Security Policy Testing via Automated Program Code Generation (Extended Abstract)

Ting Yu
North Carolina State University
yu@csc.ncsu.edu

Dhivya Sivasubramanian
North Carolina State University
dsivasu@ncsu.edu

Tao Xie
North Carolina State University
xie@csc.ncsu.edu

1. INTRODUCTION

Access control is one of the fundamental security mechanisms for information systems. It determines the availability of resources to principals, operations that can be performed, and under what circumstances. Traditionally the enforcement of access control is often hardcoded in applications or systems; such hardcoding makes it hard to verify the correctness of access control and to accommodate changes of security requirements. Recently, access control policies have been increasingly separated from enforcement mechanisms. An access control policy is explicitly specified using certain policy languages with well-defined syntax and semantics. An application then consults the policy during runtime to determine whether an access request from a principal should be allowed or denied. There are two main advantages of this approach. First, security officers can now perform systematic and formal security analysis on access control policies. Second, by separating policies from enforcement mechanisms, it is possible to change policies without affecting the underlying mechanisms, and vice versa.

One challenging problem in access control is to ensure the correct specification of access control policies. A large number of security problems are caused by policy misconfigurations. Manual inspection of policies for correctness is tedious and often incomprehensive for today's sophisticated access control policies. Similar to quality assurance of software systems, testing techniques [4–6] have been proposed recently to ensure the correctness of access control policies, including policy-language-specific coverage criteria [5, 6] and test generation algorithms [4, 5]. One limitation of these previous testing techniques is that for every different policy language, a different set of coverage criteria and test generation algorithms need to be specifically defined and developed.

In this paper, we propose a new general policy-language-independent framework for policy testing. In our new framework, we view an access control policy as a software module, which, when given an access request from a principal, returns `permit` or `deny`. Then for each policy language, we develop a translator that automatically translates a policy written in that policy language to a software module (e.g., a Java class or a Java method). We can then directly reuse existing software testing tools to test the policy. Note that during this process, we do not need to define coverage criteria or develop specific test generation algorithms or tools for each language, as the translated software module already encodes the important structure of the policy, and existing software testing tools can take the structure into consideration when conducting software testing.

The rest of the paper is organized as follows. Section 2 presents our general framework for policy testing through automated program code generation. Section 3 illustrates how to use the proposed framework to test XACML policies. Section 4 reports the preliminary evaluation results. Section 5 concludes.

CSIIRW'09, April 13–15, Oak Ridge, Tennessee, USA.

2. PROPOSED FRAMEWORK

Figure 1 shows the major functional components of a typical access control system [1]. The Policy Enforcement Point (PEP) interacts directly with users. When a user tries to access a resource, the PEP forms an appropriate access request that includes the attributes of the requester, the requested action, and the requested resource, and passes the request to the Policy Decision Points (PDP). The PDP looks up the access control policy that applies to the request, and returns a response to the PEP. The PEP then correspondingly permits or denies the user's action. If the access control policy contains faults, it can cause either unauthorized access or denial of intended access, even when the PEP and the PDP are correctly implemented.

The idea of policy testing is to generate a comprehensive set of test inputs (each of which is an access request) and feed them to the PDP to see whether the decisions from the PDP are as expected. The key problem is how to generate high-quality test inputs, so that policy faults can be detected.

Existing techniques to policy testing follow the typical software testing practice, which defines coverage criteria for policy testing, and then designs test generation algorithms and tools to maximize the coverage achieved by generated test inputs. Various policy languages have been proposed in the literature and often have quite different structures (e.g., some are rule-based while others are based on logical inferences), as they are designed for different types of applications. With these previous techniques to policy testing, we would have to identify different coverage criteria for each language and design different test generation algorithms and tools accordingly. Meanwhile, there are already many mature and successful techniques and tools for software testing. It would be much beneficial and cost-effective if we can reuse existing software testing tools to test security policies. Based on this observation, we propose a policy testing framework based on automated program code generation.

Figure 2 shows the overview of the framework. One key component of the framework is a code generator. This component is language-specific, i.e., for each policy language, we need to de-

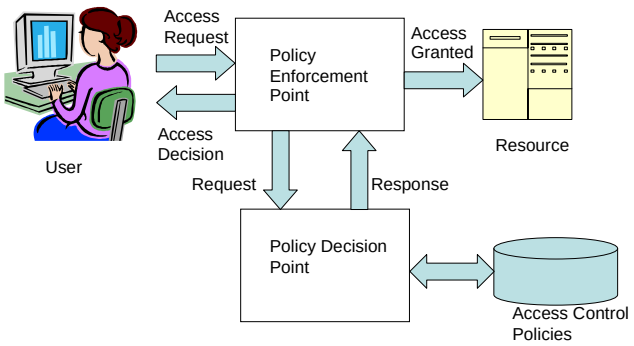

Figure 1: Major components of an access control system

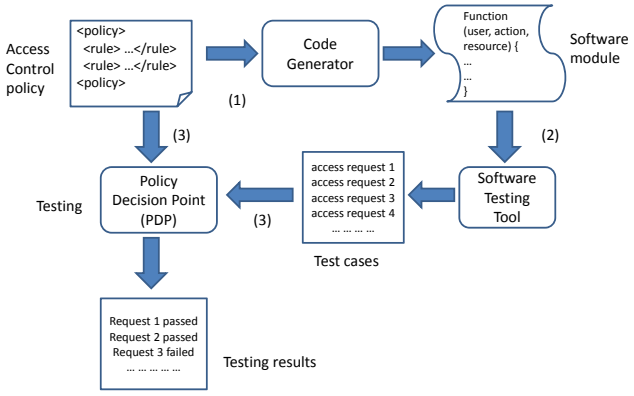

Figure 2: Proposed policy-testing framework via program code generation

velop a code generator. Given an access control policy written in a policy language, the code generator generates a software module that is semantically equivalent to the policy. Applying this component is the first step in our framework. The programming language for the generated software module can be flexible. Its choice is largely determined by the software testing tools that are used in the second step.

In the second step, the generated software module becomes the program under test for an existing software testing tool, which generates a set of test inputs. Note that since the software module is semantically equivalent to the access control policy, the output from the software testing tool (i.e., test inputs) can be straightforwardly translated to access requests from users with different attributes.

In the third step, we feed each access request to the PDP, which consults the access control policy to make a decision for each request. Combination of the outputs from the PDP with the generated access requests from the second step, we can further perform manual analysis of the testing results and detect any unexpected behavior (when policy properties are available, analysis of the testing results can be automated).

Compared with existing techniques to policy testing, our framework has only one language-specific component. Note that since policy languages have well-defined semantics, the translation from an access control policy to a software module is often straightforward. The structure of the generated software module inherits that of the original policy. The coverage of the software-module structure can be easily mapped back to the coverage of the corresponding policy structure.

3. XACML POLICY TESTING

In this section, we show how we test XACML policies with the proposed framework. The eXtensible Access Control Markup Language (XACML) [8] is an XML-based syntax used to express policies, requests, and responses. The five basic elements of XACML policies are *PolicySet*, *Policy*, *Rule*, *Target*, and *Condition*. A policy set is a container that holds other policies or policy sets. A policy is expressed through a sequence of rules. With multiple policy sets, policies, and rules, XACML must have a way to reconcile conflicting rules (i.e., multiple applicable rules with different decisions). A collection of combining algorithms [8] serves this purpose. Each algorithm defines a different way to combine multiple decisions into a single decision. Both policy combining algorithms and rule combining algorithms are provided.

To aid in matching requests with appropriate policies, XACML provides a target, which is basically a set of simplified conditions for the subject, resource, and action that must be met for a policy set, policy, or rule to be applied to a given request. Once a policy or policy set is found to be applied to a given request, its rules are evaluated to determine the response.

Attributes describe the subject, resource, and action of an access request. These attributes are compared with the conditions in policies or rules to determine whether these attributes can be applied to a request.

We next show a simplified example of XACML policies:

```
<Policy PolicyId="univ"
  RuleCombinationAlgId="permit-override">
<Target>
  <Subjects><AnySubjects/></Subjects>
  <Resources><Resource><AnyResource/>
  </Resource></Resources>
  <Actions> <AnyAction/> </Actions>
</Target>
  <Rule RuleId="1" Effect="Permit">
   <Target>
    <Subjects><Subject>Faculty</Subject>
    </Subjects>
    <Resources>Grades</Resources>
    <Actions><Action>Assign</Action>
    <Action>View</Action></Actions>
   </Target></Rule>
  <Rule RuleId="2" Effect="Deny">
   <Target>
    <Subjects><Subject>Student</Subject>
    </Subjects>
    <Resources>Grades</Resources>
    <Actions><Action>Assign</Action></Actions>
   </Target>
  </Rule>
  <!-- A final, "fall-through" rule
    that always Denies -->
  <Rule RuleId="FinalRule" Effect="Deny"/>
</policy>
```

The preceding policy governs access to grades in a university. It contains three rules. The first one states that entities with faculty roles can assign and view grades. The second rule states that students are not allowed to assign grades. If both rules cannot be applied to a request (e.g., a request from a staff), the third rule states that the request should be denied by default. When two rules can be applied to the same request (e.g., a faculty may be a part-time student of another department), then permit rules override deny rules, since its rule combining algorithm is `permit-override`.

Due to well-defined semantics of XACML, it is straightforward to translate a policy in XACML into the source code of a software module in any programming language. Specifically, an XACML policy can be viewed as a set of predicates combined by logical operators. The predicates correspond to rules in the policy, and the variables in the predicates correspond to attributes in the policy. Through a parsing pass of an XACML policy, we can easily identify all the predicates and attributes relevant to policy evaluation. These attributes are then classified according to their types (subject, resource, or action). A rule can then be translated into a conditional statement, specifying constraints on different types of attributes. An entire policy can be mapped to a series of conditional statements. The ordering of these statements is determined by the rule combining algorithm in the policy.

To facilitate test generation, we model a request as a sequence of boolean variables, indicating the presence of each attribute value (the true value indicating the presence of the attribute value in the request). In the program code generated for the policy, we encode this sequence of boolean variables as an array of boolean values, being the parameter of a method. In particular, we translate the preceding policy into the following program code:

```
String univ(boolean[] request) {
 int Faculty = 0, Student = 1, OtherRole = 2;
 int Grades = 3, OtherResource = 4;
 int Assign = 5, View = 6, OtherAction = 7;

 if (request[Faculty]}
   if (request[Grades])
     if (request[Assign] || request[View])
       return "permit";
 if (request[Student])
   if (request[Grades])
     if (request[Assign])
       return "deny";
 return "deny";
}
```

Note that if there are other *permit* rules in the policy, these rules are moved to before all the *deny* rules to preserve the semantics of the *permit-override* rule combining algorithm.

Our implementation automatically translates an XACML policy into a Java class with a single method. We then use jCUTE [7], a dynamic symbolic execution engine to generate test inputs for this single method. jCUTE logs generated test inputs that led to new feasible execution paths in the Java code translated from the policy. These inputs form optimal, non-redundant test inputs for testing the Java code (and thus the corresponding policy). We choose a symbolic-execution-based testing technique for policy testing, because test inputs generated by this technique are likely to detect fault types commonly occurring in policy specifications. Specifically, the rules in a policy are a series of conditional statements and a symbolic-execution-based testing technique can easily solve each of the constraints from the conditional statements. Appending new rules to the end of a policy often may not affect the coverage of previous rules. Thus, new test inputs can be generated and added to existing test inputs.

The preceding discussion also shows another benefit of our framework. Once policies are translated to software modules, we can readily explore existing types of software testing techniques and tools to find the most appropriate ones in this application context.

Note that we do not need to perform any test request reduction as in other existing test generation tools. Redundant requests have already been removed by jCUTE. This case is another example benefit in reusing existing testing techniques for policy testing.

policy	# set	# policy	# rule	# cond
codeA	5	2	2	0
default-2	1	13	13	12
demo-11	0	1	3	4
demo-26	0	1	2	2
demo-5	0	1	3	4
mod-fedora	1	12	12	10
simple-policy	1	2	2	0

Figure 3: Statistics of policies used in preliminary evaluation

4. PRELIMINARY EVALUATION

We compare the quality of the test inputs generated by our approach with that of Targen [4], a policy testing tool specifically designed for XACML. Targen defines a set of coverage criteria [6] for XACML policies, including policy coverage ratio, rule coverage ratio, and condition coverage ratio. It can also generate requests directly to try to maximize different coverage ratios and perform request reduction. We conduct our preliminary evaluation over various policies previously used in evaluating policy testing [5]. Figure 3 shows statistics of these policies, including the numbers of policy sets, policies, rules, and conditions.

Figure 4 shows the empirical results when comparing Targen with our approach using jCUTE. Column 1 shows the names of the policies. Columns 2, 3, and 4 show the policy, rule, and condition coverage, respectively, when using the Targen approach. Columns 6, 7, and 8 show the policy, rule, and condition coverage, respectively, when using jCUTE. The results show that requests generated by our approach achieve 100% policy, rule, and condition coverage. Targen achieves 100% policy and rule coverage; however, the achieved condition coverage is not 100% because Targen currently does not support test generation specifically for conditions. Another difference to be noted is that the requests generated by Targen consist of some redundant cases – requests that do not cause any increase in any policy, rule, or condition coverage. Targen uses a greedy reduction to identify requests that can cause an increase in coverage. This step can cause extra overhead because redundant requests are generated in the first place and then the generated requests are reduced through some greedy away. In our approach, however, the request-generation process (on leveraging jCUTE) itself ensures that a generated request covers at least one new policy element that has not been covered by previously generated requests. Such a benefit is due to that in our approach only the constraints along the path to a rule are solved. Performing the greedy reduction implemented in Targen over the requests generated by our approach showed that there was 0% reduction in the requests generated by our approach. This result shows that our requests are optimal since they achieve 100% policy, rule, and condition coverage and no reduction is needed based on the observation of 0% reduction.

We also use mutation testing [3] to assess the fault-detection capability of the generated test inputs. Mutation testing seeds simple faults in the original program, generates programs that are close to the original program, and sees how high percentage of seeded faults can be detected with a set of test inputs. Specifically this technique takes advantage of the coupling effect. In programming, the coupling effect can be defined on the basis of the empirical observation that complex faults occur due to the combination of simple faults. So, if we seed simple faults into a program by means of simple changes based on mutation operators and if these faults can be de-

	targen				jcute			
policy	pol %	rule %	con %	mut kill%	pol %	rule %	con %	mut kill%
codeA	100	100	n/a	36.36	100	100	n/a	41.8
default-2	100	100	100	50	100	100	100	29.23
demo-11	100	100	75	77.78	100	100	100	83.33
demo-26	100	100	50	78.57	100	100	100	78.57
demo-5	100	100	75	78.95	100	100	100	78.95
mod-fedora	100	100	100	56.67	100	100	100	36.66
simple-policy	100	100	n/a	44.44	100	100	n/a	55.5

Figure 4: Comparison of policy coverage ratios and mutant-killing ratio

tected by a set of test inputs, then we can be assured that this set of test inputs can also be used to detect complex faults that occur as a combination of these simple faults. In other words, mutation testing measures the capability of a set of test inputs in detecting simple faults, which could be used as an indication of its capability to detect complex faults.

To use this technique in the context of testing access control policies, it is necessary to identify simple faults in the context of an XACML policy. For example, a simple fault that a user makes when writing an XACML policy is to write a policy with a sequence of rules but write a target that is not applicable to any valid request. Here the policy is not applicable to any request. This fault can be emulated by creating a mutant policy with a target value that always evaluates to false. Another mutant policy could be one with the target always being applicable; this fault ensures that all the requests being evaluated are applicable to the policy, and the rules in the policy are always valuated. Based on this idea, Evan and Xie [5] have developed a set of mutation operators for an XACML access control policy, and a mutation testing tool to assess fault-detection capability of a set of requests.

Based on this mutation testing tool, we compare the fault-detection capability of our approach and that of Targen. The set of requests generated by both approaches are evaluated against the original policy and a mutant policy. If the evaluation results are different, the mutant policy is said to have been killed. If the evaluation results remain the same, the mutant policy lives. Columns 5 and 9 in Figure 4 show the mutation-killing ratios of Targen and our approach, respectively. We observe that the fault-detection capability of requests generated by our approach performs better or as good as the ones generated by Targen in most cases.

In summary, we observe a comparable capability of the test inputs generated by Targen and those generated by our approach while our approach brings in the benefits as described earlier.

5. CONCLUSION

In this paper, we have presented a new general framework for policy testing via automated program code generation. This framework allows to easily reuse existing software testing techniques and tools to ensure the correctness of security policies. We have demonstrated the effectiveness of the proposed approach by empirically comparing it with an existing policy testing tool specifically designed for XACML.

In future work, we plan to further evaluate the effectiveness of our approach by applying it to other policy languages such as Ponder [2]. We also plan to adapt our approach to handel stateful policies such as those for managing roles in RBAC and stateful firewall policies.

Acknowledgments

This research was sponsored by the NSF through CyberTrust grants IIS-0430166, CNS-0716579, and CNS-0716210.

6. REFERENCES

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

[2] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Proc. POLICY*, pages 18–38, 2001.

[3] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[4] Evan Martin and Tao Xie. Automated test generation for access control policies. In *Supplemental Proc. ISSRE*, 2006.

[5] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proc. WWW*, pages 667–676, 2007.

[6] Evan Martin, Tao Xie, and Ting Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. ICICS*, pages 139–158, 2006.

[7] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. CAV*, pages 419–423, 2006.

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