Automated Test Generation for Access Control Policies via Change-Impact Analysis

Evan Martin
North Carolina State University
Raleigh, NC, USA
eemartin@ncsu.edu

Tao Xie
North Carolina State University
Raleigh, NC, USA
xie@csc.ncsu.edu

Abstract

Access control policies are increasingly written in specification languages such as XACML. To increase confidence in the correctness of specified policies, policy developers can conduct policy testing with some typical test inputs (in the form of requests) and check test outputs (in the form of responses) against expected ones. Unfortunately, manual test generation is tedious and manually generated tests are often not sufficient to exercise various policy behaviors. In this paper we present a novel framework and its supporting tool called Cirg that generates tests based on change-impact analysis. Our experimental results show that Cirg can effectively generate tests to achieve high structural coverage of policies and outperforms random test generation in terms of structural coverage and fault-detection capability.

1. Introduction

Access control mechanisms control which principals such as users and processes have access to which resources in a system. To facilitate managing access control, policy languages such as XACML [1] and Ponder [7] have been increasingly used to specify access control policies for a system. After policies are specified, a software component called a Policy Decision Point (PDP) evaluates a request against specified access control policies, and returns a response that either permits or denies the request based on the policies. Assuring the correctness of policy specifications is becoming an important and yet challenging task, especially as software systems become larger and more complex, and are deployed to manage a large amount of sensitive information and resources.

Software testing aims at efficiently detecting errors in software through dynamic execution. Errors in policy specifications may also be discovered by leveraging existing techniques for software testing and applying them to policy testing. In policy testing, test inputs are access requests and test outputs are access responses. The execution of test inputs occurs as requests are evaluated by the PDP against the access control policies under test. Policy authors can inspect request-response pairs to check whether they are expected. Access control policies are often tested with manually defined access requests so that policy authors may check the PDP's responses against expected ones. Because it is tedious for developers to manually generate test inputs for policies and manually generated tests are often not sufficient for achieving high policy structural coverage, in this paper we present a novel framework that automatically generates tests for access control policies by leveraging change-impact analysis. We have implemented the framework in a tool called Cirg (**C**hange-**I**mpact **R**equest **G**eneration) and evaluated it on policies collected from various sources, most of which are complex policies being used in real systems. Our experimental results show that Cirg can effectively generate tests to achieve higher policy structural coverage and higher fault-detection capability than our random test generation tool.

2. Framework

To automatically generate high-quality test suites for access control policies, we develop a novel framework based on change-impact analysis. Figure 1 shows the overview of the framework. The framework receives a set of policies under test and outputs a set of tests (in the form of request-response pairs) for policy authors to inspect for correctness. The framework consists of four major components: version synthesis, change-impact analysis, request generation, and request reduction. The key notion of the framework is to synthesize two versions of the policy under test in such a way that test coverage targets (e.g., certain policies, rules, or conditions) are encoded as the differences of the two synthesized versions. A change-impact analysis tool can then be leveraged to generate counterexamples to witness these

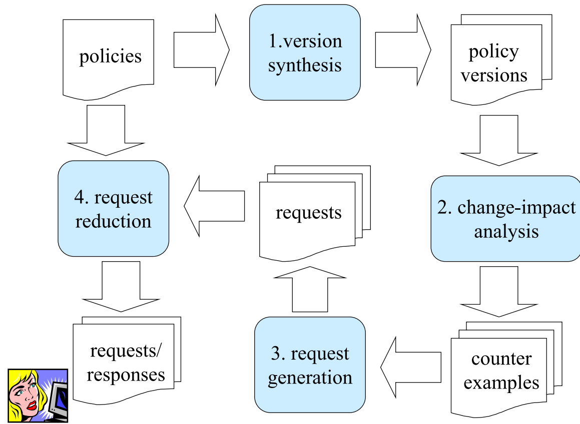


Figure 1. An overview of the framework.

differences, thus covering the test coverage targets. Based on the generated counterexamples, the framework generates tests (in the form of requests). Sometimes the number of generated tests is large and it is not feasible for developers to manually inspect their responses. To mitigate this issue, the final step of the framework reduces the number of generated tests by selecting tests based on policy structural coverage.

2.1. Version Synthesis

Given the policy under test, the version synthesis component synthesizes the policy's versions, which are later fed to a change-impact analysis tool. Our goal is to formulate the inputs to the change-impact analysis tool so that specifically targeted parts of the policy under test are covered. We provide two variants of version synthesis below called one-to-empty and all-to-negate-one. We discuss their analysis cost and the situations where they may not work well. Although the framework has been developed to support multiple policies, to simplify illustration we describe the synthesis variants with the case of a single policy p that contains n rules. To further illustrate the framework, we provide a concrete example XACML [1] policy in Figure 2. An XACML policy encodes rules in an XML syntax. Each rule has a set of constraints found in the `Target` elements that must be satisfied by a request in order for that rule to be applied. This example policy has two rules: the first one denies access requests for "dissemination" of the "demo:5" resource and the second one permits all other access requests. The first rule is defined by the `Rule` element on Line 2 and the `Target` element on Lines 3–21. The second rule is defined by the `Rule` element on Line 23. When multiple rules can be applied on a request, the decision of the first applicable rule will be returned (as specified by the "first-applicable" rule combining algorithm on Line 1).

```
 1<Policy Id="demo" RuleCombAlgId="first-applicable">
 2  <Rule RuleId="1" Effect="Deny">
 3    <Target>
 4      <Subjects> <AnySubjects /> </Subjects>
 5      <Resources>
 6        <Resource>
 7          <ResourceMatch MatchId="equal">
 8            <AttrValue>demo:5</AttrValue>
 9            <ResourceAttrDesignator AttrId="objectid" />
10          </ResourceMatch>
11        </Resource>
12      </Resources>
13      <Actions>
14        <Action>
15          <ActionMatch MatchId="equal">
16            <AttrValue>dissemination</AttrValue>
17            <ActionAttrDesignator AttrId="actionid" />
18          </ActionMatch>
19        </Action>
20      </Actions>
21    </Target>
22 </Rule>
23  <Rule RuleId="2" Effect="Permit" />
24</Policy>
```

Figure 2. An example XACML policy

one-to-empty: For each rule r in p, the two synthesized versions are an empty policy and a policy that contains only r. If r is a permitting rule, the synthesized empty policy is an empty denying policy. If r is a denying rule, the synthesized empty policy is an empty permitting policy. The reason for this mechanism is as follows. Comparing a permitting rule r with an empty permitting policy will not help generate requests to cover r because no counterexamples are generated for these two versions. Similarly, comparing a denying rule r with an empty denying policy will not help generate requests to cover r. This synthesis process is applied n times. So there are n pairs of policy versions synthesized for p. Consider the example policy written in XACML in Figure 2. The first pair of policy versions synthesized for this policy is an empty permitting policy and the original policy with Line 23 removed (i.e., the remaining rules). Applying change-impact analysis on each pair has low cost because each version contains only a single rule. Note that this variant does not take into account the interactions among different rules unlike the all-to-negate-one variant below.

all-to-negate-one: For each rule r in p, the two synthesized versions are p and p where the decision of r is negated. This process is applied n times so there are n pairs of policy versions synthesized for p. Again, consider the example policy in Figure 2. The first pair of policy versions synthesized for this policy is the original policy and the original policy with the effect on Line 2 changed to "Permit". Applying change-impact analysis on each pair has higher cost than the one-to-empty variant because the analysis complexity is heavily dependent on the size of the two versions rather than the differences between the two versions. Note that this variant takes into account interactions among different rules. This variant should be at least as good as the one-to-empty variant in terms of achieving policy structural

coverage and fault detection but it will have a higher computational cost, especially for large, complex policies.

The preceding two variants are specifically developed for achieving high rule coverage. Because the coverage of a rule implies the coverage of the policy that contains the rule, our two variants also indirectly target at achieving high policy coverage. In principle, we can develop variants of version synthesis for achieving high condition coverage by negating each condition one at a time.

2.2. Change-Impact Analysis

Given two versions of a policy, a change-impact analysis tool outputs counterexamples that illustrate semantic differences between the two policies. More specifically, each counterexample represents a request that evaluates to a different response when applied to the two policy versions. For example, a particular request r evaluates to permit for policy p but the same request evaluates to deny for policy p'. Change-impact analysis is usually performed on mature policies that are undergoing maintenance or updates to avoid accidental injection of anomalies. In our case, we exploit the functionality of change-impact analysis to automatically generate access requests by iteratively manipulating the inputs to a change-impact analysis tool.

2.3. Request Generation

Given two policies, a change-impact analysis tool outputs counterexamples that are evaluated to different responses against these two policies. We generate requests based on these counterexamples. Some change-impact analysis tools may produce abstract counterexamples, which are not immediately ready to be translated into a concrete request. For example, a change-impact analysis tool may produce an abstract counterexample for the policy in Figure 2 like if `((resource == demo:5) && (action == dissemination)),deny` becomes `permit`. Then we need to solve the constraint and derive one request (or optionally more requests) for the constraint. Other change-impact analysis tools may produce counterexamples at the concrete level, being the same as the level of requests. Our implementation leverages a change-impact analysis tool that produces counterexamples at the concrete level so we do not need to refer to a constraint solver to map from counterexamples to requests.

2.4. Request Reduction

The number of generated requests can be large for complex policies. Then it is infeasible for developers to inspect each request-response pair; therefore, we need to reduce the number of requests for inspection without incurring substantial loss in fault-detection capability. In particular, we select a subset of the request set such that the subset and superset achieve the same coverage. We have defined the request reduction problem [17] similar to the test minimization problem for program testing [10]:

Given: request set QS, a set of requirements r_1, r_2, ..., r_n that must be satisfied to provide the desired test coverage of the policies, and subsets of QS, Q_1, Q_2,..., Q_n, one associated with each of the r_is such that any one of the request q_j belonging to Q_i can be used to test r_i.
Problem: Find a representative set of requests from QS that satisfies all of r_is.

In the problem statement, the r_is can represent policy structural coverage requirements, such as covering a certain policy, a certain rule, and a certain condition. In a representative set of requests that satisfies all of the r_is, at least one request satisfies each r_i. We say a representative set is *minimal* if removing any request from the set causes the set not to be a representative set.

3. Implementation

This section presents the Cirg (**C**hange-**I**mpact **R**equest **G**eneration) tool that implements the framework and discusses technical challenges faced during the tool implementation. The tool consists of four major modules: the policy handler (for measurement support, version synthesis, and mutation), a script generator and program invoker (for change-impact analysis), a request generator (for request generation), and a request reducer (for request reduction). Our implementation operates on policies specified in the eXtensible Access Control Markup Language (XACML) [1]. XACML is a language specification standard designed by OASIS. Developers can use XACML to express domain-specific access control policies, access requests, and access responses. It offers a large set of built-in functions, data types, and combining logic. It also provides standard extension interfaces for defining application-specific features.

Policy Handler. The policy handler is used primarily to manipulate policies for experimentation and testing purposes. The policy handler is built on top of Sun's open source XACML implementation [2]. The policy handler is responsible for selecting which policy elements of the policy under test to encode to XACML during version synthesis. These modified policies are the inputs to the change-impact analysis tool. The policy handler performs policy mutation for mutation testing in order to measure fault-detection capability [16]. Mutation testing allows us to compare several techniques of request generation and selection in terms of fault-detection capability. In addition,

the policy handler plays the role of the PDP and is responsible for metric collection. During the experiment, the policy handler evaluates requests against policies in order to determine the responses. The coverage metrics and mutant-killing ratio described in Section 4.1 are also collected by the policy handler.

Script Generator and Program Invoker. The script generator and program invoker create executable scripts and programmatically executes the change-impact analysis tool, respectively. The change-impact analysis tool, Margrave [8], is a software tool suite written in PLT Scheme for analyzing access control policies written in XACML. We leverage Margrave's API to implement a request generation engine that efficiently achieves high policy structural coverage by exploiting its ability to perform change-impact analysis. Margrave represents XACML policies as multi-terminal binary decision diagrams (MTBDDs), which are a decision diagram that maps bit vectors over a set of variables to a finite set of results. The generated script uses Margrave's API to perform change-impact analysis on given policies. The script is dynamically generated as needed and subsequently executed by our Margrave invoker. The execution of Margrave can be expensive depending on the size of the policies being compared and how much the policies vary semantically. The outputs from the Scheme script are the results of the change-impact analysis. The outputs are an enumeration of all counterexamples or requests whose decisions change from `Deny` to `Permit` or vice versa. These outputs are used to drive the request generator.

Request Generator. The request generator parses Margrave's outputs and converts them into requests encoded as XACML. The request generator also generates `RequestCtx` objects on demand. These `RequestCtx` objects are the runtime representation of the request evaluated by the policy handler to generate the response.

Request Reducer. Our previous work [17] developed the request reducer for greedily removing a request from a request set if and only if the request does not increase any of the coverage metrics that are achieved by previously evaluated requests in the request set. More specifically, from a request set, we evaluate each request against the policy in order to both compute the response and measure the coverage. If the coverage increases during the evaluation of the request, then that request is added to the reduced request set; otherwise, it is removed. Note that this greedy algorithm may not produce a minimal representative set. In practice, it does, however, often produce a representative set whose size is near the size of a minimal representative set.

4. Experiment

This section presents the experiment that we conducted to evaluate our test generation approach via change-impact analysis. We first describe the experiment's objective and measures as well as the experiment instrumentation. We then present and discuss the experimental results. Finally we discuss threats to validity.

Table 1. Policies used in the experiment.

policy	# pol set	# pol	# rule	# cond	# mutants
conference	0	1	15	0	66
default-2	1	13	13	12	130
mod-fedora	1	12	12	10	120
continue-a	111	266	298	0	3565
pluto	0	1	21	0	89

4.1. Objective and Measures

The objective of the experiment is to investigate the following questions:

1. Can change-impact analysis be used to effectively generate tests that achieve high policy structural coverage?
2. Can change-impact analysis be used to effectively generate tests that have high fault-detection capability?
3. How does test generation via change-impact analysis compare to the existing random test generation in terms of policy structural coverage and fault-detection capability?

To help answer these questions, we collect several performance metrics used to compare the presented test-generation techniques. These metrics are measured for each policy under test and each test-generation technique.

Policy coverage. The number of policies involved in evaluating the request set divided by the total number of policies.

Rule coverage. The number of rules involved in evaluating the request set divided by the total number of rules.

Condition coverage. The number of true or false conditions involved in evaluating the request set divided by two times of the total number of conditions.

Mutant-killing ratio. Given a request set, the policy under test, and the set of generated mutants, the mutant-killing ratio is the number of mutants killed by the request set divided by the total number of mutants.

Test count. The size of the request set or the number of tests generated by the chosen test-generation technique. For testing access control policies, a test is synonymous with a request.

Reduced-test count. Given a policy and the generated set of requests, the reduced test count is the size of the reduced request set based on policy structural coverage.

An ideal test-generation technique should have high values for the first four metrics (policy coverage, rule coverage, condition coverage, and mutant-killing ratio) and low values for the last two metrics (test count and reduced-test count). A low test count is highly desirable because the request-response pairs may need to be inspected manually to verify that the policy specification exhibits the intended policy behavior. Intuitively a set of requests that achieve high policy structural coverage are more likely to reveal faults. This notion is easy to understand because a fault in a policy rule that is never covered by a request would never contribute to a response and thus a fault in the rule cannot possibly be revealed. We leverage our work [16] on policy mutation testing to obtain the mutant-killing ratio and use it as a measure of fault-detection capability.

4.2. Instrumentation

We compare three test-generation techniques, namely, a random test-generation technique [17] and two test generation techniques based on the two variations of version synthesis. Furthermore, we apply a greedy test reduction algorithm on each set of generated tests resulting in a reduced test set thus a total of six request sets are evaluated for each policy. We used five XACML policies collected from four different sources as subjects in our experiment. Table 1 summarizes the statistics of each policy including the subject names, the numbers of policies, rules, and conditions in the policy and the number of mutants generated for each subject. Among these five subjects, the `conference` policy is a slightly modified version of the policy used by Zhang et al. [19]. The most complex policy, `continue`, is an example policy used by Fisler et al. [8]. The `continue` policy is a policy for a real web application for supporting conference submissions, reviews, discussion and notifications. The `default-2` and `mod-fedora` policies approximate the access control of an earlier version of Fedora[1]. Fedora is an open source software that gives organizations a flexible service-oriented architecture for managing and delivering digital content. Finally, the `pluto` policy is used for the ARCHON[2] system. ARCHON is a digital library that federates physics collections with varying degrees of meta data richness.

4.3. Results

Table 2 summarizes the number of generated tests and reduced tests for each policy and each test-generation technique. We observe significant reductions in request set sizes for each of the request generation techniques. On average, we observe 80.8%, 82.8%, and 80.5% reductions in request set size for the random, one-to-empty, and all-to-negate-one techniques, respectively. The all-to-negate-one generation technique did not execute within a reasonable amount of time for continue-a and pluto due to the high runtime costs associated with change-impact analysis. This result suggests that the all-to-negate-one approach to version synthesis suffers from scalability issues. The central issue is the high memory and computational costs of change-impact analysis when the policy versions generate large MTBDD's. While the number of rules in the policy versions provide an indication to the complexity of the MTBDD, the number of constraints in each rule and number of attribute id-value pairs is a better measure. Although the pluto policy

Table 2. Number of tests generated.

	random		one-to-empty		all-to-negate-one	
policy	gen	red	gen	red	gen	red
conference	50	2	43	15	21	15
default-2	50	7	36	12	64	10
mod-fedora	50	10	31	13	110	13
continue-a	50	29	1456	257	-	-
pluto	50	0	170	1	-	-

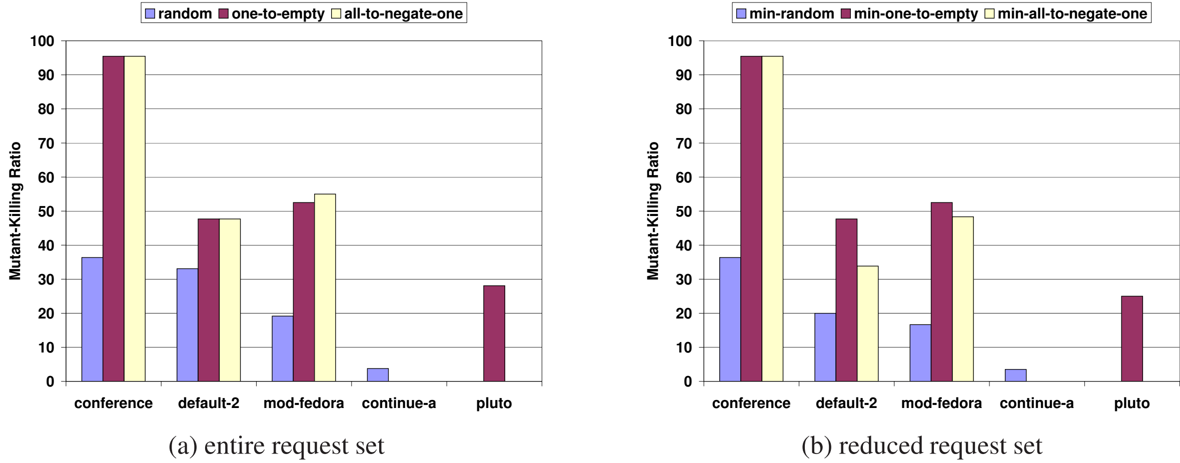
Table 3 summarizes the structural coverage metrics for each policy and each request set. We do not show the minimized request sets because they, by definition, have equivalent coverage as their superset. Each row of the table corresponds to a particular policy and each column group corresponds to a request set. Within each column group, we show the policy, rule, and condition coverage percentages. The character "-" indicates that there are no policy elements of that type and thus coverage cannot be computed. Most test generation techniques achieve close to 100% policy coverage for all subjects because it is the most coarse measure of structural coverage. We observe an average 39.5% and 64.7% improvement over the random technique in rule coverage for the one-to-empty and all-to-negate-one techniques, respectively. The most interesting policies are the continue-a and pluto policies. The randomly generated requests do not achieve sufficient rule coverage whereas the one-to-empty generation technique does. The pluto policy is complex enough that randomly generated requests fail to apply to any policy elements (i.e., they are not covered). The one-to-empty technique does better but due to its failure to account for rule interactions, it also performs poorly on rule coverage.

Figure 3 illustrates the mutant-killing ratios for the entire request set and the corresponding reduced request set, respectively. The data is grouped by policy and seeks to compare the request generation techniques in terms of fault detection capability. We observe that, on average, the one-to-empty and all-to-negate-one request generation techniques outperform the random technique by 34.1% and 21.1% in

[1] http://www.fedora.info
[2] http://archon.cs.odu.edu/

Table 3. Measured policy, rule, and condition coverage.

	random			one-to-empty			all-to-negate-one		
policy	pol %	rule %	con %	pol %	rule %	con %	pol %	rule %	con %
conference	100	13.33	-	100	100	-	100	100	-
default-2	100	100	75	100	100	50	100	100	50
mod-fedora	100	58.33	70	100	100	50	100	100	60
continue-a	99.62	4.70	-	100	69.13	-	-	-	-
pluto	0	0	-	100	4.76	-	-	-	-



(a) entire request set

(b) reduced request set

Figure 3. Mutant-killing ratios of generated request set

terms of fault-detection capability, respectively. The results indicate that change-impact analysis can be used to generate tests with higher fault-detection capability than random techniques. Furthermore, on average we can reduce the size of the request set by 80% while only incurring a 2.5% loss in fault-detection capability. This result is illustrated by comparing the near identical mutant-killing ratios for the entire request set and reduced request set in Figures 3 (a) and (b).

In summary, we found that change-impact analysis can be used to generate tests that achieve high policy structural coverage after we carefully synthesize the inputs to a change-impact analysis tool. Furthermore, the test generation techniques based on change-impact analysis achieve higher policy structural coverage and higher fault-detection capability than the random technique. We also found a high correlation between structural coverage and fault-detection capability. The fact that the mutant-killing ratio for all-to-negate-one decreases between the entire request set and reduced request set indicates that the structural coverage, as it is defined now, is an insufficient selection criterion. Our current policy structural coverage corresponds to statement or branch coverage in program testing. We expect that we can achieve higher fault-detection capability when adopting stronger policy structural coverage criteria such as one that corresponds to path coverage in program testing. We plan to explore this direction and further synthesize versions for achieving stronger policy structural coverage.

4.4. Threats to Validity

The threats to external validity primarily include the degree to which the subject policies, policy mutator, coverage metrics, and test sets are representative of true practice. These threats could be reduced by further experimentation on a wider type and larger number of policies. The threats to internal validity are instrumentation effects that can bias our results such as faults in Sun's XACML implementation, faults in Margrave's Scheme API, Margrave's limitations, as well as faults in our own implementations.

5. Related Work

Several test generation tools for software programs were developed based on static verification tools. Beyer et al. [4] developed a test generation tool for C programs based on BLAST [11]. Csallner and Smaragdakis [6] developed the CnC test generation tool that generates tests based on counterexamples produced by the ESC/Java [15] static verification tool. Different from these preceding approaches, our approach is to generate tests for policy specifications rather

than software programs. In addition, our approach generates tests by feeding synthesized versions to a change-impact analysis tool unlike most preceding tools that insert synthesized assertions into code and feed the code to a static verification tool.

A number of researchers have developed tools that generate tests for specifications by synthesizing inputs to a model checker and exploiting the counterexamples generated by the model checker. For example, Gargantini and Heitmeyer [9] insert assertions to execution branches of specifications, feed them to a model checker, and use model-checker-generated counterexamples to derive test sequences. Ammann et al. [3] mutate both specifications and properties, use a model checker to generate counterexamples for the mutated specifications and properties, and derive tests out of the counterexamples. Our approach generates tests by synthesizing inputs to a change-impact analysis tool rather than a model checker. In addition, our approach targets at a special type of specifications: access control policy specifications written in XACML.

There are several verification tools for verifying properties against XACML policies [1]. Hughes and Bultan [12] translated XACML policies to the Alloy language [13] and check their properties using the Alloy Analyzer. Zhang et al. [20] developed a model-checking algorithm and tool support to evaluate access control policies written in *RW* languages, which can be converted to XACML [19]. Fisler et al. [8] developed the Margrave tool, which uses multi-terminal binary decision diagrams [5] to verify user-specified properties and perform change-impact analysis. These existing approaches assume that policies are specified using a simplified version of XACML. In addition, most of these approaches require users to specify a set of properties to be verified; however, policy properties often do not exist in practice. Our approach is developed based on Margrave's change-impact analysis feature. In general, test generation for policies can also be conducted by synthesizing properties to exploit the preceding policy verification tools that require properties.

6. Conclusion

We have developed a novel framework and its supporting tool called Cirg that automatically generates tests for access control policies based on a change-impact analysis tool such as Margrave [8]. Because the number of generated tests is often high for complex policies, Cirg reduces the size of the generated test set based on policy structural coverage so that developers can inspect the tests with reasonable efforts. We have conducted an experiment that assesses Cirg on a set of policies collected from various sources. The experimental results show that Cirg can generate tests for complex policies to achieve high policy structural coverage. The results also show that Cirg can generate tests that have a higher policy structural coverage and higher fault-detection capability than the existing random test generation. Previous research [14, 18] synthesizes inputs to a model checker or a test generator in particular ways so that it can be leveraged to generate regression tests for multiple program versions. Within our knowledge, we are the first to go for the reverse direction: synthesizing versions as inputs to a change-impact analysis tool in particular ways so that it can be leveraged to generate tests for a single version. We believe that our general idea has the potential to expand its application scope to go beyond the domain of access control policies.

References

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

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

[3] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. 2rd IEEE International Conference on Formal Engineering Methods*, pages 46–54, 1998.

[4] D. Beyer, A. J. Chlipala, and R. Majumdar. Generating tests from counterexamples. In *Proc. 26th International Conference on Software Engineering*, pages 326–335, 2004.

[5] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis*, pages 149–169, 1993.

[6] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering*, pages 422–431, 2005.

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

[8] 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*, pages 196–205, 2005.

[9] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. 7th ESEC/FSE*, pages 146–162, 1999.

[10] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, 2002.

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

[13] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 8th ESEC/FSE*, pages 62–73, 2001.

[14] B. Korel and A. M. Al-Yami. Automated regression test generation. In *Proc. 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 143–152, 1998.

[15] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, Palo Alto, CA, October 2000.

[16] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proc. 11th International Conference on World Wide Web*, 2007.

[17] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. 8th International Conference on Information and Communications Security*, pages 139–158, 2006.

[18] L. Xu, M. S. Dias, and D. J. Richardson. Generating regression tests via model checking. In *Proc. 28th International Computer Software and Applications Conference*, pages 336–341, 2004.

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

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