Testing and Analysis of Access Control Policies

Evan Martin
North Carolina State University, Raleigh, NC 27695
eemartin@ncsu.edu
http://ase.csc.ncsu.edu

Abstract

Policy testing and analysis are important techniques for high assurance of correct specification of access control policies. We propose a set of testing and analysis techniques for access control policies and tools for empirically investigating and evaluating the proposed techniques. We propose a fault model for access control policies and investigate various fault types and their frequencies of occurrence in policy development; we develop a mutation testing framework that implements the fault model; we propose and investigate various coverage criteria for testing access control policies; we develop various test generation techniques and evaluate them using the coverage criteria and mutation testing framework; we develop a policy model to facilitate refactoring, performance optimizations, dependency identification, and other types of static analysis. To make our discussion concrete, we choose to present our techniques in the context of XACML. Note that since XACML is an application-independent, generic access control policy language, our techniques can be equally applied to test policies written in other languages.

1. A Fault Model and Mutation Testing

A fault model is an engineering model of something that could go wrong in the construction or operation of a piece of equipment, structure, or software. In our case, we will model things that could go wrong when constructing an access control policy. With this fault model, we can guide the development of testing techniques and investigate these techniques’ effectiveness against the fault model. Any fault results in a semantic change in the policy but we broadly categorize faults as being semantic or syntactic as follows:

Semantic Faults. Semantic faults are more elusive because they involve incorrect use of the logical constructs of the policy specification language. For XACML policies, these logical constructs include policy or rule combining algorithms, policy evaluation order, rule evaluation order, and various functions found in the condition. Because these are logical errors in the construction of the policy, it is unlikely that static analysis can find such errors. We define and implement several mutation operators that emulate semantic faults [3].

Syntactic Faults. Syntactic faults are easier to make and consist of simple typos that result in a syntactically correct policy but a semantically faulty one. Indeed syntactic faults may result in syntactically incorrect policies but we assume that basic static analysis tools exist to check for such inconsistencies. For example, in XACML, an XML schema definition (XSD) can be used to check for obvious syntactic flaws. Syntactic faults that do not violate the XSD can occur due to typos in attribute values. We define and implement three mutation operators that emulate syntactic faults [3].

Metrics and Evaluation. In order to support investigating relative occurrence frequencies of fault types, we will develop a policy editing tool. The editor facilitates the identification and logging of policy faults, refactoring support, optimization, and other static analysis. We measure the percentage of fault types that are not captured in the initially proposed fault model but are observed in policy development. We also measure the distribution of the fault types observed in policy development. The measurement results will offer insights in proposing improvements of policy language designs and tool support to guard against common faults, and focus testing efforts against common faults. We have conducted initial investigations of the proposed fault model in terms of detection rates for each fault type when the locations of the faults are covered by the test suite [2–4].

2. Structural Coverage Criteria

Basic Coverage Criteria. We propose basic coverage criteria based on observing whether each individual policy element is involved when a request is evaluated. We propose to reduce a test suite while preserving its achieved coverage and develop supporting tools for coverage measurement and test reduction. Given a policy, we would like to generate a set of requests, and check whether the access
control decisions on these requests are expected. Any unexpected decision indicates potential errors in the specification of the policy. If no requests are evaluated against a rule during testing, then potential errors in that rule cannot be discovered. Thus, it is important to generate requests so that a large portion of rules are involved in the evaluation of at least one of the requests. In XACML, we can see there are three major entities: policies, rules for each policy, and a condition for each rule. We define three policy coverage metrics for each of these entities [4].

Improved Coverage Criteria. There is one major issue with the basic coverage criteria primarily caused by not taking the effect of policy or rule combination and ordering into account. In particular, using the coverage criteria could lead to a false sense of high confidence (i.e., the coverage criteria are not sufficient for helping select or generate a good-enough set of tests). To address the issue, we propose improved policy coverage criteria by taking policy and rule combination and ordering into account.

Metrics and Evaluation. We have implemented an initial prototype for coverage measurement and test reduction based on the proposed basic coverage criteria [4]. The preliminary results showed that test reduction based on the basic coverage criteria substantially reduce the size of generated requests and incur relatively low loss in fault-detection capability. We expect that reduction based on improved coverage criteria would incur even lower loss in fault-detection capability and we will implement it and empirically investigate tradeoffs between reduced sizes and reduced fault-detection capability for both coverage criteria.

3. Test Generation

We propose to develop three test generation techniques, which have different levels of analysis cost and quality of generated tests.

Random Test Generation. We have developed a technique for randomly generating requests given only the policy under test [2]. This technique serves as a baseline test generation technique: our two more advanced techniques should perform at least better than this baseline technique in terms of the quality of generated tests. The random technique analyzes the policy under test and generates requests by randomly selecting requests from the set of all possible requests.

Test Generation based on Solving Single-Rule Constraints. To generate tests for achieving high coverage based on basic coverage criteria, we developed a technique that considers each rule in isolation and attempts to satisfy the constraints required for that rule to be applied [1]. A request set is generated that satisfies all possible combinations of truth values for each independent clause. Therefore, a predicate with \( n \) independent clauses has \( 2^n \) possible assignments and so at most \( 2^n \) requests are generated for each rule.

Test Generation based on Solving Multiple-Rule Constraints. The preceding technique is designed to make each individual rule to be applicable, therefore achieving good structural coverage based on the basic coverage criteria. However, it may not be effective in generating tests to achieve high structural coverage based on the improved coverage criteria. To specifically generate tests to satisfy the improved coverage criteria, we will improve the preceding technique by solving multiple-rule constraints instead of single-rule constraints.

Metrics and Evaluation. We propose to use the following main metrics to assess the quality of generated tests: (1) coverage based on basic and improved structural coverage criteria; (2) fault-detection capability measured by mutant-killing ratios; (3) the number of generated tests. An ideal generated test suite would achieve high coverage and fault-detection capability and include as few tests as possible (because in the end we need manual inspection to determine whether request-response pairs are correct). We will use the proposed three techniques to generate tests for policy examples collected from the Internet. We then measure and compare the size and quality of generated test suites. We have implemented an initial prototype for the first two techniques [1]. The preliminary results showed that the second technique outperforms the baseline random technique significantly in achieving high basic coverage and mutant-killing ratios. However, a significant percentage of mutants still cannot be killed by the tests generated by the second technique. After we implement all the proposed techniques and tools, we will empirically investigate the relationship between test generation techniques, structural coverage criteria, and fault-detection capability.

References