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

JeeHyun Hwang[1] Tao Xie[1] Donia El Kateb[2] Tejeddine Mouelhi[2] Yves Le Traon[2]
[1]Department of Computer Science, North Carolina State University, Raleigh, USA
[2]Laboratory of Advanced Software SYstems (LASSY), University of Luxembourg, Luxembourg
[3]Security, Reliability and Trust Interdisciplinary Research Center, SnT, University of Luxembourg
jhwang4@ncsu.edu xie@csc.ncsu.edu {donia.elkateb,tejeddine.mouelhi,yves.letraon}@uni.lu

***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. In order to reduce the cost of regression testing, we develop a safe regression-test-selection approach, which selects every test case that may reveal a fault in program code impacted by policy changes. Our approach develops 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. We develop a test augmentation technique, which complements our test-selection techniques by generating additional test cases to cover not-covered-impacted-rules by existing test cases. We evaluate our approach on three real world Java programs interacting with policies. Our evaluation results show that our test selection techniques find XX test cases from an existing test cases, which covers only XX% of impacted rules on average. Our evaluation results show that our test augmentation technique generate 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 legitimate users to access or share critical information. Access control mechanisms are often governed by access control policies, each of which typically includes a set of rules to control which subjects can be permitted or denied to access which resources in which conditions. To facilitate specify and maintain policies in practice, system developers specify and enforce policies independently from actual functionality (i.e., business logic) of a system.

More specifically, program code, which represents the functionality, interacts with a specified policy through a security component, called policy decision point (PDP). Consider that the program code consists of methods Ms, which require decisions (e.g., permit or deny) to determine whether a given subject can have access on critical information. Typically, Ms formulate an access request that specifies that a subject would like to have access on critical information. The Ms 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 Ms to proceed.

In this paper, we focus on the regression testing problem in the context of policy evolution. The typical regression testing for program code interacting with a policy is as follows. Given a program code C 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" portion DP 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 DP of P'.

As security requirements of software often change during development and maintenance, system developers may modify policies according to the requirements. 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 order to increase confidence that the modification of policies is correct and does not introduce unexpected behavior, developers periodically conduct regression testing. 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. Consider that the developers add a permit rule D_1 into an existing policy without any changes to the original program code. Developers validate by generating executing test cases for methods impacted by this single policy change.

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 safe regression-test-selection approach, which selects every test case that may reveal a fault 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 behavior refers that, given a request, its evaluated decisions (for P and P', respectively) are different. Our approach detects policy rules, which map such requests. These rules are dangerous portions to be validated. In our approach, we have developed three automated tools for three test-selection 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 related to identified rules 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 of rules Rs of the policy impacted by the changes. We propose a test augmentation technique, which complements our test-selection techniques by generating additional test cases to cover not-covered-impacted-rules by existing test cases.

This paper makes three main contributions:

- We develop three safe test selection techniques to select only and every test cases from an existing test cases to test program code impacted by policy changes. **Is there any section that discusses the safety of the three algorithms: I am not sure that the three techniques are safe**
- We develop a test augmentation technique to generate additional new test cases to cover not-covered-impacted-rules with existing test cases selected by the preceding technique.
- TBD: Implementation and cost and benefit comparison
- **TBD: compare cost and benefits** We evaluate our approach on three real world Java programs interacting with policies. Our evaluation results show that our test selection techniques find XX test cases from an existing test cases, which covers only XX% of impacted rules on average. Our evaluation results show that our test augmentation technique generate additional test cases to cover 100% of the impacted rules.

The rest of the paper is organized as follows.

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 what follows, we consider XACML polices written in XACML language.

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 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) [4], [5] and Organization-Based Access Control (OrBAC) [3]. 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

```
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

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 [11]. 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 [12] 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 [13], 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.

1) Regression testing: Test selection and augmentation: Regression test selection techniques involves two issues:

- The first issue is the regression test selection problem which aims at minimizing the number of tests that have to be considered Rothermel et al. [13] have provided in their work a survey on techniques that have been used in this field. The regression test selection technique is called safe when the selected tests among the initial test suite enable to detect 100% of the faults.
- The second aspect is test-suite augmentation which checks the adequacy of existing tests for the incoming changes and aims at adding new tests to the initial Test Suite to cover introduced modifications.

2) Regression testing in the context of software policy-based systems: In this work, we consider regression testing in the context of policy-based software systems and propose to verify that the changed parts of the policy have not impacted the access control mechanisms implemented in the system. Testing policy-based software system requires the deployment of test cases that trigger the rules in the policy. Whenever a policy P is updated to a policy P', the changed rules need to be tested to validate that they do not introduce a negative impact on the initial implementation of access control mechanisms. The figure **??** illustrates the overall scenario of regression testing applied to policy-based software systems
ADD schema

III. Approach

As manually selecting system tests for regression testing is tedious and error-prone, we have developed three techniques to automate system tests selection for regression in policy evolution. Among existing system tests, the objective is to select all system tests 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 system tests are taken an input; these tests invoke methods in program code. We analyze given program code and polices to select *only* system tests for regression testing in case of policy evolution. Among given system tests, our selected system tests 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 system tests to satisfy sufficiency as follows. We measure sufficiency of regressing testing with our selected system tests based on rule coverage criteria []. For not-covered changed policy behaviors (i.e., rules), our approach automatically generates system tests 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 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.

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

A. Test Selection via Mutation Analysis

Our first proposed test selection technique uses mutation analysis to select system tests as follows. The approach needs a preliminary step which is necessary to establish a rule-test correlation. 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 tests for this rule decision change. We execute system tests T on program code for P and Pr_i. To detect changed policy behaviors, the technique monitors responses of evaluated requests formulated from system tests T. The system tests, which evaluate different policy decisions against P and P', enable to map rule r_i to system Tests $t \in T$. The preliminary step ends by establishing a correlation between each rule in P and corresponding tests $t \in T$ that trigger this rule.

```
<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)

The selection of system tests 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 phase to select the subset of system tests which are correlated with changed rules.

While the technique can quickly select system tests, the technique requires rule-test setup (in the preliminary step), which could be costly in terms of execution times. 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 via System Test Execution

Our previous technique finds correlation of all of existing rules N in a given policy with the system Tests. To reduce such correlation setup efforts, we develop a technique to correlate only rules, which can be evaluated by system tests. Our intuition is that system tests may interact only with a small number of rules in a policy instead of a whole rules in the policy. Therefore, we start by establishing a correlation between system tests and triggered rules. Thus we execute

Algorithm 1: Test Selection via Mutation Analysis Algorithm

TestSelection1(P, P_m, T): T'
Input: XACML Policy P, modified policy P_m, Initial Sytem 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'

system tests T on program code that interacts with P. Our technique monitors which rules in a policy are evaluated with requests formulated from system tests T. Once the mapping test-rule is established, we proceed like the first approach by conducting change impact analysis of P and P_m to find which rules' decision are changed. Mapped tests to those rules constitutes the subset of regression test selection.

An important benefit of this technique is to reduce cost in terms of mutation analysis and execution times. 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 selects in the second step, the technique requires rule-test setup (in the preliminary step), which could be costly in terms of execution time. Consider that requests Rs are formulated from system tests 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.

Algorithm 2: Test Selection via System Test Execution

TestSelection2(P, P_m, T): T'
Input: XACML Policy P, modified policy P_m, Initial Sytem 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_1=Map(t,$\{r_p\}_{i=1..p}$) where $\{r_p\}_{i=1..p}$ are the rules in P that are triggered by t
end for
Generate a mapping rule/test from MAP_1
MAP_2=Map(r_i,T_{r_i}) $\leftarrow MAP_1$
/*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'

C. Test Selection via Play Back

To reduce such correlation setup efforts in the previous techniques, we develop a technique, which does not require a correlation setup. Our approach executes system tests T on program code for P and records all requests issued to policy decision point (PDP) for each system 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.

While our previous two techniques require correlation rule-test setup. The current approach requires the execution of system test cases 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

TBD

IV. Implementation

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. We have implemented test selection techniques and mutant generation in Java. We generate mutants by changing rules in access control policies To simulate regression in access control policies. We used three types

Algorithm 3: Test Selection via Play Back

TestSelection3(P, P_m, T): T'
Input: XACML Policy P, modified policy P_m, Initial Sytem 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'

of mutants injected in the policies; the first one is RMR (Rule Removal), RA (Rule Addition), and CRE (Change Rule Effect) mutants. [ToDo: citation and explain more] In our mutants, we only change rules in access control policies, we do not simulate regression in test code in subjects. 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. We compare also selected tests by our three techniques to compare that all of these techniques return the same tests for regression. The objective of this evaluation is to investigate how our approach impacts performance for subjects and safety to select all tests for regression.

A. Subjects

The subjects include three real-life Java programs each which interact with access control policies [**?**]. 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.

Table I shows information in our subjects. [ToDo: explain more] Table II shows information in our subjects. [ToDo: explain more] Our subjects are equipped with Sun PDP [**?**], which is a popularly used PDP to evaluate requests. Policies in LMS,VMS, and ASMS contain a total of 720, 945, and 1760 rules, respectively.

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 techniques? This question helps to show that our techniques can reuse ZZ% of a test suite. We also compare our approach with random test select technique to show how effectively our approach selects test suite to cover changed policy behaviors.
- RQ2: Do our protest selections are safe? This question helps to show that our techniques select all tests, which shows only and all test cases to reveal changed behaviors in access control policies.
- RQ3: 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.
- RQ4: 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 beahviors.

C. Metrics

We use following 4 metrics in our evaluation.

- Policy coverage information for changed policy behaviors.
- Number of test cases reused by the test selection technique.
- Elapsed time of test-rule correlation, change impact analysis, and test selection.
- Number of test cases generated by the test augmentation technique.

[ToDo: explain more]

VI. Related Work

Our previous work developed policy testing approaches for policy structural coverage [10], request generation [8], and mutation testing [9]. Our previous work [6] also proposed a generic model-based conformance checking approach for access control policies written in XACML. These pieces of work do not rely on properties for generating test requests to detect a fault in a policy. Our previous work [7] developed an approach for measuring the quality of policy properties in policy verification. Given user user-specified

Table I
SUBJECTS

	LOC	# of Test Methods	# of Security Test Methods	# of Covered Rules	# of Not-covered Rules	% of Covered Rules
LMS	3749	46	29	42	0	100
VMS	3734	52	10	13	106	12
ASMS	7836	93	91	109	21	83

Table II
ACCESS CONTROL POLICIES IN OUR SUBJECTS

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

properties, the quality of properties are measured based on fault-detection capability. While these approach focus on test request generation, in this paper, our technique targets at test selection (among existing system tests) for policy evolution.

Related Work here!!

VII. CONCLUSION

Conclusion here!!

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] 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.

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

[5] 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.

[6] 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.

[7] Evan Martin, JeeHyun Hwang, Tao Xie, and Vincent Hu. Assessing quality of policy properties in verification of access control policies. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, pages 163–172, 2008.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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