC}^{22}_{\texttt{NIST}}$ grants such an access. This vulnerability results from a variable scoping bug. Policy remediation requires a tree traversal starting at the subject who is performing the access. The engineers have implemented a recursive breadth-first search traversal, and in each recursive frame, they collect the objects that are accessible from the node being visited along with permissions for that access—they call these accessible objects, *border targets* [36]. In general, this approach is fine for the remediation process, however, their implementation stores these border targets as a variable that is included within a closure function that gets called on every recursion. The scoping is such that border targets persist once they have been collected, even when returning up the recursive call stack to branch at a previous ancestor. This results in granting permissions to users unrelated to the collected border targets. The impact of this vulnerability is that seemingly benign changes to the policy could cause some users to gain unauthorized access.

6 Concluding Discussion

In this paper, we describe a methodology for policy engine differential mutation analysis, a system we implemented and evaluated in the form of ProfessorX. Despite our findings that type of scheme can find novel vulnerabilities in the form of silently incorrect policy decisions, there are a number of open questions worth considering.

Correctness. ProfessorX offers the ability to find silently incorrect policy engine decisions, but it does not give any guarantees that it will. So it is possible that vulnerabilities can exist in the system, but may not be detected. There are two considerations on this point. First, if the set of mutations is complete—meaning it models every way a policy could change—then it is *highly* likely that an extant vulnerability will eventually be found. Second, our implementation of $\texttt{NGAC}_{\texttt{Tiny}}$ does not come with any guarantees that *it* implemented the NGAC standard correctly. However, the benefit of ProfessorX, is that it actively compares end results between implementations. In the case of a discrepancy, we first look into what the correct outcome should have been according to the specification, and then decide which of the models was incorrect. Thus our system leans more on the comparison between the $\texttt{NGAC}_{\texttt{Tiny}}$ and $\texttt{NGAC}_{\texttt{NIST}}$ than rather than the objective correctness of $\texttt{NGAC}_{\texttt{Tiny}}$. As a result, any mis-implementation in either $\texttt{NGAC}_{\texttt{Tiny}}$ or $\texttt{NGAC}_{\texttt{NIST}}$ is likely to be caught. The caveat, is that if both implementations mis-implement some component of the standard, then no discrepancies will arise and the vulnerability will not be detected.

Realistic Seed Policies. It is *de rigueur* in security research to rely on datasets that reflect real world implementations. Unfortunately, we do not have access to any real enterprise's security policies, because the release of such a dataset would itself be a major security issue for that enterprise. However, it appears likely that more realistic seed policies do not necessarily make ProfessorX more effective at finding discrepancies, as we detect them at the same rate for both of the seed policies we based our testing on. On the other hand, realistic policies may be much larger—potentially containing tens of thousands of users and objects—this, we are certain, would have a significant negative impact on ProfessorX runtime. Therefore, smaller seed policies may be the best option.

Mutations. The ability of ProfessorX to catch discrepancies is impacted by the set of mutations are available to be tested. For example, during our development of ProfessorX, we did not catch any association overwrite discrepancies until adding the `CreateAssoc` mutation. In retrospect, this is unsurprising, however that still leaves us in the dark about whether there are further discrepancies that remain to be uncovered if we only implement the right mutations. Future work could address this by implementing all isolated changes that could be made to a policy. In particular, instead of only adding policy elements or relations, they could delete them. They could also transform one relation into another. There are further questions that would need to be answered to do so correctly, but this is within the realm of possibility.

Usability. We conclude by suggesting that this would not be a difficult tool to integrate into a development workflow. Many *continuous integration* tools provide hooks for running tests of all sorts, so adding this via the same mechanisms would not be difficult. This approach could also be generalized for other policy frameworks, or extended to include new policy engine implementations. Broader adoption could only enhance these applications which find themselves so central to enterprise security.

Acknowledgements

This work was supported in part by ONR grant N00014-20-1-2696. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] 2023. https://github.com/coq/coq

[2] Iffat Anjum, Daniel Kostecki, Ethan Leba, Jessica Sokal, Rajit Bharambe, William Enck, Cristina Nita-Rotaru, and Bradley Reaves. 2022. Removing the Reliance on Perimeters for Security using Network Views. In *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*. ACM, New York NY USA, 151–162. doi:10.1145/3532105.3535029

[3] Iffat Anjum, Jessica Sokal, Hafiza Ramzah Rehman, Ben Weintraub, Ethan Leba, William Enck, Cristina Nita-Rotaru, and Bradley Reaves. 2023. MSNetViews: Geographically Distributed Management of Enterprise Network Security Policy. In *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies* (Trento, Italy) *(SACMAT '23)*. Association for Computing Machinery, New York, NY, USA, 121–132. doi:10.1145/3589608.3593836

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).

[5] D Elliott Bell and Leonard J La Padula. 1976. Secure computer system: Unified exposition and multics interpretation. *(No Title)* (1976).

[6] Matt Bishop. 2004. *Introduction to computer security.* Addison-Wesley Professional.

[7] Bruno Blanchet et al. 2016. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security* 1, 1-2 (2016), 1–135.

[8] David FC Brewer and Michael J Nash. 1989. The Chinese Wall Security Policy.. In *S&P*. 206–214.

[9] Timothy Alan Budd. 1980. *Mutation analysis of program test data.* Yale University.

[10] Ramaswamy Chandramouli and Zack Butcher. 2023. A Zero Trust Architecture Model for Access Control in Cloud-Native Applications in Multi-Location Environments. doi:10.6028/NIST.SP.800-207A

[11] Ramaswamy Chandramouli, Zack Butcher, and Aradhna Chetal. 2021. Attribute-based Access Control for Microservices-based Applications Using a Service Mesh. doi:10.6028/NIST.SP.800-204B

[12] Erzhuo Chen, Vladislav Dubrovenski, and Dianxiang Xu. 2021. Mutation Analysis of NGAC Policies. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies* (Virtual Event, Spain) *(SACMAT '21)*. Association for Computing Machinery, New York, NY, USA, 71–82. doi:10.1145/3450569.3463563

[13] Hong Chen, Ninghui Li, and Ziqing Mao. [n. d.]. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. ([n. d.]).

[14] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 793–804.

[15] Luke Deshotels, Razvan Deaconescu, Mihai Chiroiu, Lucas Davi, William Enck, and Ahmad-Reza Sadeghi. 2016. SandScout: Automatic Detection of Flaws in iOS Sandbox Profiles. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna Austria, 704–716. doi:10.1145/2976749.2978336

[16] David Ferraiolo, Vijayalakshmi Atluri, and Serban Gavrila. 2011. The Policy Machine: A novel architecture and framework for access control policy specification and enforcement. *Journal of Systems Architecture* 57, 4 (2011), 412–424. doi:10.1016/j.sysarc.2010.04.005

[17] David F. Ferraiolo, Serban I. Gavrila, Wayne Jansen, and Paul E. Stutzman. 2015. Policy Machine: Features, Architecture, and Specification. doi:10.6028/NIST.IR.7987r1

[18] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.* 4, 3 (Aug. 2001), 224–274. doi:10.1145/501978.501980

[19] InterNational Committee for Information Technology Standards. 2020. INCITS 565-2020: Information technology – Next Generation Access Control. https://standards.incits.org/higherlogic/ws/public/projects/2328/details

[20] Carrie Gates. 2007. Access control requirements for web 2.0 security and privacy. *IEEE Web* 2, 0 (2007), 12–15.

[21] Edward L Glaser. 1967. A brief description of privacy measures in the Multics operating system. In *Proceedings of the April 18-20, 1967, spring joint computer conference.* 303–304.

[22] Grant Hernandez, Swarnim Yadav, Byron J Williams, and Kevin R B Butler. [n. d.]. BigMAC: Fine-Grained Policy Analysis of Android Firmware. ([n. d.]).

[23] Boniface Hicks, Sandra Rueda, Luke St.Clair, Trent Jaeger, and Patrick McDaniel. 2010. A logical specification and analysis for SELinux MLS policy. *ACM Transactions on Information and System Security* 13, 3 (July 2010), 1–31. doi:10.1145/1805974.1805982

[24] G.J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (May 1997), 279–295. doi:10.1109/32.588521

[25] Vincent Hu, David Ferraiolo, Richard Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. 2019. *Guide to attribute based access control (ABAC) definition and considerations.* Technical Report NIST Special Publication (SP) 800-162. National Institute of Standards and Technology. doi:10.6028/NIST.SP.800-162

[26] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis.* MIT press.

[27] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28.* Springer, 422–430.

[28] Wonhoi Kim, Hocheol Nam, Muoi Tran, Amin Jalilov, Zhenkai Liang, Sang Kil Cha, and Min Suk Kang. 2025. Fork State-Aware Differential Fuzzing for Blockchain Consensus Implementations. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE).* IEEE Computer Society, 622–622.

[29] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* ACM, Big Sky Montana USA, 207–220. doi:10.1145/1629575.1629596

[30] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25.* Springer, 696–701.

[31] Dimitrios Michail, Joris Kinable, Barak Naveh, and John V. Sichi. 2020. JGraphT—A Java Library for Graph Data Structures and Algorithms. *ACM Trans. Math. Softw.* 46, 2, Article 16 (May 2020), 29 pages.

[32] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP).* IEEE, 615–632.

[33] Scott Rose, Oliver Borchert, Stuart Mitchell, and Sean Connelly. 2020. Zero Trust Architecture. doi:10.6028/NIST.SP.800-207

[34] Sandra Rueda, Dave King, and Trent Jaeger. [n. d.]. Verifying Compliance of Trusted Programs. ([n. d.]).

[35] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop.* Springer, 245–251.

[36] NGAC Implementation Team. 2025. NGAC Reference Implementation. Retrieved November 2024 from https://github.com/usnistgov/policy-machine-core. https://github.com/usnistgov/policy-machine-core

[37] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. Swi-prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.

[38] Jiahe Zhang, Jianjun Chen, Qi Wang, Hangyu Zhang, Chuhan Wang, Jianwei Zhuge, and Haixin Duan. 2024. Inbox Invasion: Exploiting MIME Ambiguities to Evade Email Attachment Detectors. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 467–481. doi:10.1145/3658644.3670386

A Functional Architectures

Policy engines play a central role in enterprise security architecture. Hosted as a cloud service or on the enterprise local network, policy engines work in concert with Policy Enforcement Point (PEPs) components, e.g., routers, to control and reconfigure the network access for protecting resources. The NGAC-compatible policy engine is consistent with the policy engine in the ZTA conceptual architecture [33]. We now describe the functional architecture of NGAC to give context to how access requests are processed by policy engines.

For a resource governed by NGAC, a user can only access it through resource access. With the functional architecture of NGAC [19], the resource access information flow is illustrated in Figure 7. A process on behalf of a user attempts to access a resource through a PEP (Step ❶). The PEP exposes an interface for instances of NGAC-aware processes, enabling them to access resources. The PEP then sends the access request for decision to a Policy Decision Point (PDP), which provides an interface for use by the PEP (Step ❷). In order to make decisions, the PDP needs to interact with a Policy Administration Point (PAP) to retrieve the required policy information from the PAP (Step ❸) after the PAP has searched and manipulated the policy information persisted at a Policy Information Point (PIP).

The NGAC policy is persisted at the PIP and constitutes the authorization state of the system. It is collectively defined by the basic elements and relations. The PDP makes a decision on the access request and returns the decision and resource locator to the PEP (Step 4). The PEP then issues a directive to a Resource Access Point (RAP) (Step 5) to access the resource. The RAP performs the operation on the resource and receives the status information and data (if any), which is eventually returned to the process through the PEP (Step 6).

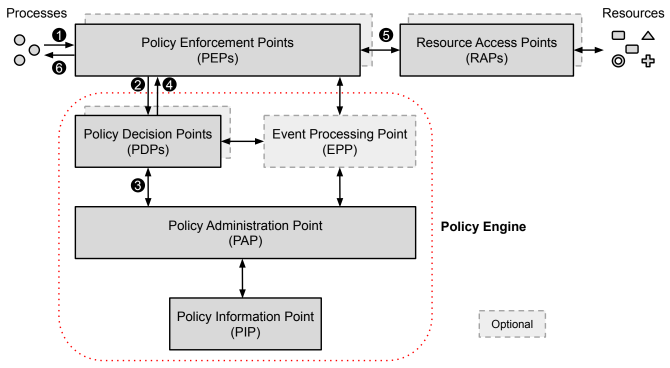

Processes
Policy Enforcement Points (PEPs)
Resource Access Points (RAPs)
Resources
Policy Decision Points (PDPs)
Event Processing Point (EPP)
Policy Administration Point (PAP)
Policy Information Point (PIP)
Policy Engine
Optional
1
2
3
4
5
6

Figure 7: NGAC functional architecture. NGAC-compatible policy engine comprises the functional entities in the red dashed box.

B Complex Seed Policy

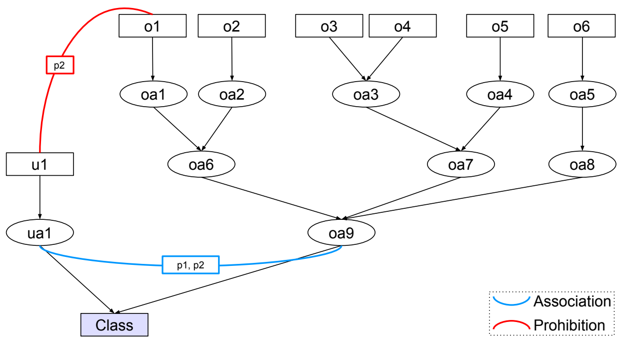


Figure 8: Graphical representation of the complex seed policy.

C Additional NGAC Model Details

Obligations. Obligations define actions automatically triggered in response to specific events matching predefined patterns. An obligation relation is a ternary relation $(u, pattern, response) \in OBLIG$, where $u \in U$ represents the user responsible for establishing the obligation and under whose authorization the response is carried out. The aforementioned policy elements and relations constitute the NGAC authorization state. An obligation modifies the authorization state by the actions specified in the response, including changing existing policy elements and relations.

In ProfessorX, we do not include obligation elements, which change the effectively enforced policy. In practice though, we are able to model the downstream effects of obligations, because a processed obligation simply results in either a change in policy elements or relations—all of which we model. Thus effective policy changes induced by obligations have the same effect as the mutations we model in our differential mutation analysis framework.

D Prolog Implementation of NGAC$_{\texttt{Tiny}}$

```
legalAssignment(X, Y) :-
    assign(X, Y),
    ((u(X), ua(Y)); (ua(X), ua(Y));
     (o(X), oa(Y)); (oa(X), oa(Y))).

isContained(X, Y) :-
    X = Y;
    legalAssignment(X, Y);
    legalAssignment(X, Z),
    isContained(Z, Y).

legalAssociation(UA, OA, ARS) :-
    association(UA, OA, ARS),
    (
        (ua(UA), ua(OA), legalAccessRights(ARS));
        (ua(UA), oa(OA), legalAccessRights(ARS));
        (ua(UA), o(OA), legalAccessRights(ARS))
    ).

legalAccessRights([H|T]) :-
    (ar(H), T = []);
    (ar(H), legalAccessRights(T)).

disjProhibited(U, AT, AR) :-
    disjunctiveProhibition(U_or_UA, ATI, ARS),
    legalAccessRights(ARS),
    (ua(AT); oa(AT); o(AT)),
    member(AR, ARS),
    isContained(U, U_or_UA),
    inInclusionSet(AT, ATI).

inInclusionSet(AT, [Head|Tail]) :-
    isContained(AT, Head);
    inInclusionSet(AT, Tail).

decideAll(U, PE, AR) :-
    legalAssociation(UA, PE_Parent, ARS),
    member(AR, ARS),
    isContained(U, UA),
    isContained(PE, PE_Parent),
    \+ (
        isContained(PE, PE_Prohib),
        disjProhibited(U, PE_Prohib, AR)
    ).

decide(U, O, AR) :-
    once(decideAll(U, O, AR)).
```