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, 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 approach reduces a substantial number of system test cases efficiently.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Security, Reliability

Keywords

Security 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 security policies such as 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 separately from actual functionality (i.e., business logic) of a system.

With the change of security requirements, developers may modify policies to comply with the requirements. After the modification, it is important to validate and verify the given system to determine that this modification is correct and does not introduce unexpected behaviors (i.e., regression faults). Consider that the system's original policy P is replaced with a modified policy P'. The system may exhibit different system behaviors affected by different policy behaviors (i.e., given a request, its evaluated decisions against P and P', respectively, are different) caused by the policy changes. Such different system behaviors are "dangerous" portions where regression faults could be exposed.

In order to validate the dangerous portions with existing test cases, a naive strategy of regression testing is to rerun all existing system test cases. However, rerunning these test cases could be costly and time-consuming, especially for large-scale systems. Instead of this strategy, developers use regression-test selection before execution of test cases. This regression-test selection selects and executes only test cases that may 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. If the cost of regression-test selection and selected-test execution is smaller than rerunning all of the initial system test cases, regression-test selection helps reduce overall cost in validating whether the modification is correct.

In addition to cost effectiveness, safety is an important aspect in regression-test selection. A safe approach of regression-test selection selects every test case that may reveal a fault in a modified program [7]. 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 a superset of fault-revealing test cases, i.e., 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, 7] that deals with changes in program code, our work deals with code-related components such as policies, which impact system behaviors.

Our approach includes three regression-test selection techniques: the first one based mutation analysis, the second one based on coverage analysis, and the third one based on recorded request evaluation. The first two techniques are based on correlation between test cases and rules R_{imp} where R_{imp} are rules being involved with syntactic changes across policy versions. The first technique selects a rule r_i in P and creates P's mutant $M(r_i)$ by changing r_i's decision. This technique selects test cases that reveal different policy behaviors by executing test cases on program code interacting with P and $M(r_i)$, respectively. Our rationale is that, if a test case is correlated with r_i, the test case may reveal different system behaviors affected by modification of r_i in P. However, this technique is costly because it requires at least $2 \times n$ executions of each test case to find all correlations between test cases and rules where n is the number of rules in P.

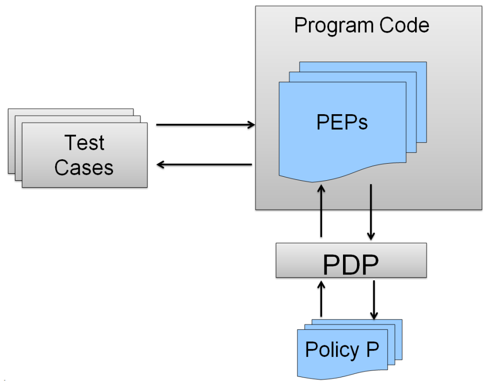

Figure 1: Evaluation process of XACML policies in a policy-based software system.

```
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.

The second technique uses coverage analysis to establish correlations between test cases and rules by monitoring which rules are evaluated (i.e., covered) for requests issued from program code. Compared with the first technique, this technique substantially reduces cost during the correlation process because it requires execution of each test case 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. This technique then selects only test cases that issue requests evaluated to different decisions.

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 evaluation process of XACML policies. 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 to 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 further 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 given system.

More than one rule in a policy may be applicable to a given request. A *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 what 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.

```
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 the first rule's decision (i.e., effect).

Figure 2 shows an example policy specified in XACML. Due to space limit, we describe only one rule in the policy in a simplified XACML format. 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 program code interacts with a PDP loaded with a policy P. Let P' denote P's modified policy. Let S_P denote program code interacting with P. For regression-test selection, our goal is to select $T' \subseteq T$ where T is an existing test suite and T' reveals different system behaviors due to the modification between P and P'.

3.1 Test Selection based on Mutation Analysis

Our first technique first establishes correlation between rules and test cases based on mutation analysis before regression-test selection.

Correlation between rules and test cases. For rule r_i in P, we create P's rule-decision-change (RDC) mutant $M(r_i)$ by changing r_i's decision (e.g., Permit to Deny). Figure 3 illustrates an example mutant by changing the decision of the first rule in Figure 2. The technique next executes T on S_P and $S_{M(r_i)}$, respectively, and monitors evaluated decisions. If the two decisions are different for $t \in T$, the technique establishes correlation between r_i and t.

Regression-test selection. This step selects test cases correlated with rules that are involved with syntactic changes between P and P'. In particular, this technique analyzes syntactic difference, `SDiff`, between P and P' (e.g., a rule's decisions or locations are changed) and identifies rules that are involved in the syntactic difference.

The drawback of this technique is that it requires the 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 the policy is modified, the correlation step should be done again for the changed rules. As this regression-test selection is based on `SDiff`, this technique may select rules that may not be involved with actual policy behavior changes (i.e., semantic policy changes).

3.2 Test Selection based on Coverage Analysis

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

Correlation between rules and test cases. Our technique executes test cases T on S_P and monitors which rules are evaluated for requests issued from the execution of test case $t \in T$. Our technique establishes correlation between a rule r_i and $t_i \in T$ if and only if r_i is evaluated for requests issued from PEPs while executing t_i.

Regression-test selection. We use the same selection step in the preceding technique. An important benefit of this technique is to reduce cost in terms of execution of test cases. This technique requires executing T only once. Similar to the preceding technique, this technique finds the modified rules based on `SDiff` between P and P', which may not be involved with actual policy behavior changes.

3.3 Test Selection based on Recorded Request Evaluation

To reduce correlation cost in the preceding techniques, we develop a technique that does not require correlation between test cases and rules. The third technique executes T on S_P. The technique captures and records requests $R_r s$ issued from PEPs while executing T on S_P. For test selection, our technique evaluates $R_r s$ against P and P'. Our technique selects test case $t \in T$ that issues requests engendering different decisions for P and P'.

This technique requires the execution of T only once. Moreover, this technique is useful especially when polices are not available, but only evaluated decisions are available. As different decisions are reflected by actual policy behavior changes (i.e., semantic changes) between P and P', this technique can select fault-revealing test cases more effectively.

3.4 Safe Test-Selection Techniques

A test-selection algorithm is `safe` if the algorithm includes the set of every fault-revealing test case that would reveal faults in a modified version. In our work, the first test-selection technique is `safe` when a policy uses the *first-applicable* algorithm. If the policy uses other combining algorithms, we use our previous approach [5] to convert the policy to its corresponding policy using the *first-applicable* algorithm. The second and third techniques are `safe` for any policies specified in XACML. Due to space limit, proof of safety of our three techniques is presented on our project website[1].

4. EXPERIMENTS

We conducted experiments for evaluating our proposed techniques of regression-test selection. We carried out our experiments on a PC, running Windows 7 with Intel Core i5, 2410 Mhz processor, and 4 GB of RAM. As experimental subjects, we collected three Java programs [6] each interacting with policies written in XACML. The Library Management System (`LMS`) provides web services to borrow/return/manage books in a library. The Virtual Meeting System (`VMS`) provides web conference services to organize online meetings. The 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 security checks and policies. The test cases cover 100%, 12%, and 83% of 42, 106, and 129 rules from policies in `LMS`, `VMS`, and `ASMS`, respectively.

Instrumentation. We implemented a regression simulator, which injects any number of policy changes based on three predefined 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 attributes randomly selected among attributes collected from P. Combination of the three regression types can incur various policy changes.

For our 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 test-reduction percentage, the number of fault-revealing test cases, and elapsed time.

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

- RQ1: How high percentage of test cases (from an existing test suite) are reduced by our test-selection techniques? This question helps show that our techniques can reduce the cost of regression testing.
- RQ2: How high percentage of selected test cases can reveal regression faults? This question helps show that our techniques can effectively select fault-revealing test cases.
- RQ3: How much time do our techniques take to conduct test selection? This question helps 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 is 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 test-selection techniques, one based on mutation analysis (TS_M), one based on coverage analysis (TS_C), and one based on recorded request evaluation (TS_R), respectively. We observe that TS_R selected a fewer number of test cases than the other two techniques. The reason is that, while TS_M and TS_C select test cases based on syntactic difference, TS_R selects test cases based on actual policy behavior changes (i.e., semantic policy changes). As illustrated in Section 3, syntactic difference may not result in actual policy behavior changes.

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

To answer $RQ2$, we show the percentage of selected test cases that reveal regression faults. Detection of regression faults is dependent on the quality of test oracles in test cases. The test cases for our three subjects include test oracles, which check correctness of decisions evaluated for all the requests issued from PEPs. Therefore, selected test cases by TS$_R$ would all detect regression faults (caused by semantic policy changes). On average, the percentages of selected test cases that reveal regression faults are 87%, 87%, and 100% for our three techniques TS$_M$, TS$_C$, and TS$_R$, respectively

To answer $RQ3$, we measure elapsed time of conducting test selection. The goal of this research question is to compare efficiency

[1]http://research.csc.ncsu.edu/ase/projects/regpolicy/

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

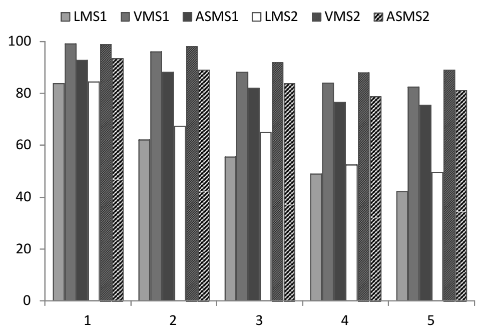

Figure 4: LMS1 (LMS2), VMS1 (VMS2), and ASMS1 (ASMS2) show test-reduction percentages for our subjects with modified policies, respectively, using TS$_M$ and TS$_C$ (TS$_R$). Y axis denotes the percentage of test reduction. X axis denotes the number of policy changes on our subjects.

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

of our three test-selection techniques. Table 2 shows the evaluation results. For TS$_M$ and TS$_C$, the results show the elapsed time of correlation ("Cor") and test selection ("Sel"), respectively. For TS$_R$, the results show the elapsed time of request recording ("Col") and test selection ("Sel"). We observe that correlation (11,714 milliseconds on average) of TS$_C$ takes substantially less time than correlation (69,505 milliseconds on average) of TS$_M$. The reason is that TS$_C$ executes the existing test cases only once but TS$_M$ 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 by each technique, we observe that the total elapsed time of TS$_R$ is 43 and 8 times faster than that 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 a larger number of policies. The threats to internal validity are instrumentation effects that can bias our results such as faults in the PDP, and faults in our implementation.

5. RELATED WORK

Various techniques have been proposed on regression testing of software programs [2, 4, 7]. These techniques aim to select test cases that could reveal different behaviors after modification in programs. These techniques are related to regression-test selection [7], [4], and test-suite prioritization [2]. 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 security policies specified separately). Polices and general programs are fundamentally different in terms of structures, semantics, and functionalities, etc. Therefore, techniques for regression testing of programs are not suitable for addressing the test-selection problem for policy evolution. Our work is the first automatic test-selection approach for policy evolution.

Another approach closest to ours is Fisler et al.'s approach [3]. They developed a tool called Margrave that enables conducting change-impact analysis between two XACML policies. We could use Margrave to identify semantic policy changes between two policies. However, Margrave supports for only limited functionality of XACML. Moreover, Margrave does not support test selection as our work does.

6. CONCLUSION

Our approach could be practical and effective to select test cases for policy-based software systems interacting not only with XACML policies but also with 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. To the best of our knowledge, our paper is the first one for automatic test selection in the context of policy evolution. Second, we conducted experiments to assess the effectiveness and efficiency of our three test-selection techniques.

Acknowledgments. This work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, ARO grant W911NF-08-1-0443, an NSA Science of Security Lablet grant, and a NIST grant.

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. 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] A. X. Liu, F. Chen, J. Hwang, and T. Xie. XEngine: A fast and scalable XACML policy evaluation engine. In *Proc. International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 265–276, 2008.
[6] 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)*, pages 171–180, 2009.
[7] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22:529–551, 1996.