Detection of Multiple-Duty-Related Security Leakage in Access Control Policies

JeeHyun Hwang[1] Tao Xie[1] Vincent C. Hu[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206

[2] Computer Security Division, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930

jhwang4@ncsu.edu xie@csc.ncsu.edu vincent.hu@nist.gov

Abstract

Access control mechanisms control which subjects (such as users or processes) have access to which resources. To facilitate managing access control, policy authors increasingly write access control policies in XACML. Access control policies written in XACML could be amenable to multiple-duty-related security leakage, which grants unauthorized access to a user when the user takes multiple duties (e.g., multiple roles in role-based access control policies). To help policy authors detect multiple-duty-related security leakage, we develop a novel framework that analyzes policies and detects cases that potentially cause the leakage. In such cases, a user taking multiple roles (e.g., both r_1 and r_2) is given a different access decision from the decision given to a user taking an individual role (e.g., r_1 and r_2, respectively). We conduct experiments on 11 XACML policies and our empirical results show that our framework effectively pinpoints potential multiple-duty-related security leakage for policy authors to inspect.

Keywords: Validation, Policy Verification, Access Control Policies.

1 Introduction

Access control is one of the most fundamental and widely used security mechanisms. It controls which principals (such as users or processes) have access to which resources in a system. To facilitate managing and maintaining access control, access control policies are increasingly written in specification languages such as XACML [1] and Ponder [7]. Whenever a principal requests access to a resource, that request is passed to a software component called a *Policy Decision Point* (PDP). A PDP evaluates the request against the specified access control policies to decide whether the request is permitted or denied accordingly. In this paper, we focus on access control policies written in XACML (eXtensible Access Control Markup Language) [1, 13, 14], which has become the de facto standard for specifying access control policies for various applications, especially web applications.

Assuring the correctness of policy specifications is becoming an important and yet challenging task, especially as access control policies become more complex and are used to manage a large amount of sensitive information organized into sophisticated structures in a network. Identifying discrepancies between policy specifications and their intended function is crucial because correct implementation and enforcement of policies by applications is based on the premise that the policy specifications are correct. As a result, policy specifications must undergo rigorous verification and validation to ensure that the policy specifications truly encapsulate the desires of the policy authors.

Role-Based Access Control (RBAC) [8] assigns permissions of specific actions on resources to authorized users called roles. In XACML policies, rules are written to specify such permissions on roles. However, specifying correct behaviors of roles in policies is not an easy task, especially for a user with multiple roles. For example, in a policy, multiple rules can be applicable to a request (including multiple roles), and the policy needs to be specified correctly to select which rule should be given higher priority than other rules if access decisions from the rules are conflicting. In XACML, such scenarios are described using rule combining algorithms. These algorithms describe when a certain rule overrides other rules. Although each rule could be specified correctly, rule combining algorithms or the rule order may include a fault to produce security leakage (i.e., actual and expected decisions are inconsistent), where multiple rules are applicable to a request involving a user taking multiple roles and the given priority among the rules is incorrect. We refer to such a security issue as multiple-duty-related security leakage.

In XACML, as the *first-applicable algorithm*, a commonly used rule combining algorithm, takes into account the order of rules in evaluation, incorrect order may be a source to introduce multiple-duty-related security leakage. In addition, some roles may be mutually exclusive and their combination is restricted through Separation of Duty (SoD) enforcement on policies [8]. SoD is used to avoid or deny a request that a user in mutually disjoint roles tries to request access permissions. Incorrect SoD enforcement may produce multiple-duty-related security leakage as well.

When rules are specified on a case-by-case basis, the policy authors may often forget to handle corner cases where a user can take multiple roles. As multiple-duty-related security leakage is often caused by mistakes in handling multiple roles, efforts to verify each role's behavior individually is not sufficient to detect the leakage. In particular, if a role is added/deleted in policies, the policy authors should verify how the role interacts with other roles besides verifying each role's behavior individually. Without effective tool support, it is tedious and error-prone for policy authors to manually verify interactions among roles and hence detect security leakage related to corner cases.

One typical approach is to enforce Separation of Duty (SoD) on every combination of roles for preventing such leakage. Therefore, requests with multiple role are discarded, and only requests with a single role are valid to be evaluated in that approach. This approach is often found to be rigid and limits the flexibility of XACML. More specifically, Static SoD (SSD) limits the conflicting-role assignments that are associated with a user [8]. For example, if a `Student` role and a `Faculty` role are declared as SoD constraints statically, no permission is given to a user who is assigned to both the `Student` role and the `Faculty` role. However, an SSD policy is often found to be too rigid for practical use because in some cases a faculty member should be allowed to get enrolled to become a student at the same time. The only restriction is that if a user has both the `Student` and the `Faculty` roles, the user cannot write grades. While Dynamic SoD (DSD) [8] is known to be more flexible than SSD, the XACML profile for RBAC [5] does not provide direct support of the DSD enforcement mechanism. Due to the limitations of these existing approaches, our work focuses on helping write policies in a way that can avoid multiple-duty-related security leakage.

To the best of our knowledge, there is no prior research work on verifying policies with regards to multiple-duty-related leakage. To help policy authors detect multiple-duty-related security leakage, we develop a novel static policy analysis framework that detects cases where a user taking multiple roles (e.g., r_1 and r_2) is given a different access decision than the user taking an individual role (r_1 or r_2). The policy authors inspect such cases to determine whether the cases can cause security leakage. If the policy authors determine that the reported leakage is real, they can modify the policy to prevent such leakage.

This paper makes three main contributions:

- To the best of our knowledge, we are the first to present the problem of multiple-duty-related leakage in XACML policies for policy authors to verify.
- Our framework addresses the problem of detecting potential multiple-duty-related leakage where a user taking multiple roles (e.g., r_1 and r_2) is given a different access decision than the user taking an individual role (e.g., r_1 and r_2, respectively).
- We conduct experiments on 11 XACML policies collected from various sources. Our empirical results show that our framework can effectively pinpoint potential multiple-duty-related security leakage for policy authors to inspect.

The rest of the paper is organized as follows. Section 2 presents the XACML specification language. Section 3 presents an example to illustrate potential security leakage with regards to multiple duties. Section 4 presents the request evaluation for policies. Sections 5 and 6 present our framework and its implementation, respectively. Section 7 describes the evaluation where we apply our framework on various XACML policies. Section 8 discusses related work. Section 9 concludes the paper.

2 Background

XACML (eXtensible Access Control Markup Language) [1] is a language specification standard published by OASIS (Organization for the Advancement of Structured Information Standards). It was mainly designed as a standard for expressing both access requests and access control policies. XACML offers a large set of built-in functions, data types, combining logic, and standard extension interfaces for defining application-specific features. Various domain-specific access control languages [13,14] have been designed using XACML. Open source XACML implementations are also available for different platforms (e.g., Sun's XACML implementation [2] and XACML.NET [3]).

An XACML access control specification consists of a policy set and a policy combining algorithm. A *policy set* is an ordered list of policies. A *policy* includes a target, a rule set, and a rule combining algorithm. A *target* is a predicate over the subject, the resource, and the action of access requests, specifying the type of requests to which the policy can be applied. If a request satisfies the target of an access control policy, then the request is further checked against the rule set of the policy; otherwise, the policy is skipped without further examining its rules. A *rule set* is an ordered list of rules. A *rule* consists of a target, a condition, and an effect. The *target* of a rule is similar to the target of a policy, but specifies whether the rule is applicable to a request. Given a request, if a rule is applicable, the *condition* (i.e., a boolean function) associated with the rule is evaluated. If the condition is evaluated to be true, the rule's *effect* (i.e., `Permit` or `Deny`) is returned as a *decision*; otherwise, `NonApplicable` is returned as a decision.

```
 1<Policy PolicyId="univ" RuleCombAlgId="first-applicable">
 2 <Target>
 3   <Subjects> <AnySubjects/> </Subjects>
 4   <Resources> <AnyResource/> </Resources>
 5   <Actions> <AnyAction/> </Actions>
 6 </Target>
 7 <Rule RuleId="1" Effect="Permit">
 8  <Target>
 9   <Subjects><Subject> Faculty </Subject></Subjects>
10   <Resources>
11          <Resource> ExternalGrades </Resource>
12          <Resource> InternalGrades </Resource>
13   </Resources>
14   <Actions><Action> View </Action>
15            <Action> Write </Action></Actions>
16  </Target></Rule>
17 <Rule RuleId="2" Effect="Permit">
18  <Target>
19   <Subjects><Subject> Student </Subject></Subjects>
20   <Resources>
21       <Resource> ExternalGrades </Resource>
22   </Resources>
23   <Actions><Action> View </Action></Actions>
24  </Target>
25 </Rule>
26 <Rule RuleId="3" Effect="Deny">
27  <Target>
28   <Subjects><Subject> Student </Subject></Subjects>
29   <Resources>
30        <Resource> ExternalGrades </Resource>
31   </Resources>
32   <Actions><Action> Write </Action></Actions>
33  </Target>
34 </Rule>
35 <!-- A final, "fall-through" rule that always Denies -->
36 <Rule RuleId="FinalRule" Effect="Deny"/>
37</policy>
```

Figure 1. An example XACML policy

In general, `NonApplicable` is considered `Deny`. If an error occurs when a request is applied against policies or their rules, `Indeterminate` is returned as a decision.

More than one rule in a policy may be applicable to a given request. The *rule combining algorithm* is used to combine multiple rule decisions into a single decision. There are four standard rule combining algorithms: `deny-overrides`, `permit-overrides`, `first applicable`, and `only-one-applicable`, whose meanings are self-evident from the algorithm naming [1]. Four similar *policy combining algorithms* are also used to combine multiple policy decisions into a single decision.

3 Example

Figure 1 shows an example XACML policy adapted from a sample policy used by Fisler et al. [10]. This example illustrates a policy that uses the `first-applicable` algorithm, which determines to return the evaluated decision of the first applicable rule. In this example, there are two subjects or roles (`Faculty`, `Student`), two resources (`ExternalGrades`, `InternalGrades`), and two actions (`View`, `Write`).

There are four rules in the policy. Lines 7-16 define the first (permit) rule, which allows a faculty to view or write external or internal grades. Lines 17-25 define the second (permit) rule, which allows a student to view external grades. Lines 26-34 define the third (deny) rule, which denies a student to write external grades. Line 36 defines the last default (deny) rule, which denies any request that does not match any of the three preceding rules.

We simplify and represent the illustration of the example policy in Figure 1 in IF-THEN statements as shown in Figure 2. As the example policy uses the first applicable algorithm, the IF-THEN statements illustrate the same policy behavior by mapping the predicates and their decisions as the conditional elements and the values in the branches, respectively.

```
 1 If role = Faculty
 2   and resource = (ExternalGrades or InternalGrades)
 3   and action = (View or Write) then Permit
 4 If role = Student
 5   and resource = ExternalGrades
 6   and action = View then Permit
 7 If role = Student
 8   and resource = ExternalGrades
 9   and action = Write then Deny
10 Deny
```

Figure 2. Rules in the example XACML policy

Consider that the following three requests are evaluated against the example policy by a PDP:

- Req$_1$: a member of `Faculty` role wishes to `Write ExternalGrades`.
- Req$_2$: a member of `Student` role wishes to `Write ExternalGrades`.
- Req$_3$: a member of `Student` and `Faculty` roles wishes to `Write ExternalGrades`.

Req$_1$ and Req$_2$ are *single-valued requests*, each of which has only a single attribute id-value pair of the subject, object, and action attributes. Req$_1$ is permitted by the specified condition (Lines 1-3 of Figure 2). Req$_2$ is denied by the specified condition (Lines 7-9 of of Figure 2). For the two cases, the decision is specified in the rule. As shown before, the evaluation of single-valued requests by the PDP is straightforward. The corresponding policy behavior (about a request) is often described explicitly in a rule of the policy.

Req$_3$ is a *multi-valued request*, which contains multiple id-value pairs in the subject, resource, or action attribute. Req$_3$ is applicable to multiple conditions (Lines 1-3 and Lines 7-9) in the example policy. The PDP's final decision is made according to the decision of each rule and a conflict-resolution algorithm. Because the policy uses the `first-applicable` combining algorithm, the earlier-placed condition is given higher priority. Req$_3$ is permitted based on the decision of the first-applicable condition (Lines 1-3). To this request, the later-placed condition

(Lines 7-9) is shadowed by the condition (Lines 1-3) that is the first-applicable condition.

Req$_3$'s partial attribute values (`Faculty`, `Write`, and `ExternalGrade`) is sufficient to satisfy the first condition (Lines 1-3) without using a `Student` role. However, because the permission is given to Req$_3$, a student (when holding a faculty role together) can write an external grade; this case is not explicitly specified in any rule of the example policy. Req$_3$ represents cases, where the request evaluation interacts with multiple conditions to output inconsistent decisions with Req$_1$ or Req$_2$, and may indicate a potential fault. A fault in the policy specification can be discovered by investigating the responses of Req$_1$, Req$_2$, and Req$_3$.

In this example, the policy authors shall not intend for a student to have permissions to write her own grade. This fault is the same discrepancy found by Fisler et al. [10] (whose approach requires application-specific policy properties not required in our framework), which is a result of a subtlety of the XACML language. The root cause of the problem is that XACML allows an arbitrary number of values for a given attribute. This example illustrates that the investigation of requests and responses can lead to the detection of a fault in policy specifications. The fault may lead to leakage of privilege that is not intended by the policy authors. In this paper, we call this fault as multiple-duty-related security leakage; the fault may not be trivial to be detected in complex policies without using an effective detection mechanism. To address these issues, we propose a framework to analyze the policy under test statically and detect such corner cases for the policy authors to inspect.

4 XACML Policies and Evaluation

This section describes the XACML structure to evaluate a request.

4.1 Rule Applicability

XACML policies are composed of a set of rules, which specify under what conditions a subject is allowed or denied access to certain objects (i.e., resources) in a system. To discuss rule applicability, we model access requests and policies in this paper as follows.

Let $\mathcal{S}$, $\mathcal{O}$, and $\mathcal{A}$ denote respectively the set of all the subjects, objects, and actions in an access control system. Each subject, object, or action is associated with a set of attributes that may be used for access control decisions. For example, a subject's attributes may include a user's role, rank, and security clearance. An object's attributes may include a file type, a document's security class, and a printer's location.

An access control policy P is a sequence of rules, each of which is of the form as shown below.

$Rule : (Cond_s, Cond_o, Cond_a, decision, Cond_g)$

where $Cond_s$, $Cond_o$, and $Cond_a$ are constraints over the attributes of a subject, object, and action, respectively. $Cond_g$ is a general constraint that may potentially be over all the attributes of subjects, objects, actions, and other properties of a system (e.g., the current time and the load of a system). A $decision$ is either $deny$ or $permit$.

An access request q is a tuple (s, o, a), where $s \subseteq \mathcal{S}$, $o \subseteq \mathcal{O}$ and $a \subseteq \mathcal{A}$. A request (s, o, a) means that subject s requests to take action a on object o. Given a request (s, o, a), if $Cond_s(s)$, $Cond_o(o)$, $Cond_a(a)$, and $Cond_g$ are all evaluated to be $true$, then the request is either permitted or denied, according to the $decision$ in the rule. In such a case, we say that the rule is *applicable* to the request. Note that a rule is still *applicable* by a request that includes additional attributes over the minimal set of (s, o, a) to satisfy the rule's constraints.

4.2 Policy and PolicySet

An XACML policy is constructed as a tree structure. An XACML policy consists of a *policy set*, which consists of *policy sets* and *policies*. A *policy* consists of a sequence of *rules*. A *policy set* and a *policy* are of the forms as shown below.

$PolicySet$:
$(Cond_s, Cond_o, Cond_a, PSSet, PSet, ComAlg)$
$Policy$:
$(Cond_s, Cond_o, Cond_a, RSet, ComAlg)$
where $Cond_s$, $Cond_o$ and $Cond_a$ are constraints over the attributes of a subject, object, and action, respectively, and $PSSet$, $PSet$, $RSet$, and $ComAlg$ are a set of XACML *policy sets*, a set of XACML *policies*, a set of rules, and a combining algorithm, respectively.

Given a request (s, o, a), applicable *policy sets* and *policies* are collected in a depth-first traversal. For applicable *policy sets* and *policies*, the corresponding $Cond_s(s)$, $Cond_o(o)$, $Cond_a(a)$, and $Cond_g$ should be evaluated to be $true$; otherwise, the *policy set* or *policy* is skipped when evaluating the request. In the policy and rule collection, from the top to the bottom, we apply the combining algorithm $ComAlg$ to resolve multiple applicable policies or rules. The algorithm can be denial overriding permission (where a request is denied if it is denied by at least one rule/policy), permission overriding denial (where a request is permitted if it is permitted by at least one rule/policy), or first applicable (where the final decision is the same as that of the first applicable rule/policy in a sequence of policies or rules).

5 Framework

This section presents our framework for detecting multiple-duty-related security leakage. Our framework in-

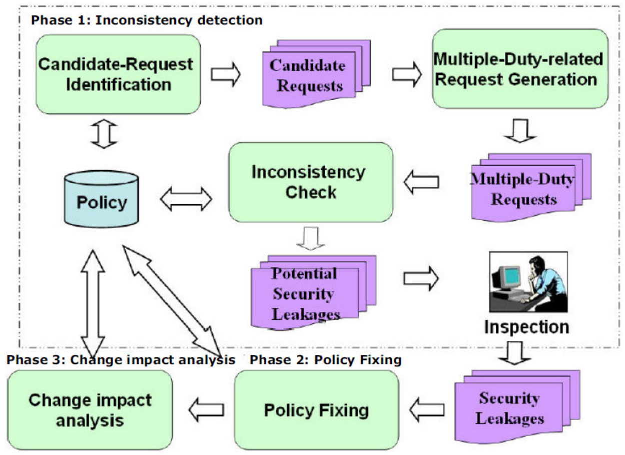
Phase 1: Inconsistency detection
Candidate-Request Identification
Candidate Requests
Multiple-Duty-related Request Generation
Policy
Inconsistency Check
Multiple-Duty Requests
Potential Security Leakages
Inspection
Phase 3: Change impact analysis
Phase 2: Policy Fixing
Change impact analysis
Policy Fixing
Security Leakages

Figure 3. Framework overview

cludes three phases: inconsistency detection, policy fixing, and change-impact analysis. In the inconsistency detection phase, our inconsistency detection component detects inconsistency of the responses of multiple-valued requests and their corresponding single-valued requests. In the policy fixing phase, the policy authors can inspect the reported inconsistencies and fix the policy by modifying the policy in appropriate ways. In the change-impact analysis phase, our framework detects whether the policy fixes are expected, i.e., whether only the expected behavioral changes, i.e., bug-fixing changes, (and no other unintended changes) are caused by the policy modification. If there is any unexpected change, the policy fixing phase and the change-impact analysis phase are iterated until the policy behavior is expected.

5.1 Inconsistency Detection

The inconsistency detection phase aims at capturing potential faults in policy specifications. When the policy authors specify and combine rules, the policy authors may often forget to handle corner cases where a user can have multiple roles. In particular, we are interested in identifying why decisions are changed when a user take an additional role. In such a case, we suspect that an additional role contributes to change the evaluated decision. Although each of rules is correctly specified, requests with multiple roles likely match with multiple rules and could produce a wrong decision due to incorrectly specified rule combining or rule order. Inconsistent decisions could reveal such faults.

Figure 3 shows the three steps in the inconsistency detection phase: candidate-request identification, multiple-duty-related request generation, and inconsistency checking. In candidate-request identification, we start with a request q_1 as (s, o, a) and its evaluated decision dec_1. To generate a multiple-duty-related request, we add a role s' on q_1 and generate q_2 as $(s \cup s', o, a)$ and its evaluated decision dec_2. In inconsistency checking, we check if the two decisions dec_1 and dec_2 are the same. If two decisions are not the same, these requests may reveal multi-duty-related security leakage that should be inspected.

We classify inconsistencies with regard to a pair of inconsistent decisions as dec_1 and dec_2. In XACML, three different decisions are useful in practice including `Deny`, `Permit`, and `Not applicable` decisions. Therefore, all possible combinations of different decisions pairs lead to six categories. In the paper, we call each category as a dec_1-dec_2 inconsistency. For example, we call a (`Deny`,`Permit`) pair as the deny-permit inconsistency.

Evaluating a single request each time by a PDP is often tedious if a large number of requests should be evaluated. For efficiency, we collect a policy behavior report that describes all requests and their corresponding decisions. The policy behavior report captures the policy's behavior information and is reusable until any change is introduced to the policy. Our framework analyzes the policy behavior report and automatically selects a request-decision pair (where one request q_1 has one additional role over another request q_2) and checks whether there is inconsistency between the decisions of q_1 and q_2.

There are dynamic or static ways to collect policy behavior. In a dynamic way, we can generate and evaluate all possible requests to find the corresponding decisions. For an XACML policy, the number of possible requests is proportional to the number of attribute values in the policy. If the number of attribute values is large, the cost to evaluate these requests is also high.

Instead of using a dynamic way, we collect a policy behavior report statically. As dynamic features are outside the scope of existing XACML PDP implementations [5], the static way is sufficient to detect every case for potential multiple-duty-related security leakage. Moreover, the static way effectively reduces the number of the cases to be inspected using a summarized format in describing all possible request/decision pairs. For example, Fisler et al. [10] developed a tool called Margrave that analyzes and represents XACML policies. Margrave can output all request sets with their corresponding decisions in a summarized format. In this paper, our framework detects inconsistencies using the policy behavior report that is generated statically by Margrave.

5.2 Policy Fixing

In the policy fixing phase, the policy authors can inspect the reported inconsistencies and fix the policy by modifying the policy in appropriate ways. Given a faculty policy, the

```
If role = (Faculty and Student) then Deny
If role = Faculty
  and resource = (ExternalGrades or InternalGrades)
  and action = (View or Write) then Permit
If role = Student
  and resource = ExternalGrades
  and action = View then Permit
If role = Student
  and resource = ExternalGrades
  and action = Write then Deny
Deny
```

Figure 4. The example policy fixed by static separation of duty

```
 1 If role = Faculty
 2   and resource = (ExternalGrades or InternalGrades)
 3   and action = (View or Write)
 4   and role != Student then Permit
 5 If role = Student
 6   and resource = ExternalGrades
 7   and action = View then Permit
 8 If role = Student
 9   and resource = ExternalGrades
10   and action = Write then Deny
11 Deny
```

Figure 5. The example policy fixed by adding constraint on the first rule

```
If role = Student
  and resource = ExternalGrades
  and action = Write then Deny
If role = Faculty
  and resource = (ExternalGrades or InternalGrades)
  and action = (View or Write) then Permit
If role = Student
  and resource = ExternalGrades
  and action = View then Permit
Deny
```

Figure 6. The example policy fixed by moving the originally last rule to the top

policy authors identify security leakage and find the corresponding rule. The policy authors can add constraints to the rule to deny a request with certain multiple duties. The policy authors can also change the policy structure to fix the policy. However, this task is not easy to achieve our goal since one small change can impact other policy behavior, calling for the change-impact analysis phase described in Section 5.3.

We illustrate the ways of policy fixing through the example policy shown in Figure 2. Recall requests Req$_1$, Req$_2$, and Req$_3$ in Section 3 and the case of security leakage, where a student has permissions to write her own grades in Figure 2. After the policy authors confirmed reported cases to be security leakage, they can fix the XACML policy to meet the authors' expected policy behavior by preventing a student from writing her own grades.

First, the policy authors can enforce static Separation of Duty (SoD) constraints on an XACML policy. An SoD-constrained policy declares mutually exclusive set of roles; a member cannot be assigned to the SoD-declared set of roles simultaneously. Figure 4 shows the fixed example policy where a `Faculty` role and a `Student` role are specified as mutually exclusive. To achieve the goal, the policy authors can add the IF-condition "`If role = (Faculty and Student) then Deny`" on top of other conditions (in XACML specifications, one additional rule is added on top of other rules). This IF-condition ensures that every request that holds both a `Faculty` role and a `Student` role is denied.

Second, the policy authors can fix rules in XACML policies by adding appropriate predicates, which are $\langle Condition \rangle$ tags in an XACML policy. In $\langle Condition \rangle$ tags, the policy authors can apply various standard functions and custom functions to subtly handle a rule behavior. To prevent the detected security leakage, the policy authors can add the predicate "`role != Student`" in the $\langle Condition \rangle$ tag (Line 4) in the first condition of Figure 5. This predicate prevents an access request that includes a `Student` role from being evaluated in the corresponding condition. Therefore, with this additional predicate, the first condition cannot permit a request that holds both roles.

Third, the policy authors can modify a policy structure including policy combining algorithms, rule combining algorithms, policy order, and rule order such as moving the condition (Lines 7-9) in Figure 2 on top of all other conditions. Figure 6 shows that the moved condition is to be evaluated with a higher priority and returns a deny decision on Req$_3$.

5.3 Change-Impact Analysis

The purpose of change-impact analysis is to analyze what would be affected by a change to a given original policy. In the change-impact analysis phase, we have two versions of policy p_1 (an original policy) and p_2 (a revised policy produced in the policy fixing phase). Although p_1 is carefully modified to produce p_2, the modifications may incur unintended changes. Therefore, we conduct change-impact analysis to check if p_1 is correctly modified to produce p_2. If there is any unexpected change, the policy fixing phase and the change-impact analysis phase are iterated until the policy behavior is expected.

A change-impact analysis tool analyzes and traces differences between two policies. More specifically, given two versions of a policy, a change-impact analysis tool outputs counterexamples that illustrate semantic differences between the two policies. Indeed, each counterexample represents a request that evaluates to a different response when applied to the two policy versions. For example, a particular request r evaluates to permit for policy p but the same request evaluates to deny for policy p'. In our framework, change-impact analysis is performed on policies that are undergoing maintenance or updates (e.g., fixing) to avoid accidental injection of faults.

6 Implementation

The implementation of our framework leverages an existing access control policy verification tool called Margrave [10]. Margrave is a tool suite written in PLT Scheme [9] for analyzing access control policies written in XACML. Our implementation uses the generic APIs to efficiently print out all the requests with their corresponding decisions in a summarized format, called `print-out`. `print-out` consists of a sequence of rules to represent each request set (which is represented as a bit string of 1 and 0) and its decision. Each bit represents a certain attribute value. In a request set, a value of 1 denotes that a certain attribute value is used and a value of 0 denotes that a certain attribute value is not used.

To detect inconsistencies induced by multiple roles, our framework parses the given policy behavior (i.e., `print-out`) and collects candidate request sets, each of which has a single attribute value. As a request set contains a single subject attribute value, our framework flips a "0" bit to a "1" bit in subject attribute values in the bit string to generate multiple-duty-related requests. Our framework next checks whether inconsistency responses are produced for an original (indicating a single-duty-related request) and flipped request representation (indicating a multiple-duty-related request). This inconsistency checking refers to the policy behavior report to find the corresponding decision for the request indicated by a flipped request representation. Our framework reports inconsistent responses for the policy authors to inspect.

7 Evaluation

We next present the evaluation conducted to assess our framework.

7.1 Objectives and Measures

In the evaluation, we investigate that our framework can detect various types of inconsistencies, which can cause multiple-duty-related security leakage in the policy under test. We collect the following metric for various types of inconsistencies for each policy under test.

- **dec$_i$-dec$_3$ inconsistency count** (i being 1 or 2). The dec$_i$-dec$_3$ inconsistency count is the count of a request-decision pair (r$_3$, dec$_3$) where dec$_3$ is inconsistent with dec$_i$.

dec$_1$ and dec$_2$ (i.e., dec$_i$) denote the decisions of two single-valued requests r$_1$ and r$_2$ that share the same object values and action values but have different subject values. dec$_3$ denotes the decision of a multi-valued request r$_3$ formed by combining the subject values from r$_1$ and r$_2$, and reusing the object values and action values from r$_1$ or r$_2$.

For example, permit-deny inconsistency specifies that a request r$_1$ is evaluated to be permitted and r$_1$ with an additional subject attribute value s is evaluated to be denied.

7.2 Subjects

We collected XACML policies from various sources to check if a policy contains inconsistencies related to multiple duties. We describe three types of policy structures closely related to inconsistencies caused by multiple duties.

- *Permit policy*. A permit policy consists of permit rules (i.e., rules with permit as their decisions) and one deny fall-through rule (if any) at last. A deny fall-through rule is to deny requests that are not applicable (satisfied) by the preceding permit rules.

- *Deny policy*. A deny policy consists of deny rules (i.e., rules with deny as their decisions) and one permit fall-through rule (if any) at last. A permit fall-through rule is to permit requests that are not applicable (satisfied) by the preceding deny rules.

- *Hybrid policy*. A hybrid policy consists of permit rules and deny rules. In a hybrid policy, deny rules and permit rules are mixed and given priorities based on factors such as its location in a policy structure and the used combining algorithms.

In many cases, the policy authors design an access control policy as either a permit policy or a deny policy. In permit or deny policies, as one type of rules dominates, the PDP often resolves conflicting decisions to only one type (`Deny` or `Permit`) of decisions decided based on the combining algorithms; the policies may reveal either permit-deny inconsistencies or deny-permit inconsistencies but not both. In contrast, a hybrid policy may show both permit-deny inconsistencies and deny-permit inconsistencies based on its complex policy structure and combining algorithms.

In our evaluation, we used 11 XACML policy subjects. Figure 7 summarizes the characteristics of each policy. Columns 1-9 show the subject names, the policy types, the number of policy sets, policies, permit rules, deny rules, and distinct attribute id-value pairs in the subject, resource, and action attributes in the policy, respectively.

The `demo-5` policy is modified from the XACML policies in Fedora[1] by preserving its original behavior. Fedora

[1]`http://www.fedora.info`

Policy	Policy type	# set	# pol	# permit rules	# deny rules	# sub	# res	# act
demo-5	deny	0	1	1	2	6	3	2
CodeA	permit	2	2	2	0	3	2	2
CodeB	permit	3	2	2	0	3	3	3
CodeC	permit	8	4	4	0	3	2	3
CodeD	permit	11	5	5	0	4	2	3
mod-CodeD	hybrid	11	5	3	2	4	2	3
pluto	permit	0	1	21	1	4	90	1
freeCS	deny	0	1	2	5	4	4	3
gradeSheet	permit	0	1	13	1	11	11	6
weirdX	hybrid	0	1	3	3	2	5	5
health-care	hybrid	0	1	13	3	6	13	8

Figure 7. Subject policies used in the evaluation

Policy	DENY TO PERMIT	PERMIT TO DENY	NA TO DENY	NA TO PERMIT	DENY TO NA	Permit to NA
demo-5	0	0	0	0	0	0
CodeA	0	0	0	5	0	0
CodeB	0	0	0	9	0	0
CodeC	0	0	0	7	0	0
codeD	0	0	0	8	0	0
mod-codeD	2	2	2	6	0	0
pluto	348	0	0	0	0	0
freeCS	0	5	0	0	0	0
gradeSheet	24	0	0	0	0	0
weirdX	0	1	0	0	0	0
health-care	10	2	0	0	0	0

Figure 8. Evaluation results

is open source software that gives organizations a flexible service-oriented architecture for managing and delivering digital content. Four of the policies, namely `CodeA`, `CodeB`, `CodeC`, `CodeD`, are examples[2] used by Fisler et al. [10].

The `mod-CodeD` policy are modified versions of the `CodeD` policy by inverting decisions in some rules or adding rules. In particular, we set all combining algorithms of the `mod-CodeD` policy as the first-applicable algorithm. The `pluto` policy is used for the ARCHON[3] system. ARCHON is a digital library that federates physics collections with varying degrees of meta data richness. Three of the policies, namely `freeCS`, `gradeSheet`, `weirdX`, are policies used by Birgisson et al. [6]. The `health-care` policy is an RBAC policy used by Stoller et al. [16]. As its original policy is not written in XACML, we specified its policy behaviors in XACML. We used the first-applicable algorithm as its rule combining algorithm.

7.3 Results

We next present the evaluation results of our framework. The second to the last columns of Figure 8 show the number of inconsistencies of deny-permit, permit-deny, nonapplicable-deny, nonapplicable-permit, deny-nonapplicable, and permit-nonapplicable detected in each policy.

[2]http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/college
[3]http://archon.cs.odu.edu/

As shown in Figure 7, six of our subjects (`CodeA`, `CodeB`, `CodeC`, `CodeD`, `pluto`, and `gradeSheet`) are permit policies. The first four policies, namely `CodeA`, `CodeB`, `CodeC`, and `CodeD`, describe only permit rules. These policies do not include any deny rules. In such a case, if multiple permit rules can be applicable, an expected decision is consistent as "permit" and no deny-permit inconsistencies or permit-deny inconsistencies are found. In contrast, the `pluto` and `gradeSheet` policies use the `permit-overrides` combining algorithm and include permit rules and one fall-through deny rule at the end of the policy. This deny rule is given lower priority than other (earlier-placed) permit rules. Therefore, these policies may contain deny-permit inconsistencies, where a request (with a single role) to be applicable on only the deny rule and would be evaluated by other (earlier-placed) permit rules with an additional role on the request. From Figure 8, we observe that our framework detected 348 and 24 deny-permit inconsistencies for the `pluto` and `gradeSheet` policies, respectively.

In the `gradeSheet` policy, we observe that one of its deny-permit inconsistencies indicates a potential fault: a student cannot write grades; however, a student (holding a TA) can write grades. If a student works as a TA in the same course that the student is enrolled in, the student may write her own grades.

For deny policies, `demo-5` and `freeCS` include deny rules that override permit rules and can contain permit-deny inconsistencies. In our subject policies, `demo-5` and `freeCS` are categorized as deny policies. We observed that `freeCS` contains 5 permit-deny inconsistencies for policy authors to inspect.

The `mod-codeD`, `weirdX`, and `health-care` policies can contain both deny-permit inconsistencies and permit-deny inconsistencies. The `mod-codeD` policy is a hybrid policy modified from permit policies and this modification causes inconsistencies to be observed. The `mod-codeD` policy contains 2 deny-permit inconsistencies and 2 permit-deny inconsistencies. Hybrid policies are complex in their policy structure (in comparison with permit or deny policies) where permit and deny rules are entangled in various ways; specifying rules for multiple-duty related requests is not trivial and various inconsistencies are to be inspected for correctness. The `weirdX` policy does not have any reported inconsistencies. We observe that the policy uses only two duties (in subject attributes) and these two duties do not cause any conflicting decisions for the same resource and action.

The `health-care` policy includes permit rules and deny rules and uses the `first-applicable` algorithm. We next illustrate one permit-deny inconsistency in the `health-care` policy. Two roles, `Manager` and `Doctor`, have conflicting decisions on viewing private notes of patients: `Manager` cannot view private notes; however, `Doctor` can view private notes. If a person takes the two roles at the same time, the policy evaluates such a request to be denied considering `Manager` is given higher priority than `Doctor`. However, we often do not want to restrict doctor's privilege when a doctor can be a manager. Therefore, we observe that the permit-deny inconsistency indicates a likely fault and we can fix this fault by specifying rules related to `Doctor` before rules related to `Manager`.

We observed that five of our subjects (`CodeA`, `CodeB`, `CodeC`, `CodeD`, and `mode-CodeD`) contain nonapplicable-deny or nonapplicable-permit inconsistencies in Figure 8. Each of these policies does not include a fall-through deny (permit) rule at the end of the policy. Due to lack of such a fall-through rule, it is likely that some requests (including a single role) do not match any rules and the PDP produces a nonapplicable decision. With an addition of a role, such requests can include multiple roles and may find matching rules to cause the PDP to produce a permit or deny decision. A nonapplicable decision is often regarded as a deny decision and policy authors carefully inspect such inconsistencies to detect multiple-duty-related security leakage. However, as the rest of policies in our evaluation include one fall-through deny (permit) rule, the policies do not contain nonapplicable-deny or nonapplicable-permit inconsistencies.

In summary, the results indicate that a policy type is closely correlated to the detection of multiple-duty-related security leakage. Because only one type of decision is given higher priority in a permit (deny) policy, permit-deny and deny-permit inconsistencies cannot be found together in the same policy. In a hybrid policy, we can inspect each permit-deny and deny-permit inconsistency to determine if these inconsistencies are caused by multiple-duty-related security leakage.

8 Related Work

To help ensure the correctness of policy specifications, researchers and practitioners have developed formal verification tools for policies. Several policy verification tools are developed specifically for firewall policies. Al-Shaer and Hamed [4] developed the Firewall Policy Advisor to classify and detect policy anomalies. Yuan et al. [17] developed the FIREMAN tool to detect misconfiguration of firewall policies. The multiple-duty-related security leakage generally does not exist in these firewall policies.

There are several verification tools available for XACML policies. Hughes and Bultan [11] translated XACML policies to the Alloy language [12], and checked their properties using the Alloy Analyzer. Zhang et al. [19] developed a model-checking algorithm and tool support to evaluate access control policies written in *RW* languages, which can be converted to XACML [18]. Fisler et al. [10] developed Margrave, which can verify XACML policies against the given properties and perform change-impact analysis on two versions of policies. Schaad and Moffett [15] leverage Alloy [12] to check that role-based access-control policies do not allow roles to be assigned to users in ways that violate SoD constraints. These XACML policy verification tools can be used when policies such as SoD properties are explicitly specified by the policy authors.

Different from these preveous tools, which require properties or SoD constraints to be specified, our framework can be seen as a type of anomaly detection by detecting suspicious cases for the policy authors to inspect; our framework does not require the policy authors to explicitly specify properties. Instead, our framework detects suspicious cases for inspection.

9 Conclusions

Access control policies such as those written in XACML could be amenable to multiple-duty-related security leakage, which grants unauthorized access to a user when the user takes multiple duties (e.g., multiple roles in role-based access control policies). We have developed a novel framework that detects cases where a user taking multiple roles (e.g., r_1 and r_2) is given a different access decision than the user taking an individual role (e.g., r_1 and r_2, respectively). We have implemented the framework and conducted experiments on XACML policies collected from various sources.

Our empirical results show that our framework can effectively pinpoint potential multiple-duty-related security leakage for policy authors to inspect.

XACML is designed to be flexible by allowing a request with multiple roles (duties). Potential multiple-duty-related security leakage can be awkwardly addressed by restricting only a single role per request without this flexibility. However, it is still not trivial to select only a single role when a subject can be assigned with multiple roles. Such selection requires resolving the most probable role among all the possible ones. Therefore, allowing a request with multiple roles is necessary in practice and our framework can help detect multiple-duty-related security leakage in XACML policies.

Acknowledgment

This work is supported in part by NSF grant CNS-0716579 and its NIST supplement.

References

[1] OASIS eXtensible Access Control Markup Language (XACML). `http://www.oasis-open.org/committees/xacml/`, 2005.

[2] Sun's XACML implementation. `http://sunxacml.sourceforge.net/`, 2005.

[3] XACML.NET. `http://mvpos.sourceforge.net/`, 2005.

[4] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proc. 23rd Conf. IEEE Communications Soc. (INFOCOM 2004)*, pages 2605–2616, 2004.

[5] A. Anderson. XACML profile for role based access control (RBAC). OASIS Committee Draft 01, 2004.

[6] A. Birgisson, M. Dhawan, Úlfar Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Proc. 15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 223–234, 2008.

[7] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proc. International Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, pages 18–38, 2001.

[8] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[9] R. B. Findler, J. Clements, M. F. Cormac Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A Progamming Environment for Scheme. *Journal of Functional Programming*, 12:159–182, 2002.

[10] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. 27th International Conference on Software Engineering (ICSE 2005)*, pages 196–205, 2005.

[11] G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, 2004.

[12] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2001)*, pages 62–73, 2001.

[13] M. Lorch, D. Kafura, and S. Shah. An XACML-based policy management and authorization service for Globus resources. In *Proc. International Workshop on Grid Computing (GRID 2003)*, pages 208–212, 2003.

[14] T. Moses, A. Anderson, S. Proctor, and S. Godik. XACML Profile for Web-Services (WSPL). OASIS Working Draft, 2003.

[15] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proc. 7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, pages 13–22, 2002.

[16] S. D. Stoller, P. Yang, C. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 445–455, 2007.

[17] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for FIREwall Modeling and ANalysis. In *Proc. 2006 IEEE Symposium on Security and Privacy (S&P 2006)*, pages 199–213, 2006.

[18] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *Proc. 2004 ACM Workshop on Formal Methods in Security Engineering (FMSE 2004)*, pages 56–65, 2004.

[19] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *Proc. 8th Information Security Conference (ISC 2005)*, pages 446–460, 2005.