Selection of Regression System Tests for Security Policy Evolution

JeeHyun Hwang[1] Tao Xie[1] Donia El Kateb[2] Tejeddine Mouelhi[3] Yves Le Traon[3]

[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, developers may modify security policies such as access control policies (policies in short) according to evolving requirements. To increase confidence that the modification of policies is correct, developers conduct regression testing. However, rerunning all of existing system test cases could be costly and time-consuming. To address this issue, we develop a regression-test-selection approach, which selects every system test case that may reveal regression faults caused by policy changes. Our evaluation results show that our test-selection techniques reduce a significant number of system test cases efficiently.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Access Control Policy; Regression Testing; Test Selection

1. INTRODUCTION

Access control is one of the privacy and security mechanisms for granting only legitimate users with access to critical information. Access control is governed by access control policies (policies in short), each of which includes a sequence of rules to specify which subjects are permitted or denied to access which resources under which conditions. To facilitate specifying policies, system developers often use policy specification languages such as XACML [1], which helps specify and enforce policies separated from actual functionality (i.e., business logic) of a system.

With the change of security requirements, developers may modify policies to comply with requirements. After the modification, it is important to validate and verify program code and policies together to determine that this modification is correct and does not introduce unexpected behaviors of a given system (i.e., regression faults). Consider that the system changes its original policy P with a modified policy P'. Due to this modification, the system may reveal different system behaviors, which are "dangerous" portions. Different system behaviors are caused by different policy behaviors (i.e., given a request, its evaluated decisions against P and P', respectively, are different). In order to validate the "dangerous" portions efficiently, the developers select and execute only test cases (from existing test cases) that exercise the dangerous portions.

To exercise the dangerous portions with existing test cases, a naive regression testing strategy is to rerun all existing non-redundant system test cases. However, rerunning these test cases could be costly and time-consuming, especially for large-scale systems. Instead of this strategy, developers often adopt test selection before execution of test cases. This regression-test selection selects and executes only test cases to expose different behaviors across different versions of the system. This regression-test selection may require substantial cost to select and execute such system test cases. In theory, if the cost of regression-test selection is smaller than rerunning all of initial system test cases, regression-test selection helps reduce overall cost in validating whether the modification is correct.

In addition to cost-effectiveness, safeness is important as well. An approach of safe regression-test selection selects every test case that may reveal a fault in a modified program [6]. In contrast, an unsafe approach of regression-test selection may omit test cases that reveal a fault in the modified program.

In this paper, we propose a safe approach of regression-test selection to select superset of the set of test cases (i.e., fault-revealing test cases) that reveal faults due to the policy modification. To the best of our knowledge, our paper is the first one for test selection in the context of policy evolution. Different from prior research work on test selection [2, 4, 6] that deals with changes in program code, our work deals with code-related components such as policies, which impact behaviors of the program code.

We have developed an automated approach for three regression-test selection techniques: the one based mutation analysis, the one based on coverage analysis, and the most efficient one based on recorded request evaluation. The first two techniques are based on correlation between test cases and rules R_{imp} impacted by policy changes. R_{imp} are rules revealing different policy behaviors due to policy changes.

More specifically, given a rule r_i in P, the first technique first creates its mutant $M(r_i)$ by changing decision of the rule. On executing test cases on program code interacting with P and $M(r_i)$, respectively, this technique collects test cases that reveal different

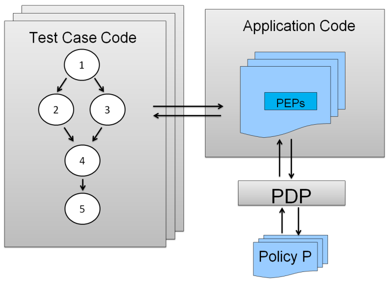

Figure 1: XACML policy evaluation process in a policy-based software system

```
 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>
...
35 </policy>
```

Figure 2: An example policy specified in XACML

policy behaviors. Our rationale is that, if a test case is correlated with r_i, the test case should reveal different policy behaviors induced by policy changes. However, this technique is costly because this technique requires at least $2{\times}n$ execution of test cases to find all correlation between test cases and rules where n is the number of rules in P.

The second technique uses coverage analysis to find correlation. While executing test cases, this technique correlate rules, which can be evaluated (i.e., covered) by test cases. Compared with the first technique, this technique significantly reduces cost during correlation process because it requires test case execution at once.

The third one first captures requests issued from program code while executing test cases. This technique evaluates these requests against P and P', respectively. Among the requests, This technique selects only requests R_o that reveal different policy behaviors and find R_o's corresponding test cases.

2. BACKGROUND

Our approach is based on policy-based software systems regulated by policies specified in XACML [1]. XACML has become the de facto standard for specifying policies. Typically, XACML policies are specified separately from actual functionality (i.e., business logic) in program code. Figure 1 illustrates XACML policy evaluation process. At an abstract level, program code interacts with policies as follows. Program code includes security checks, called Policy Enforcement Points (PEPs), to check whether a given subject can have access on protected information. The PEPs formulate and send an access request to a security component, called Policy Decision Point (PDP) loaded with policies. The PDP next evaluates the request against the policies and determines whether the request should be permitted or denied. Finally, the PDP sends the decision back to the PEPs to proceed.

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.

```
 1 <Policy PolicyId="Library Policy" RuleCombAlgId="Permit-overrides">
 2  <Target/>
 3    <Rule RuleId="1" Effect="Deny">
 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>
...
35 </policy>
```

Figure 3: An example mutant policy by changing $R1$'s rule decision (i.e., effect)

More than one rule in a policy may be applicable to a given request. The *combining algorithm* is used to combine multiple decisions into a single decision. There are four standard combining algorithms. 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 2 shows a policy specified in XACML. A policy may have more than one XACML rule. Due to space limitation, we describe only one rule in the example policy. 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`.

3. APPROACH

As manual selection of test cases for regression testing is tedious and error-prone, we have developed three techniques to automate selection of test cases for security policy evolution. Consider that a program code interacts with a PDP loaded with a policy P. Let P' denote P's modified policy. For regression-test selection, our gale is to select $T' \subseteq T$ where T is an existing test suite and T' reveals different policy behaviors for P and P'.

3.1 Test Selection based on Mutation Analysis

Our first technique first establishes correlation (i.e., rule-test correlation) between rules and test cases based on mutation analysis before regression-test selection.

Correlation between rules and test cases. This step establishes correlation that which rule in P are related with test cases $t \in T$. For each rule r_i in P, we create its rule-decision-change (RDC) mutant $M(r_i)$ by changing decision (e.g., Permit to Deny) of r_i. Figure 3 illustrates an example mutant by changing decision of the first rule in Figure 2. The technique next executes T on program for P and $M(r_i)$, respectively, and monitors evaluated decisions. If the two decisions are different while executing $t \in T$, the technique establishes correlation between r_i and t.

Regression-Test selection. This step first analyzes syntactic difference between P and P' (i.e., which rules are changed due to the policy changes). Once these rules (which are reflected by syntactic difference) are identified, our technique selects the subset of test cases which are correlated with the changed rules.

The drawback is that this technique requires correlation step, which could be costly in terms of execution time. This technique executes T for $2{\times}n$ times where n is the number of rules in P. Moreover, if a policy is modified, correlation step should be done again for changed rules. As this regression-test selection is based on syntactic differences, this technique may select rules that may not result in actual policy behavior changes (i.e., semantic policy changes) due to various reasons such as a newly added rule is overwritten by existing rules.

3.2 Test Selection based on Coverage Analysis

To reduce cost of correlation step in the preceding technique, our second technique correlate only rules, which can be evaluated (i.e., covered) by test cases.

Correlation between rules and test cases. Our technique executes test cases T on program that interacts with P and monitors which rule r_i is evaluated with requests issued from test cases $t \in T$. Our technique establishes correlation between r_i and $t \in T$.

Regression-Test selection. We use the same regression-test selection step in the preceding technique

An important benefit of this technique is to reduce cost in terms of execution time. This technique requires executing T only once. Similar to the preceding technique, this technique find the changed rules based on syntactic differences between P and P', which may not result in actual policy behavior changes as well.

3.3 Test Selection based on Recorded Request Evaluation

To reduce such correlation efforts in the preceding techniques, we develop a technique, which does not require correlation between test cases and rules. The third technique executes test cases T on a program for P. Our technique captures and records requests issued from test cases T. For test selection, our technique evaluates all recorded requests against P and P'. Our technique selects test cases $t \in T$ that issue requests that engender different decisions for P and P'.

This technique requires the execution of T only once. Moreover, different from the two preceding techniques, this technique does not require the availability of policies in practice. It is useful especially when polices are not available, but only evaluated decisions are available. As different decisions are reflected by actual policy behaviors changes (i.e, semantic changes) between P and P_m, this technique can select only test cases precisely.

3.4 Safe Test-Selection Techniques

4. EXPERIMENTS

We conducted to evaluate our proposed techniques of regression-test selection. We carried out our evaluation on a PC, running Windows 7 with Intel Core i5, 2410 Mhz processor, and 4 GB of RAM. We collected three Java programs [5] each interacting with policies written in XACML. Library Management System (LMS) provides web services to borrow/return/manage books in a library. Virtual Meeting System (VMS) provides web conference services to organize online meetings. Auction Sale Management System (ASMS) provides web services to manage online auction. These three subjects include 29, 10, and 91 security test cases, which target at testing of security checks and policies. The test cases cover 100%, 12%, and 83% of 42, 106, and 129 rules in policies in LMS,VMS, ASMS, respectively.

Instrumentation. We implemented regression simulator, which injects any number of policy changes based on predefined three regression types. RMR (Rule Removal) removes a randomly selected rule. RDC (Rule Decision Change) changes the decision of a randomly selected rule. RA (Rule Addition) adds a new rule consisting of random attributes collected from P in a random place. Combination of the three regression types can incur any policy changes.

For experiments, the regression simulator injects 5, 10, 15, 20, and 25 policy changes, respectively. Our experiments are repeated 12 times to avoid the impact of randomness of policy changes. We measure effectiveness and efficiency of our three techniques by measuring the number of selected test cases and elapsed time during test selection process, respectively.

Research questions. We intend to address the following research questions:

- RQ1: How high percentage of test cases (from an existing test suite) is reduced by our test-selection techniques? This question helps to show that our techniques can reduce the cost of regression testing.
- RQ2: How many selected test cases can reveal regression faults? This question helps to show that our techniques can detect regression faults.
- RQ3: How much time do our techniques take to conduct test selection by given subjects? This question helps to compare performance of our techniques by measuring their efficiency.

Results. To answer $RQ1$, we measure test reduction percentage ($\%TR$), which is the number of selected test cases divided by the number of existing security test cases. Table 1 shows the number of selected test cases on average for each technique.. "Regression - m" denotes a group of modified policies where m denotes the number of policy changes on P. "$\#S_M$", "$\#S_C$", and "$\#S_R$" denote the number of selected test cases on average by our three techniques, test selection based on mutation analysis (TS_M), test selection based on coverage analysis (TS_C), and test selection based on recorded request evaluation (TS_R), respectively. We observe that TS_R selected less number of test cases than other two techniques. The reason is because, while TS_M and TS_C select test cases impacted by syntactic policy changes, TS_R select test cases impacted by semantic policy changes. As illustrated in Section 3, syntactic policy changes do not always incur semantic policy changes (e.g., a newly added rule is overwritten by existing rules).

Figure 4 shows the results of test reduction percentage for the policies modified from our three subjects. LMS1 (LMS2), VMS1 (VMS2), and ASMS1 (ASMS2) show test reduction percentages for modified policies on our three subjects, respectively, using TS$_M$ and TS$_C$ (TS$_R$). We observe that our techniques achieve 42%$\sim$97% of test reduction for a modified policy with 5$\sim$25 policy changes. Such test reduction may reduce a significant cost in terms of test execution time for regression testing.

To answer $RQ2$, we show that our selected test cases detect regression faults. Detection of regression faults is dependent on the quality of test oracles in test cases. We observe that test cases is our subjects include strong test oracles. Given requests rs issued from test cases, the test oracles checks that rs's evaluated decisions are consistent with expected evaluated decisions. Due to the strong test oracles, when policy changes introduce actual different policy behaviors (reflected by semantic policy changes), our selected test cases by TS$_R$ fail for testing. For our subjects, all of test cases selected by TS$_R$ detect regression faults.

To answer $RQ3$, we measure elapsed time. The goal of this research question is to compare efficiency of our three test-selection techniques. Table 2 shows the evaluation results for the three subjects and each technique. For TS$_M$ and TS$_C$, the result shows the elapsed time of correlation ("Cor") and test selection ("Se;"), respectively. For TS$_R$, the table shows the elapsed time of request recording ("Col"), and test selection ("Sel"). We observe that correlation e_1 of TS$_C$ takes significantly more time than that e_2 of TS$_M$. e_1 and e_2 take 11,714 milliseconds and 69505 milliseconds on average, respectively. The reason is because TS$_C$ executes the

Table 1: The number of selected test cases on average for each policy group by each technique

Subject	Regression - 5			Regression - 10			Regression - 15			Regression - 20			Regression - 25		
	#S$_M$	#S$_C$	#S$_R$	#S$_M$	#S$_C$	#S$_R$	#S$_M$	#S$_C$	#S$_R$	#S$_M$	#S$_C$	#S$_R$	#S$_M$	#S$_C$	#S$_R$
LMS	4.7	4.7	4.5	11.0	11.0	9.5	12.9	12.9	10.2	14.8	14.8	13.8	16.8	16.8	14.6
VMS	0.1	0.1	0.1	0.4	0.4	0.2	1.2	1.2	0.8	1.6	1.6	1.2	1.8	1.8	1.1
ASMS	6.6	6.6	5.9	10.9	10.9	10.0	16.4	16.4	14.8	21.3	21.3	19.3	22.4	22.4	17.2
Average	3.8	3.8	3.5	7.4	7.4	6.6	10.2	10.2	8.6	12.6	12.6	11.4	13.7	13.7	11.0

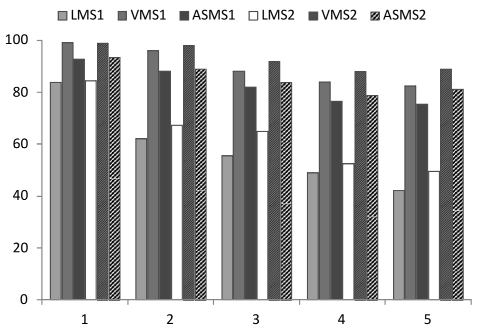

Figure 4: LMS1 (LMS2), VMS1 (VMS2), and ASMS1 (ASMS2) shows test reduction percentages for modified policies on our three subjects, respectively, using TS$_M$ and TS$_C$ (TS$_R$). Y axis denotes the percentage of test reduction. X axis denotes the number of policy changes on our subjects.

Table 2: Elapsed time (millisecond) for each test-selection technique, and each policy

Subject	TS_M		TS_C		TS_R	
	Cor	Sel	Cor	Sel	Col	Sel
LMS	70,496	4	5,214	4	2,096	2
VMS	19,771	1	7,506	1	1,873	2
ASMS	118,248	11	22,423	11	1,064	21
Average	69,505	5	11,714	5	1,678	8

existing test cases only once but TR_2 executes the existing test cases for $2\times n$ times where n is the number of rules in a policy under test. For total elapsed time for each technique, we observe that the total elapsed time of TS$_R$ is 43 and 8 times faster than those of TS$_M$ and TS$_C$, respectively.

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. The threats to internal validity are instrumentation effects that can bias our results such as faults in PDP, and faults in our implementation.

5. RELATED WORK

Various techniques have been proposed on regression testing of software programs in software engineering and programming language communities [2,4,6,7]. These techniques aim to select every test case to reveal different behaviors correctly after modification in program or augment test cases. These techniques are related to regression test selection [6], [4], test-suite prioritization [2], and test-suite augmentation [7]. Note that these techniques focus on changes at code level. None of these techniques consider potential changes that can arise from code-related components (such as a policy that interacts with application code). Polices and general programs are fundamentally different in terms of structures, semantics, and functionalities, etc. Therefore, techniques for regression testing of general programs are not suitable for addressing the test selection problem for policy evolution. Our work is the first one for automatic test-selection approach in the context of policy evolution.

Another work closest to ours is Fisler et al.'s work [3]. They developed a tool called "Margrave" that enables verifying properties against policies written in XACML and conducts change impact analysis. Margrave supports for detecting changed policy behaviors between two XACML policies. However, Margrave supports for limited functionality of XACML and does not handle with situations where program behaviors are changed by policy changes as our work focuses on.

6. CONCLUSION

We believe that our approach could be practical and effective to select test cases for policy-based software systems interacting not only XACML policies but also policies specified by other policy specification languages (e.g., EPAL). We make two key contributions in this paper. First, we proposed three test selection techniques for access control policy evolution. To the best of our knowledge, our paper is the first one for automatic test-selection approach in the context of policy evolution. Second we conduct an evaluation 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 and efficient to select test cases for policy evolution.

7. REFERENCES

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

[2] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proc. 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 102–112, 2000.

[3] 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)*, pages 196–205, 2005.

[4] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, pages 184–208, 2001.

[5] T. Mouelhi, Y. Le Traon, and B. 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.

[6] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22:529–551, August 1996.

[7] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 218–227, 2008.