Test Selection and Augmentation of Regression System Tests for Access Control Policy Evolution

***Abstract*—As security requirements of software often change during development and maintenance, developers may modify policies according to evolving requirements. In order to increase confidence that the modification of policies is correct and does not introduce unexpected behavior, developers conduct regression testing. For regression testing, rerunning all of existing system test cases is costly and time-consuming, especially for large-scale systems. In order to address this issue, we propose an efficient regression-test-selection approach, which selects every test case that may reveal a fault caused by policy changes. Our approach includes three techniques: the first one is based on mutation analysis (that converts each rule's decision in turn and executes and finds test cases related to each rule), the second one is based on coverage analysis (that records which rules are evaluated by executing each test case), and the third one is based on evaluated decisions of requests issued from test cases. However, selected test cases may not evaluate (i.e., cover) all of modified policy behaviors, which refer to rules in a policy impacted by policy change. To address this issue, we propose a test augmentation technique. The technique modifies existing test cases to generate additional test cases to cover not-covered-impacted-rules with existing test cases. We evaluate our approach on three real world Java programs each interacting with policies. Our evaluation results show that our test selection techniques reduce up to 51%$\sim$97% of test reduction for a modified version with given 5$\sim$25 policy changes. Our evaluation results show that our test augmentation technique generates additional test cases to cover 100% of the impacted rules.**

***Keywords*-access control policy evolution; regression testing; test selection; test augmentation; reliability**

I. Introduction

Access control is one of the privacy and security mechanisms that authorizes only legitimate users for access to critical information. Access control is governed by access control policies (policies in short), each of which includes a set of rules to specify which subjects are permitted or denied to access which resources in which conditions. To facilitate specify and maintain policies separately from entangled code, system developers specify and enforce policies independently from actual functionality (i.e., business logic) of a system.

In the system, program code, which represents the actual functionality, interacts with a policy through a security component, called policy decision point (PDP). Consider that the program code consists of methods Ms including policy enforcement points (PEPs). PEPs are functionalities that require decisions (e.g., permit or deny) whether a given subject can have access on critical information.

Typically, PEPs in Ms formulate an access request that specifies that a subject would like to have access on critical information. The PEPs next submit the request to a PDP, which evaluates the request against the policy and determines whether the request should be permitted or denied. Finally, the PDP formulates and sends the decision to PEPs to proceed.

As security requirements of software often change during development and maintenance, developers may modify policies to comply with requirements. In order to increase confidence that the modification of policies is correct and does not introduce unexpected behavior, developers periodically conduct regression testing. For example, new security requirements include new security concerns to be added into a policy. Developers may change policies without changing program code related to actual system functionality. In such a situation, validating and verifying program code and policies together after policy changes increases confidence of correct behaviors of a given system.

In this paper, we focus on the regression testing problem in the context of policy evolution. For policy evolution, regression testing is important because policy behavior changes may result in unexpected behaviors of program code, these behaviors can even be undesirable. The typical regression testing for program code interacting with a policy is as follows. Given a program code and a policy P, the developers prepare initial system test cases, where each test case maps to rules (in the policy) exercised by the test case. Given P and its modified policy P', the developers compare impacts of P and P' to reveal different policy behaviors, which are "dangerous" portions to be validated with test cases. For validating this "dangerous" portion, the developers often select only test cases (from test cases for P), which exercise the dangerous portions of P'.

For regression testing, instead of writing new system test cases, developers often write initial system test cases and reuse the test cases in practice. The naive regression testing strategy is to rerun all system test cases. However, this strategy is costly and time-consuming, especially for large-scale systems. Moreover, if the number of the initial system test cases is large, this strategy may require a significant time for developers to conduct testing. In order to reduce the cost of regression testing, developers often adopt regression test selection, which selects and executes only test cases to expose different behaviors across different versions of the system. This approach also requires substantial costs to select and execute such system tests from the initial test cases that could reveal faults introduced by the changes. If regression-test-selection cost is smaller than rerunning all of initial system test cases, test selection helps reduce time-consuming task in validating whether the modification is correct.

In order to address the issue, we propose a regression-test-selection approach, which selects every test case that may reveal different behaviors in program code impacted by policy changes. In general, our approach automatically compares an original policy P and its modified policy P' to detect different policy behaviors. In the policy context, different policy behaviors refer that, given a request, its evaluated decisions (for P and P', respectively) are different. These different policy behaviors are dangerous portions to be validated. As a policy consists of rules, we refer such dangerous portions as rules impacted by policy changes. We next find test cases to reveal such policy changes.

Our approach includes three techniques. The first one is based on mutation analysis (that converts each rule's decision in turn and executes and finds test cases related to each rule), the second one is based on coverage analysis (that records which rules are evaluated by executing each test case), and the third one is based on evaluated decisions of requests issued from test cases. Our approach next selects only test cases to execute program code impacted by policy changes.

While test-selection techniques are useful for selecting test cases for program code impacted by policy changes, these test cases may not sufficiently cover all the rules in the policy impacted by the changes. We develop a test-augmentation technique, which complements our test-selection techniques. We first use the regression-test-selection approach before using the test-augmentation technique to achieve additional policy coverage for not-covered-impacted-rules N_r by existing test cases. Our technique modifies existing test cases to generate test cases, which issue requests to cover N_r with minimal test cases changes. For example, attribute change (e.g., subject role change from $Student$ to $Professor$) may cover N_r. We then verify that augmented test cases increase covering N_r.

This paper makes five main contributions:

- We develop a test selection approach to select every test case from existing test cases to test program code impacted by policy changes. Our approach uses three techniques; the first one is based on mutation analysis, the second one is based on coverage analysis, and the third one is based on evaluated decisions of requests issued from test cases.
- We develop a test augmentation technique to generate additional new test cases to cover not-covered-impacted-rules with selected test cases by the preceding techniques.
- We evaluate our approach on three real world Java programs interacting with policies. Our evaluation results show that our test selection techniques reduce up to 51%∼97% of test reduction for a modified version with given 5∼25 policy changes.
- Among our three techniques, our results show that the third technique is the most efficient compared with the first and the second techniques in terms of elapsed time. The third technique is 43 and 8 times faster than the first and the second techniques, respectively.
- Our evaluation results show that our test augmentation technique generate additional test cases to cover 100% of the impacted rules by policy changes.

The rest of the paper is organized as follows. Section II presents background information about policy-based software systems, policy context, and regression testing. Section III present our approach. Section IV presents our implementation. Section V describes the evaluation results where we apply our approach on three projects. Section VI discusses issues. Section VII discusses related work. Section VIII concludes the paper.

II. Background

A. Policy-based Software Systems: PEP-PDP separation

Policy-based software systems are regulated by access control policies that specify the different actions that subjects are allowed to perform on system resources. The policy is enforced by one or multiple Policy Enforcement Points (PEP) at the application level. A given PEP receives a subject access request and sends it to a Policy decision Point (PDP) that evaluates access control requests against policy rules. The evaluation process involves resolving all the rules in the policy. Current access control implementations follow a design strategy that separates between the PEP and the PDP, this separation is advocated mainly for two reasons:

- Specifying the policy independently from the program code enables to save time, cost and effort during the maintenance phase since developers are not required to update the application code at each policy update.
- When the policy is not hard-coded into the application, it can be specified in a standardized language and transported through many different systems platforms.

Access control policies can be specified in many languages like EPAL [1] or XACML [2]. In this paper, we consider XACML polices described in Section II-B.

B. XACML Policies

This section describes the background in the field of XACML policies, policy models, and regression testing.

XACML (eXtensible Access Control Markup Language) [2] is a language specification standard published by the Organization for the Advancement of Structured

```
 1 <Policy PolicyId="Library Policy" RuleCombAlgId="Permit-overrides">
 2  <Target/>
 3    <Rule RuleId="1" Effect="Permit">
 4      <Target>
 5        <Subjects><Subject> BORROWER </Subject></Subjects>
 6        <Resources><Resource> BOOK </Resource></Resources>
 7        <Actions><Action> BORROWERACTIVITY </Action></Actions>
 8      </Target>
 9     <Condition>
10        <AttributeValue> WORKINGDAYS </AttributeValue>
11      </Condition>
12    </Rule>
13    <Rule RuleId="2" Effect="Deny">
14      <Target>
15        <Subjects><Subject> BORROWER </Subject></Subjects>
16        <Resources><Resource> BOOK </Resource></Resources>
17        <Actions><Action> BORROWERACTIVITY </Action></Actions>
18      </Target>
19     <Condition>
20        <AttributeValue> HOLIDAYS </AttributeValue>
21      </Condition>
22    </Rule>
23    <Rule RuleId="3" Effect="Permit">
24      <Target>
25        <Subjects><Subject> SECRETARY </Subject></Subjects>
26        <Resources><Resource> BOOK </Resource></Resources>
27        <Actions><Action> FIXBOOK </Action></Actions>
28      </Target>
29     <Condition>
30        <AttributeValue> MAINTENANCEDAY </AttributeValue>
31      </Condition>
32    </Rule>
33      <!-- A final, "fall-through" rule that always Denies -->
34    <Rule RuleId="FinalRule" Effect="Deny"/>
35 </policy>
```

Figure 1. An example policy specified in XACML

Information Standards (OASIS). An XACML access control specification consists of a policy set and a policy combining algorithm. XACML supports various policy models such as Role Based Access Control (RBAC) [5], [6] and Organization-Based Access Control (OrBAC) [4]. For RBAC and OrBAC, various job functions are associated to roles (e.g., staff or employee) that a user possesses. Permissions or denials to take an action on certain objects are assigned to specific roles (instead of users). OrBAC supports extra features such as role, activity (i.e., action), and view (i.e., object) hierarchy. Figure 1 is an OrBAC policy example where roles such as “Secretary” and “Borrower” are associated with subject attributes and policy decisions are made through specific roles.

Let $Conditions$ be the set of the environmental context such as working days or holidays, a file size, and a user's task authorization level. Let S, O, A, and C denote all the subjects, objects, actions, and $Conditions$, respectively, in an access control system.

An XACML policy consists of a *policy set*, which consists of *policy sets* and *policies*. A *policy* consists of a sequence of *rules*, each of which specifies under what conditions C subject S is allowed or denied to perform action A (e.g., read) on certain object (i.e., resources) O in a system.

A rule R is of the following form:

$$R \subseteq S \times A \times O \times C \longrightarrow Dec$$

where Dec denotes a decision, which is either permit or deny. A user's request is evaluated against a policy. A request matches with a rule if the request satisfies the rule's attributes. Then the rule's decision can be returned. Formally, a request Q in the following form matches R:

$$Q \subseteq S_q \times A_q \times O_q \times C_q \text{ where } S_q \subseteq S,\ A_q \subseteq A,\ O_q \subseteq O, \text{ and } C_q \subseteq C.$$

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`. The *deny-overrides algorithm* returns `Deny` if any rule evaluation returns `Deny` or no rule is applicable. The *permit-overrides algorithm* returns `Permit` if any rule evaluation returns `Permit`. Otherwise, the algorithm returns `Deny`. The *first-applicable algorithm* returns whatever the evaluation of the first applicable rule returns. The *only-one-applicable* algorithm returns the decision of the only applicable rule if there is only one applicable rule, and returns error otherwise.

Figure 1 shows a policy specified in XACML. The policy consists of three XACML rules used in our real-life library access control policy subject, called LSM [13]. Note that we simplified XML formats to reduce space for this example. Lines 3-12 describe a rule that `borrower` is permitted to `borroweractivity` (e.g., borrowing books) `book` in `working days`. Lines 13-22 describe a rule that `borrower` is denied to `borroweractivity` `book` in `holidays`. Lines 23-32 describe a rule that `secretary` is permitted to `fixing` for `book` in `maintenance day`. As these rules are combined using the permit-overrides algorithm, the *permit-overrides algorithm* returns the `Permit` if a request matches at least one of rules with permit decisions. For example, if rules 2 and 3 are applicable to a request, the decision evaluated from rule 3 is given higher priority than that of rule 2.

C. Regression Testing

Software testing [14] refers to the activity of generating Tests Cases to verify the conformity of output results provided by a program to the expected output that meets its functional and non functional requirements. With the increasing complexity of software systems, this activity is gaining more and more interest in the research field and aiming to establish a trade-off between cost, time and quality.

Software is subject to changes that occur at the design stage or in later stages at the deployment or maintenance phases. These changes are usually supposed to meet changes in the requirements or to overcome errors that can be detected in later stages of software life cycle. Regression testing refers to the research field that is interested in retesting the system to verify that the new changes have not altered the initial system behavior. As highlighted by Rothermel et al. in [15], regression testing is defined like the following: “Given a program P, a modified version P', and a set T of test cases used previously to test P, regression analysis and testing techniques attempt to make use of a subset of T to gain sufficient confidence in the correctness of P' with respect to behaviors from P retained in P'”.

The main objectives of regression testing is to reduce the costs from rerunning initial test cases and to maximize the capability of selected tests to detect potential errors that may be induced by changes.

To the best of our knowledge, there is no prior research work on regression testing with regards to policy changes in policy-based software systems. To help developers reduce cost in testing for such systems, we propose novel test selection and augmentation approach. In order to test policy-based software system, developers deploy test cases that trigger the rules in the policy. Our approach selects test cases by verifying that the changed parts of the policy do not reveal unexpected behaviors in a given system. Whenever a policy P is updated to a new policy P', existing test cases impacted by the policy changes need to be tested to validate that the policy changes do not introduce a negative impact on the initial implementation of a system.

III. Approach

As manually selecting test cases for regression testing is tedious and error-prone, we have developed three techniques to automate tests selection for regression in policy evolution. Among existing test cases, the objective is to select all test cases for regressing testing as follows.

Our approach takes two versions of program code, which interact with v_1 (original) and v_2 (new) access control policy, respectively. The existing test cases are taken as an input; these tests invoke methods in program code. We analyze given program code and policies to select *only* test cases for regression testing in case of policy evolution. Among given test cases, our selected test cases invoke methods to reveal changed policy behaviors between v_1 and v_2.

Our test selection techniques cannot guarantee sufficiency of regression testing (i.e., whether these tests are sufficient to reveal all of changed policy behaviors). We have developed a technique to automatically augment new test cases to satisfy sufficiency as follows. We measure sufficiency of regressing testing with our selected test cases based on rule coverage criteria [12], which measures which rules in a policy are evaluated (i.e., covered) during test case execution. For not-covered changed policy behaviors (i.e., rules), our approach automatically generates test cases to cover such behaviors.

Formally, C denotes a program code, which interacts with an access control policy P. P_m is the modified version of P. T denotes an initial test suite for C. Our first step involves the regression test selection. We select $T' \subseteq T$ where T' is a set of test cases. T' execute on C and reveal changed policy behaviors between P and P_m. In the second step, we measure rule coverage of changed policy behaviors of P and P_m with T'. If we find not-covered policy behaviors, we augment T' and create T'' to cover all changed rules.

```
<Policy PolicyId="Library Policy" RuleCombAlgId="Permit-overrides">
 <Target/>
   <Rule RuleId="1" Effect="Deny">
     <Target>
       <Subjects><Subject> BORROWER </Subject></Subjects>
       <Resources><Resource> BOOK </Resource></Resources>
       <Actions><Action> BORROWERACTIVITY </Action></Actions>
     </Target>
    <Condition>
       <AttributeValue> WORKINGDAYS </AttributeValue>
     </Condition>
   </Rule>
...
</policy>
```

Figure 2. An example mutant policy by changing $R1$'s rule decision (i.e., effect)

Algorithm 1: Test Selection based on Mutation Analysis Algorithm

```
TestSelection1($P$, $P_m$, $T$): $T'$
Input: XACML Policy $P$, modified policy $P_m$, Initial Test Cases $T$
Output: $T' \subseteq T$ where $T'$ is the subset of $T$ selected for use in regression testing $P_m$
$T'$=Ø
/*Rule-test set-up phase*/
for each rule $r_i$ in Policy $P$ do
  $T_{r_i}$=Ø where $T_{r_i} \subseteq T$ are the tests correlated to $r_i$
  /*We mutate the policy $P$ by creating a rule-decision change (RDC) on $r_i$ to get $P_{r_i}$*/
  $P_{r_i} \xleftarrow[RDC(r_i)]{} P$
  Execute $T$ with $P_{r_i}$
  for each $t$ in $T$ do
    Let $E(t)$ be the test execution result,
    $E(t) = Success, Failure$
    if $E(t) \leftarrow$ Failure then
      $T_{r_i} \leftarrow T_{r_i} \cup t$
    end if
  end for
  Map($r_i$,$T_{r_i}$)
end for
/*Test selection phase*/
$\{r_m\}_{i=1..m} \leftarrow diff(P, P_m)$
for Each rule $r_i$ in $\{r_m\}_{i=1..m}$ do
  $T' \leftarrow T' \cup T_{r_i}$
end for
return $T'$
```

We next describe our proposed three test selection techniques and test augmentation technique.

A. Test Selection based on Mutation Analysis

Our first proposed test selection technique uses mutation analysis to select test cases as follows. The approach needs a preliminary step which is necessary to establish a rule-test correlation. We next describe rule-test correlation and test selection steps.

Rule-Test Correlation Setup. Given a policy P, we create its rule-decision-change (RDC) mutant policy Pr_i by changing decision (e.g., Permit to Deny) of one single rule r_i in P. An example of a mutated policy is shown in Figure 2. In this policy, original Rule 1's decision Permit is changed to Deny. The technique finds affected test cases for this rule decision change. We execute test cases T on program code for P and Pr_i. To detect changed policy behaviors, the technique monitors responses of evaluated requests formulated from test cases T. The test cases, which evaluate different policy decisions against P and P', enable to map rule r_i to test cases $t \in T$. The preliminary step ends by establishing a correlation between each rule in P and corresponding tests $t \in T$ that trigger this rule.

Test Selection. The selection of test cases for regression on P and its modified policy P_m starts by conducting change impact analysis of P and P_m to find which rules' decision are changed. Once these rules are identified, we use the mapping established in the preliminary step to select the subset of test cases which are correlated with changed rules.

Algorithm 1 describes our algorithm used for the technique. While the technique can quickly select test cases, the technique requires rule-test correlation setup (in the preliminary step), which could be costly in terms of execution time. Given n rules in P, we execute T for 2×n times. As the preliminary step is applied for only existing rules R in P, our technique requires addition of rule-test correlation for newly added rules R_n where $R_n \notin R$ in P_m. In addition, if a new system test is introduced, we execute this test for 2×n times. However, an important benefit is that we are enabled to conduct rule-test set-up once before encountering policy changes in terms of correlated rules.

B. Test Selection based on Coverage Analysis

Our previous technique finds correlation of all of existing rules N in a given policy with the test cases. To reduce such correlation setup efforts, we develop a technique to correlate only rules, which can be evaluated (i.e., covered) by test cases. Our intuition is that test cases may interact only with a small number of rules in a policy instead of all the rules in the policy. We next describe rule-test correlation step.

Rule-Test Correlation Setup. Given a policy P, Thus we execute test cases T on program code that interacts with P. Our technique monitors which rules in a policy are evaluated with requests formulated from test cases T. Then, we establish correlation between test cases and evaluated (i.e., covered) rules in P.

Test Selection. Once the mapping test-rule is established, we proceed test selection step described in the first approach with the results of change impact analysis of P and P_m. The results include information to show which rules' decision have changed. This test selection maps test cases with those rules to constitute the subset of existing test cases.

Algorithm 2: Test Selection based on Coverage Analysis Algorithm

```
*TestSelection2*($P$, $P_m$, $T$): $T'$
**Input:** XACML Policy $P$, modified policy $P_m$, Initial Test Cases $T$
**Output:** $T' \subseteq T$ where $T'$ is the subset of $T$ selected for use in regression testing $P_m$
$T'$=Ø
/*Rule-test set-up phase*/
**for** Each test case $t$ in $T$ **do**
    Execute $t$ with $P$
    $MAP$=Map($t$,$\{r_p\}_{i=1..p}$) where $\{r_p\}_{i=1..p}$ are the rules
    in $P$ that are evaluated (i.e., covered) by $t$
**end for**
/*Test selection phase*/
$\{r_m\}_{i=1..m} \leftarrow diff(P, P_m)$
**for** Each rule $r_i$ in $\{r_m\}_{i=1..m}$ **do**
    $T' \leftarrow T' \cup T_{r_i}$
**end for**
return $T'$
```

Algorithm 2 describes our algorithm used for the technique. An important benefit of this technique is to reduce cost in terms of mutation analysis and execution time. This technique does not require generating mutants by changing rule's decision in turn. Moreover, the technique can significantly reduce execution time. While the technique can quickly select system tests in the second step, the technique requires rule-test correlation setup (in the preliminary step), which could be costly in terms of execution time. Consider that requests Rs are formulated from test cases interact only n_1 rules ($n_1 \leq$ n) in a policy. We execute T only once. Our technique requires addition of rule-test correlation for newly added rules R_n where $R_n \notin R$ in P_m as the same with the previous technique.

C. Test Selection based on Recorded Request Evaluation

To reduce such correlation setup efforts in the previous techniques, we develop a technique, which does not require rule-test correlation setup. Instead of correlation, our technique records test cases and their issued requests as a preliminary step. More specifically, our technique executes test cases T on program code for P and records all requests issued to policy decision point (PDP) for each test case. For test selection, our technique evaluates all issued requests against P and P_m and selects the test subset of requests (with corresponding system test cases) that engender different decisions for two different policy versions.

Algorithm 3 describes our algorithm used for the technique. The current approach requires the execution of system

Algorithm 3: Test Selection based on Recorded Request Evaluation

```
TestSelection3($P$, $P_m$, $T$): $T'$
Input: XACML Policy $P$, modified policy $P_m$, Initial Test Cases $T$
Output: $T' \subseteq T$ where $T'$ is the subset of $T$ selected for use in regression testing $P_m$
$T'$=Ø
$R_{T'}$=Ø where $R_{T'}$ are the requests corresponding to $T'$
Execute system requests $R$
for each request $Re$ in $R$ do
    if $decision(Re/_P) \neq decision(Re/_{P_m})$ then
        $R_{T'} \leftarrow R_{T'} \cup Re$
    end if
end for
$T' \leftarrow R_{T'}$
return $T'$
```

test cases T only once. Moreover, while the two previous techniques are white-box testing since access control policies are available, the present technique does not require the availability of access control policies. This can present a considerable advantage when developers don't want to reveal their access control policies.

D. Test Augmentation

While test-selection techniques are useful for selecting test cases for program code impacted by policy changes, these test cases may not sufficiently cover all the rules in the policy impacted by the changes. We develop a test augmentation technique, which complements our test-selection techniques by generating additional test cases to cover not-covered-impacted-rules N_r by existing test cases.

Our augmentation technique first analyzes change impact analysis between P and P' to find rules impacted by policy changes. The technique finds N_r by evaluating selected test cases and monitoring which rules are evaluated during test case execution. For N_r, we next generate requests, which can cover N_r. We then analyze existing test cases to create test cases, which issue the requests to cover N_r with minimal changes in test cases. For example, attribute change (e.g., subject element change from $Student$ to $Professor$) may cover N_r. We then verify that augmented test cases increase coverage of N_r.

IV. Implementation

Our implementation (written in Java) includes five components: regression simulator, test-rule correlation, request-record, test selection, and test augmentation. To simulate regression in access control policies, we used three types of mutants injected in the policies; the first one is RMR (Rule Removal), RA (Rule Addition), and CRE (Change Rule Effect) mutants. The regression simulator analyzes a policy and injects one of such changes. For RMR, given a randomly selected rule in a policy P, the regression simulator removes the rule. For CRE, given a randomly selected rule in a policy P, the regression simulator changes the decision of the rule. For RA, the regression simulator adds a randomly generated rule with random attributes collected from P in a random place. The simulator can inject more than one type of changes to P for simulating regression on P.

For test selection based on mutation analysis technique, our test test-rule correlation mutates a given policy using CRE , executes all of test cases for each mutated policy, and monitors changed behaviors by comparing test results of the test cases with an original policy or its mutated policy. The component also automatically compares the testing results (on the policy and the mutated policies) and logs killed mutant information if the results are inconsistent. We use such killed mutant information for rule-test correlation. For test selection based on coverage analysis technique, we execute all of test cases once and monitors which rules are evaluated for each test case execution.We use such coverage information for rule-test correlation.

For the test selection based on recorded request evaluation, the request-record component records all requests issued by test cases and evaluated rules in a policy.

For our first two techniques, the test selection component requires change impact analysis to show changed policy behaviors of two versions of policies. The implementation leverages an existing access control policy verification tool called Margrave [7]. Margrave is a tool suite for analyzing access control policies written in XACML. Given two versions of policies, P and P', our implementation uses the generic APIs of Margrave to print out all the policy changes with their corresponding changed decisions in a summarized format. Our first two techniques analyze the results of Margrave and find which rules R_i impacted by policy changes. We then find all of test cases R_t impacted by R_i.

For the test selection based on recorded request evaluation, the component does not require change impact analysis. The test selection component log recorded requests to reveal different evaluated decision against P and P'.

The test argumentation component compares not-covered policy behaviors r_1s (i.e., represented by requests) with requests r_2s issued from existing test cases. The test argumentation computes similarity between r_1 and r_2 to decide applicability of modification for test argumentation. Given recommended test cases with high similarity score, we manually modify the test cased to cover not-covered policy behaviors. If not applicable for modification, the component generates a test cases to issue such requests to cover not-covered policy behaviors.

Table I
SUBJECTS USED IN OUR EVALUATION

Subjects	LOC	# Test Methods		AC Rule Coverage		
		# Total	# Security Tests	# Cov	# Not-Cov	% Cov
LMS	3749	46	29	42	0	100
VMS	3734	52	10	13	106	12
ASMS	7836	93	91	109	21	83

Table II
POLICY STATISTICS USED IN OUR SUBJECTS

Subject	Attributes				# AC Rules		
	# Subjects	# Actions	# Resources	# Conditions	# Explicit	# Implicit	# Total
LMS	6	10	3	4	42	678	720
VMS	7	15	3	3	106	839	945
ASMS	8	11	5	4	129	1631	1760

V. EXPERIMENT

We carried out our evaluation on a desktop PC, running Windows 7 with Intel Core i5, 2410 Mhz processor, and 4 GB of RAM. For our test selection approach, we use test selection based on mutation analysis (Mut-$Selection$), test selection based on coverage analysis (Cov-$Selection$), and test selection based on recorded request evaluation (Req-$Selection$) described in Section III.

For regression, we simulate policy evolution by changing/adding/removing rules in access control policies. We do not simulate regression in test code in subjects. For evaluation, our implementation randomly chooses one of regression types, which are RMR, RA, and CRE, and feeds such change in the rule. We configured that our implementation to inject 5, 10, 15, 20, and 25 changes in a policy, respectively. Our evaluation is repeated for 12 times in order to avoid the impact of randomness of changes.

To measure the effectiveness of our three techniques, we measure how many test cases are reduced for regression testing. To measure the efficiency of our three techniques, we conducted evaluation as follows. We compared elapsed time to analyze test-rule correlation analysis, change impact analysis, and test selection by each technique. For the first two techniques, we require test-rule correlation analysis and change impact analysis, which should be done before actual test selection. The objective of this evaluation is to investigate how our three test selection techniques impact performance for subjects. To measure the effectiveness of our test augmentation technique, we measure test coverage with selected and new augmented test cases.

A. Subjects

The subjects include three real-life Java programs each, which interacts with access control policies [13]. We next describe our three subjects.

- Library Management System (LMS) provides web services to manage books in a public library.
- Virtual Meeting System (VMS) provides web conference services. VMS allows users to organize online meetings in a distributed platform.
- Auction Sale Management System (ASMS) allows users to buy or sell items on line. A seller initiates an auction by submitting a description of an item she wants to sell with its expected minimum price. Users then participate in bidding process by bidding the item. For the bidding on the item, users have enough money in her account before bidding.

Our subjects use a Sun PDP [3] to evaluate requests against a policy. Sun PDP is popularly used PDP to evaluate requests. Table I summarizes the basic statistics of our subjects. The first column shows the subject names. Columns 2-5 show the lines of code (“LOC”), the numbers of total test cases (“# Total”), security test cases (“# Security Tests”), rules covered with existing test cases (“# Cov”), rules not-covered with existing test cases (“# Not-Cov”), and the percentage of rule coverage (“% Cov”), respectively. For coverage, we refer rules are explicitly specified rules. In addition, over all of existing test cases, we distinguish security test cases, which issue more than one request to the PDP from test cases without interaction of policies.

Table II summarizes the basic statistics of policies in our subjects. The first column shows the subject names. Columns 2-5 show the numbers of subjects, actions, resources, conditions, explicitly specified rules, implicit rules, and total rules for each policy. The largest one consists of 129 rules. For explicitly specified rules, developers first write rules for denying or permitting access explicitly. Implicit rules refer to rules that are evaluated in a given policy, while the developers do not write such rules. For example, In Figure 1, Line 34 specifies a rule. This rule denies all the requests R_d that do not match with its preceding rules while no rules (i.e., implicit rules) are specified to handle such requests explicitly.

B. Objectives and Measures

In the evaluation, we intend to answer the following research questions:

- RQ1: How many of test cases in a test suite (from an existing test suite) selected by our test selection tech-

Table III

COVERAGE RESULTS OF TEST CASES SELECTED FOR CHANGED POLICY BEHAVIORS FOR EACH POLICY

Subject	Regression - 5			Regression - 10			Regression - 15			Regression - 20			Regression - 25		
	# CT	# Cov	% Cov	# CT	# Cov	% Cov	# CT	# Cov	% Cov	# CT	# Cov	% Cov	# CT	# Cov	% Cov
LMS	5.0	2.4	48.3	9.0	4.8	52.8	13.0	6.3	48.1	17.4	7.8	44.5	20.7	8.8	42.7
VMS	4.9	0.4	8.5	9.3	0.7	7.2	13.8	1.3	9.6	18.9	1.1	5.7	23.8	1.8	7.4
ASMS	5.0	1.9	38.3	9.8	4.8	48.7	14.4	7.3	50.9	18.7	9.4	50.4	23.3	12.1	51.8
Average	4.97	1.58	31.71	9.33	3.39	36.23	13.75	4.97	36.19	18.33	6.08	33.56	22.58	7.56	33.97

Table IV

ELAPSED TIME FOR EACH STEP OF TEST SELECTION TECHNIQUE, AND EACH POLICY

Subject	Mut-Selection			Cov-Selection			Req-Selection	
	TEST-Rule	CIA	Test Selection	TEST-Rule	CIA	Test Selection	Req Collection	Test Selection
LMS	70496	2083	4	5214	2083	4	2096	2
VMS	19771	3333	1	7506	3333	1	1873	2
ASMS	118248	4000	11	22423	4000	11	1064	21
Average	69505	3139	5	11714	3139	5	1678	8

niques? This question helps to show that our techniques can reduce a significant number of test cases in an existing test suit. We also show how many changed policy behaviors are covered with our selected test cases.

- RQ2: What are elapsed time for our techniques to conduct test selection by given subjects? This question helps to compare performance of our techniques by measuring efficiency with regards to elapsed time.
- RQ3: How higher we can achieve additional policy coverage ratio by our test augmentation technique? This question helps to show that our technique can generate/augment test suite to cover 100% of changed policy behaviors. We also compare our approach with random test generation technique to show how effectively our approach augment test suite to cover not-coverted changed policy behaviors.

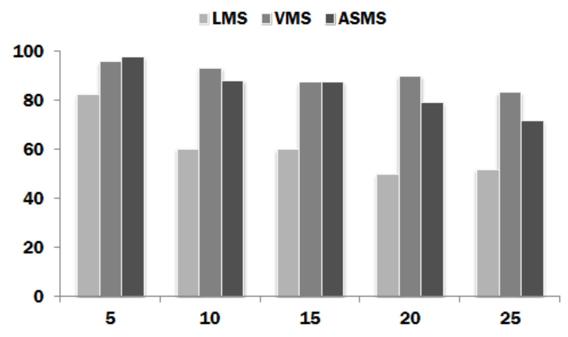

Figure 3. Regression test selection results (test reduction) for our subjects and each modified version. Y axis denotes the percentage of test reduction. X axis denote the number of policy changes in modified policy of our subjects.

C. Results

To answer $RQ1$, we measure how many test cases are reduced for regression testing. The goal of this research question is to show the reduction in the number of test cases that are selected using our techniques. The reduction in the number of test cases is the same across our three techniques. Figure 3 shows the results of test reduction for the three subjects. We observe that 82%∼97% of test reduction for a modified version with 5 policy changes. We observe that 51%∼83% of test reduction for a modified version with 25 policy changes. Such test reduction may reduce a significant cost in terms of test execution time for regression testing. We found that all of our test techniques show the same set of selected test cases to show that all of our techniques can detect every test case impacted policy changes.

To answer $RQ2$, we measure how many test cases are reduced for regression testing. The goal of this research question is to compare performance of our three test selection techniques. Table IV shows the results, which is elapsed time for the three subjects and each technique. For $Mut\text{-}Selection$ and $Cov\text{-}Selection$, the table shows the elapsed time of test-rule correlation (“Test-Rule”), change-impact-analysis (“CIA”), and test selection (“Test Selection”), respectively. For $Req\text{-}Selection$, the table shows the elapsed time of request recording (“Req-Collection”), and test selection (“Test Selection”). Note that “Test-Rule” and “Req-Collection” are preliminary step that could be done before test selection. We observe that $Cov\text{-}Selection$ (11,714 milliseconds on average) a significant elapsed time compared to $Mut\text{-}Selection$ (on average 69505 milliseconds) in terms of test-rule correlation. Moreover, we observe that a total elapsed time of $Req\text{-}Selection$ is 43 and 8 times faster than those of $Mut\text{-}Selection$ and $Cov\text{-}Selection$, respectively.

Table III shows the coverage results of selected test cases for the three subjects. For each subject, the table shows the number of changed policy behaviors (“# CT”), test cases that were selected (“# Cov”), and the percentage of test cases that were selected (“% Cov”) for each version of its policy. We denote our modified version of a policy as “Regression - N” where N denotes the number of changes injected into the policy. “# CT” is equal or less than N because our injected changes may not introduce policy behavior changes at the final modified version of the policy. For example, if one changes one rule with CRE and again changes the rule with CRE. As the results, the rule does not impact on policy changes.

Table V
TEST CASE TYPE RATIOS OF AUGMENTED TEST CASES FOR OUR SUBJECTS

Subjects	% M-Sub	% M-Cond	% M-SubCond	% Others
LMS	21.83	4.93	11.27	61.97
VMS	3.41	0.38	1.89	94.32
ASMS	16.30	9.63	7.41	66.67
Average	13.85	4.98	6.86	74.32

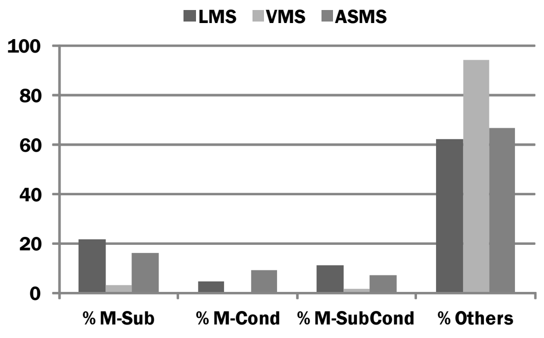

Figure 4. Augmented test results for our subjects and each modified version. Y axis denotes the percentage of augmented test cases. X axis denote the types of the augmented test cases.

To answer $RQ3$, we first identify changed policy behaviors (i.e., rules impacted by policy changes), which are not-covered with existing test cases. Note that existing test cases cannot reveal faults introduced by such policy behaviors. In Table III, we observe that our selected test cases cover 31.71∼36.23% of changed policy behaviors. The result show that about 70% of changed policy behaviors (i.e., rules impacted by policy changes) are not-covered. We apply our test augmentation technique to add new test cases to cover such not-covered-rules.

Table V shows the results of test augmentation for the three subjects. When developers know which policy behaviors to cover, the developers could generate test cases with requests to cover the policy behaviors.

For each subject, the table shows the percentage of generated test cases for each argumentation type type defined in Section III-D. While we generate/modify test cases to cover 100% percentage of not-covered policy behaviors, we may augment test cases based on following different types of modification/generation. "M-Sub" refers to generate test cases by modifying subject elements in test cases. For example, Given test cases, by changing subject elements (e.g, BORROWER instead of SECRETARY), the test cases cover the changed behaviors, which was not-covered in existing test cases shown in Table V. Similarly, "M-Cond" refers to generate test cases by modifying condition elements in test cases. When two of the preceding modification could not find test cases to modify, our technique finds "M-SubCond", which refers to generate test cases by modifying subject elements and condition elements together. At the last column, "Others" denotes an argumentation type, which may at least change action and resource elements.

In Table V, we observe that a total of 26% of the augmented test cases are modified by existing test cases in columns 2,3, and 4. However, for more than 74% of the augmented test cases are generated by scratch since we could not find similar test cases. Figure 4 illustrated the same data shown in Table V.

By comparing these results with those in Table III, we observe that there is indeed a correlation between coverage and test augmentation. The higher percentage of rule coverage for each subject, the higher percentage of tests are augmented by the first three types with modifying subject or condition elements. The reason is that, if the subject has test cases with high rule coverage, it is likely to find similar test cases to modify than subjects with test cases with low rule coverage.

D. *Threats to Validity*

The threats to external validity primarily include the degree to which the subject programs, the policies and regression model are representative of true practice. These threats could be reduced by further experimentation on a wider type of policy-based software systems and larger number of policies. In particular, our proposed regression model based on `RMR`, `RA`, and `CRE` modifies policy elements such as subject, resource, action, and condition attributes. Additional regression model to modify various policy attributes could simulate various policy modification. The threats to internal validity are instrumentation effects that can bias our results such as faults in Sun's PDP, faults in Margrave, and faults in our implementation.

VI. DISCUSSION

We believe that our approach can be applied to select test cases in programs each interacting with policies written in languages (such as EPAL [1] and Alloy [9]) other than XACML. For policies written in other languages, we could evaluate test cases or requests issued from test cases to set up test-rule correlation or capture requests. As our approach requires change impact analysis (against a policy and its modified policy) provided by the existing policy change impact tool to adopt XACML policies, we could convert the policies to XACML policies while maintain semantics of policies for change impact analysis.

VII. RELATED WORK

Very few research works have been done in the area of regression testing for policy-based software applications.

In [12], Martin et al. have proposed a framework to detect policies faults by analyzing requests-responses pairs, their tool generates an appropriate set of requests to maximize fault capability coverage in the policy. The same authors have developed a tool "Cirg" presented in [10] that minimizes the test cases generated through the analysis of structural coverage of policies. In [11], the authors have proposed a set of mutation operators to test the policy implementation. The test is performed by the analysis of requests-responses whenever a rule in the policy is mutated. In [8], the authors have proposed a generic model-based conformance checking approach for access control policies written in XACML. While these different approaches focus on test request generation to test the policy implementation, in this paper, our technique focuses on regression testing at the implementation level whenever the access control mechanism evolves. In [13], Mouelhi et al. have used transformed functional tests to security tests to test the access control mechanims, they have used mutation testing to detect functional tests that can be used for security testing.

VIII. Conclusion

We make three key contributions in this paper. First, we proposed a test selection approach for access control policy evolution. To the best of our knowledge, our paper is the first one for automatic test-selection approach in context of policy evolution. We present three automatic test-selection techniques. Second, we presented a test augmentation technique to select test cases for modification to cover not-covered policy changes. Third, we conduct evaluation with three metrics to measure the effectiveness of our approach and the efficiency of our three test selection techniques. The evaluation results demonstrated that our approach is effective to select test cases for test reduction. Among the proposed three test-selection techniques, the evaluation results demonstrated that the technique based recorded request evaluation is the most efficient compared with other two techniques. Besides, we also achieve 100% coverage of changed policy behaviors with augmented test cases.

References

[1] IBM, Enterprise Privacy Authorization Language (EPAL), Version 1.2 . http://www.w3.org/Submission/2003/SUBM-EPAL- 20031110, 2003.

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

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

[4] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *Proc. 4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 203)*, June 2003.

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

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

[7] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. 27th International Conference on Software Engineering (ICSE)*, pages 196–205, 2005.

[8] Vincent C. Hu, Evan Martin, JeeHyun Hwang, and Tao Xie. Conformance checking of access control policies specified in XACML. In *Proc. 1st IEEE International Workshop on Security in Software Engineering (IWSSE)*, 2007.

[9] Daniel Jackson, Ilya Shlyakhter, and Manu 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)*, pages 62–73, 2001.

[10] Evan Martin and Tao Xie. Automated test generation for access control policies via change-impact analysis. In *Proc. 3rd International Workshop on Software Engineering for Secure Systems (SESS)*, pages 5–11, 2007.

[11] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proc. 16th International Conference on World Wide Web (WWW)*, pages 667–676, 2007.

[12] Evan Martin, Tao Xie, and Ting Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. 8th International Conference on Information and Communications Security (ICICS)*, pages 139–158, 2006.

[13] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. Transforming and selecting functional test cases for security policy testing. In *Proc. 2nd International Conference on Software Testing, Verification, and Validation (ICST 2009)*, 2009.

[14] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[15] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22:529–551, August 1996.