Automatic XACML requests generation for testing access control policies

Li Yongchao  Li You  Wang Linzhang  Chen Guanling

Postprint Version. Originally Published in: International Conference on Software Engineering and Knowledge Engineering(SEKE 2014)

Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.
Automatic XACML requests generation for testing access control policies

Yongchao Li, You Li, Linzhang Wang
State Key Laboratory of Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, 210046, China
Email: {lyc, le086}@seeg.nju.edu.cn, Izwang@nju.edu.cn

Guanling Chen
Department of Computer Science
University of Massachusetts Lowell
1 University Avenue, Lowell, Massachusetts, 01854, USA
Email: glchen@cs.uml.edu

Abstract—XACML has become increasingly popular for specifying access control policies in mission critical domains to protect sensitive resources. However, manually crafted XACML policies may contain errors which can only be identified with manual policies review. Recent progress in policy testing still requires tedious and inefficient manual efforts to compose access requests. In this paper, we propose an automatic XACML requests generation for testing access control policies by employing symbolic execution techniques. Firstly, the access control policy under test is converted into semantically equivalent C Code Representation (CCR). Secondly, the CCR is symbolically executed to generate test inputs. Finally, the test inputs are used to compose access control requests, which can be automatically evaluated with existing tools. We also implemented a prototype tool called XPTester (Xacml Policy Tester) and conducted extensive experiments upon real-world policies to demonstrate the scalability, efficiency and effectiveness.

Keywords—Access control policy; XACML; test generation; symbolic execution

I. INTRODUCTION

Strong access control is necessary to protect the sensitive resources for security and privacy. It is even more important in mobile computing. XACML [1] has become increasingly popular for access control specification and enforcement for protecting sensitive resources. And it is increasingly being adopted in mobile systems, especially in mission critical domains [2], [3], [4], [5], [6], [7]. The effectiveness of an access control system in protecting sensitive resources, however, relies on the correctness of access control policies. On the other hand, manually created access control policies, e.g., written in XACML, may be faulty and is prone to severe security consequences. According to the Internet Security Threat Report of 2013 by Symantec, the volume of web-based attack in 2012 has increased almost one-third of that in 2011. An early study by University of Pennsylvania ranks Broken Access Control in the second place of top 10 web application security vulnerabilities. Faulty access control configuration or specification that imports errors into access control policies is one of the most common vulnerabilities that lead to web-based attacks.

Therefore, XACML policies should be tested to ensure that they were specified as expected before they are implemented.

For testing an XACML policy, the testers need to send access requests to the policy and observe the access decisions. Every unexpected decision indicates that errors exist in the XACML policy. Test requests that cover the policy sufficiently are more effective to discover errors in the policy. However, it is tedious and inefficient to compose XACML requests manually. Therefore, it is attractive to apply automated testing approaches to test the XACML policies. There are several research work on testing XACML policies that are proposed recently [8], [9], [10], [11]. However, they are either not scalable or not effective enough. For example, Cirg [10] is based on change impact analysis, which is a high time-consuming process and its efficiency is related to the size of the policy. So Cirg suffers from scalability issues and its application to real-world policies is limited. X-CREATE simply takes all possible combinations of the attribute values in the policy to generate test requests. This will lead to a lot of redundant test requests thus reducing the test efficiency.

In this paper we propose an automatic, scalable and effective approach that employs symbolic execution techniques to generate access requests to test XACML access control policies. Because symbolic execution is able to generate high coverage tests for programs, we consider leveraging it to compose adequate test requests for access control policies. Firstly, the access control policy under test is converted into semantically equivalent C Code Representation (CCR). Secondly, the CCR is symbolically executed to generate test inputs. Finally, the test inputs are used to compose access control requests for testing. The results of the experiments on real-world XACML policies demonstrated that our approach is efficient and effective in aiding the testers to detect faults in the policies.

The contributions of this paper are as follows:

- We propose an automatic access request generation approach for testing access control policies by leveraging existing symbolic execution technique.
- We bridge non-executable XACML and executable code by using a code generation approach based on code schemas.
- We have implemented a supporting prototype tool, XPTester, and conducted controlled experiments using real-world access control policies.

The rest of the paper is organized as follows. Section II
introduces the background and provides a motivation example. Section III details the proposed approach. In Section IV we present the evaluation of our approach. In Section V we discuss the most related work, and we conclude and discuss the future work in Section VI.

II. BACKGROUND

A. XACML

XACML is an attribute based access control system and was proposed by OASIS in 2003. In XACML, attributes are associated with four elements, that is, Subject, Action, Resource and Environment. An XACML policy is composed of three elements: PolicySet, Policy and Rule. A PolicySet may contain a sequence of Policies or PolicySets and a Policy consists of a sequence of Rules. Each of the three elements consists of at most one Target, which specifies the constraint on Subject, Action, Resource and Environment to restrict the type of requests to which the PolicySet, Policy or Rule can be applied. The authorization decision to the access requests is defined and stored in every Rule as Effect, the possible value of which is Permit or Deny. In systems that adopt XACML, a component named PEP (Policy Enforcement Point) composes access requests in XACML format and sends them to another component called PDP (Policy Decision Point) for requests evaluation. The PDP makes authorization decision based on the XACML policy. Sometimes there will be more than one Policies or Rules can be applied to an access request, but only one authorization decision is allowed for every single request. Then in XACML, Combining Algorithm is designed to handle such a situation. There are Policy Combining Algorithm and Rule Combining Algorithm dealing with conflict of Policies and Rules, respectively. In this paper, we deal with the most common four of the twelve types of Combining Algorithm: First-Applicable, Permit-Overrides, Deny-Overrides and Only-One-Applicable.

B. Symbolic Execution

The concept of Symbolic Execution[12] was first put forward by JC King in 1976. It is proposed to facilitate the automated generation of high code coverage test cases. In symbolic execution, instead of supplying normal inputs to the program, the inputs are made symbolic to represent any values. Symbolic execution process is same as normal execution except that the values obtained during execution may be formulas on symbolic values. During symbolic execution, the executing path will fork at condition statement like if-else. Both branches will be executed and the corresponding constraints will be added to the current path constraint. At the end of an execution path, the path constraint on symbolic variables will be solved for concrete values. Thus a concrete input that can make this path be executed is obtained. After all paths are executed and the corresponding path constraints are solved (if solvable), a test suite is obtained, containing test inputs that is capable of covering the executed paths during symbolic execution. There are several tools supporting symbolic execution, i.e., KLEE [13]. KLEE is able to generate tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs, specifically for C language. For efficiency purpose, we chose KLEE in our framework for test case generation.

C. A Motivating Example

Here we use a motivating example concerning mobile security to further demonstrate the importance of testing the access control policies applied to the mobile devices. In recent years, in order to improve working efficiency of the employees as well as cutting costs, more and more companies allow their employees to bring their personal devices into the workplace and use them to do daily work. This practice is called Bring Your Own Device (BYOD) [14]. Thus the employees can use their personal devices to access the company’s privileged applications and information networks. At the same time the devices may also be connected to the Internet or have installed many other applications that may include malwares. On one hand, strong access control policies must be applied to protect the private resources of the company, i.e., to specify who can access what resources. On the other hand, before implementing the access control policy, testers should test the potentially complex policy thoroughly to prevent fault policy causing severe security consequences (e.g. confidential information leakage). This motivates us to investigate effective and efficient approaches to automatically test XACML policies.

III. APPROACH

In this section, we describe the proposed approach in details. The framework of our approach is illustrated in Figure 1. There are three steps in the framework for a complete testing process, each of which is represented by a rectangle in the figure. Firstly, the XACML policy under test is converted into semantically equivalent C Code Representation (CCR). Secondly, test inputs of the CCR are generated via symbolic execution and afterwards are used to compose XACML test requests. Finally, the test requests are evaluated against the XACML policy under test to generate a testing report.

### A. Generate C Code Representation

1) **Attribute Numeralization:** Symbolic execution is more capable of dealing with variables of integer type. However, the attribute values in XACML policies are in various types including string, URL, boolean and so on. Therefore, in order to take advantage of symbolic execution, XACML policy needs to be preprocessed by numeralizing the attributes appeared in the policy. Each attribute is assigned with a unique integer value to represent the attribute value [15]. We put each pair

---

3https://www.oasis-open.org/
of mapping between an attribute value and its corresponding integer value into an AIM (Attribute Integer Mapping) set. One example of AIM is listed in TABLE I.

<table>
<thead>
<tr>
<th>Attribute value</th>
<th>Attribute type</th>
<th>Integer value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technician</td>
<td>string</td>
<td>1</td>
</tr>
<tr>
<td>Manager</td>
<td>string</td>
<td>2</td>
</tr>
<tr>
<td>User</td>
<td>string</td>
<td>3</td>
</tr>
</tbody>
</table>

TABLE I: Example of AIM sets

2) Code Schema: Real-world XACML policies may be complex and large, which makes it challenging to effectively converting them to code representations. The core of CCR generation is the Code Schema, which is shown in Figure 2. It is designed to facilitate systematic and scalable CCR generation from XACML policies. The Code Schema specifies the structure of the CCR for a XACML policy. Precisely, PolicySets, Policies, and Rules are represented as functions in CCR according to the Code Schema. A CCR is composed of a sequence of the function definitions. The functions typically contain an if statement representing the evaluation of Target. The if statement is called Evaluation code block, which may contain calls to other functions for XACML elements. Based on the Code Schema, we can able to systematically generate CCR from XACML policies in any size or complexity.

![Fig. 2: Code Schema of C Code Representation (CCR)](image)

3) Mapping Rules from Combining Algorithms to Code: While it is easy to represent the policy with if-else statements, the greatest challenge to convert an XACML policy into CCR is the complex semantics of Combining Algorithms. There are four Combining Algorithms defined in XACML: First-Applicable, Deny-Overrides, Permit-Overrides and Only-One-Applicable. The different semantics of Combining Algorithms are reflected in CCR by the distinctions between Evaluation code blocks. Table II shows the mapping rules from Combining Algorithms to specific Evaluation code blocks. When converting the XACML policies, the specific Evaluation code block will be selected from the mapping rules according to the type of Combining Algorithm to compose the CCR.

4) Algorithm for CCR Generation: The CCR generation algorithm, which is shown in Algorithm 1, converts an XACML policy into its CCR. The algorithm traverses the XACML policy in DFS (Depth First Search) manner starting from the root (PolicySet or Policy). BuildPolicySet, BuildPolicy and BuildRule are functions that convert PolicySet, Policy and Rule into corresponding functions. In BuildPolicySet or BuildPolicy, it checks the type of Combining Algorithm defined in the PolicySet (Policy). Based on the type of Combining Algorithm detected, the algorithm chooses the proper Evaluation code block by referring to the mapping rules from XACML policy to Evaluation code blocks. It then fills the Evaluation code blocks with the integer values for the attribute values found in Target. Subsequently, the algorithm recursively converts the PolicySets, Policies or Rules contained in the PolicySet or Policy until arriving at the end of the policy.

B. Generate Test Requests via Symbolic Execution

For test generation, automation, adequacy, non-redundancy, and efficiency are mutually restraint factors. Symbolic execution techniques satisfy all above mentioned factors simultaneously. In our approach, we use a state-of-the-art symbolic execution tool KLEE [13] to explore every possible path of CCR to generate test inputs to cover adequate paths in an efficient manner. In the generated CCR, there are four integer variables (Sub, Act, Res, Env) defined at the start of the main function, representing Subject, Action, Resource and Environment, respectively. They should be made symbolic by using klee_make_symbolic. The CCR will be compiled into executable program using LLVM [16], and symbolically executed by KLEE to obtain test inputs for the CCR.

Symbolic execution generates a set of test cases containing inputs for CCR. Each test case contains values of four CCR inputs, which are Sub, Act, Res and Env. For each of these values, our framework queries the AIM set for the corresponding attribute values. Once the attribute values of Subject, Action, Resource and Environment are obtained, an XACML request can be constructed.

C. Evaluate the Test Requests

Our framework composes a set of test requests based on the test inputs generated from CCR, each of which is an access request. We use PDP to evaluate each test request to produce an authorize decision. We instrumented the PDP to record the evaluation trace (i.e., sequence of policy elements that are applicable to the request) into a testing report. Each decision will be compared with what is specified in the access control requirement specification (supplied by the tester). If the decision is inconsistent with the corresponding expected
Algorithm 1 Converting a XACML policy into CCR

Require:
- XP, the XACML policy to convert.
- A5, the AIM sets

Ensure: The CCR for the XACML policy.
1: CCR ccr = NULL;
2: function func = NULL;
3: for each PolicySet PS in XP do
4:   func = BuildPolicySet(PS, ccr);
5:   ccr.addFunction(func);
6: end for
7: for each Policy P in XP do
8:   func = BuildPolicy(P, ccr);
9:   ccr.addFunction(func);
10: end for
11: return ccr;
12: function BuildPolicySet(PolicySet PS, CCR ccr)
13:   query into AS to find integer values for the attributes;
14:   create if statement according to the Target of PS;
15:   CombiningAlgorithm CA;
16:   = PS.CombiningAlgorithm;
17:   create Evaluation code block according to CA;
18:   for each PolicySet ps in PS do
19:     = BuildPolicySet(ps, ccr);
20:       BuildPolicySet(func, func);
21:   end for
22: end function
23: return function for PS;
24: function BuildPolicy(Policy P, CCR ccr)
25:   query into AS to find integer values for the attributes;
26:   create if statement according to the Target of P;
27:   CombiningAlgorithm CA;
28:   = P.CombiningAlgorithm;
29:   create Evaluation code block according to CA;
30:   for each Rule r in P do
31:     = BuildRule(r);
32:       BuildRule(func, func);
33:   end for
34: return function for P;
35: function BuildRule(Rule R)
36:   query into AS to find integer values for the attributes;
37:   create if statement according to the Target of R;
38: return function for P;
39: end function

decision, there must exist a fault in the XACML policy. The request will be highlighted as a failed test in the generated report. By observing the evaluation trace of the test request, it is easy for the tester to locate the fault in the policy.

IV. IMPLEMENTATION AND EVALUATION

Based on the proposed approach, we implemented a prototype tool, XPTester, using Java version 1.7 under Linux (Ubuntu 12.10) with LLVM (version 2.9) and llvm-gcc (we used binary version for LLVM 2.9) [16], with KLEE [13] as the symbolic execution engine. In this section, we present the results of empirical experiments and give a comprehensive evaluation analysis.

Generally, there are two effectiveness measurements in evaluating an access control testing approach: the policy coverage ratios and the defect detection capabilities of the generated requests. There are also two other important aspects of efficiency that are often overlooked, number of test requests and time spent in generating them. The goal of this evaluation is to evaluate both the efficiency and effectiveness of our approach.

In the experiments, XPTester was compared with the latest XACML policy testing tool X-CREATE [8]. X-CREATE generates XACML requests by assigning attribute values to the intermediate-request obtained by exploring the XACML Context Schema. For each XACML policy, the number of requests X-CREATE can generate is variable, and it is the testers’ responsibility to decide how many requests to generate.

All our experiments were executed on a quad-core machine with an Intel Core (TM) i5-2400M 3.10GHz processor and 4GB memory size.

1) Experiment Subjects: TABLE III lists the subjects we tested in our experiment, including names of the policies, line of code (LOC) in the XACML file, numbers of type of Combining Algorithm(CA) appearing in the policies, Policy-Set, Policy and Rule. continue-a is used as experiment subject to evaluate Cirg [10], collected from a real-world conference management system. SyntheticPolicy-1 and SyntheticPolicy-2 are synthetic policies obtained from the benchmarks used by Xengine [15]. These two policies are huge in size and we used them to evaluate the effectiveness of XPTester to deal with large scale policies. WSO2 is a sample policy composed by WSO2, which is a leading company in service-oriented architecture solutions providers. demo-5, demo-11 and demo-26 are example policies used to illustrate Fedoras’ usage of XACML policy. Policy-EMC is an example policy presented by EMC[4]. IIIIC003Policy is one of the test cases for XACML 2.0 conformance tests[5].

2) Evaluation Results: We report the experiment results in Table IV. X-CREATE allows the users to decide how many requests to generate [8] while XPTester generate fixed amount of requests. For each of the experimental subjects, we used XPTester to generate one set of requests and X-CREATE to generate two, one of which has the same number of the requests generated by XPTester while the other contains all the requests X-CREATE can generate. We found that the number of requests generated by X-CREATE could be far more than that generated by XPTester, especially when the size of the policy is big. We summarize the evaluation results as follows.

Result 1: XPTester is more time efficient to generate even a large number of test requests than X-CREATE.

We define Time Consumed in Obtaining Requests (TCOR) of an access control testing framework as the total time spent in obtaining a certain number of requests. Intuitively, the smaller the TCOR is, the more efficient the

---

2) https://community.emc.com/docs/DOC-7410
framework is in generating test requests. We found that the time of XPTester consumed in generating test requests is almost one tenth of that of X-CREATE on average. This is because X-CREATE generates test requests by exploring the XACML Context Schema and taking the attribute value combinations, both of which are time consuming processes. Especially, for policies with large number of attribute values, such as continue-a, the time X-CREATE spends in generating test requests exploded.

**Result 2:** XPTester is able to generate less test requests while achieving the same policy coverage as X-CREATE.

Regarding the policy coverage of requests generated by XPTester and X-CREATE, the results show that X-CREATE achieves less Rule coverage than XPTester when generating the same number of requests. For an unbiased comparison, we also experimented to have X-CREATE achieve the same rule coverage as XPTester did by generating a larger set of requests. It is worth noting that XPTester generated far less number of requests than X-CREATE. X-CREATE achieves certain degree of policy coverage by simply taking combinations of all attribute values that may lead to redundant requests.

**Result 3:** XPTester is more capable of detecting faults in XACML policies than X-CREATE with less test requests.

We carried out mutant testing to evaluate the fault detection capability of test requests. We referred to the mutation operation proposed by Martin [17] and automatically generated mutant policies for each subject. We observed that the mutation killing ratios of requests from XPTester and X-CREATE are totally the same when using X-CREATE to generate the maximum number of requests it could. However, if we utilized X-CREATE to generate the same number of requests as XPTester, its mutant killing ratios declined significantly compared to XPTester. This is intuitively understandable as fault detection capability associates with policy coverage ratio.

Our experiment results confirmed that XPTester is efficient in generating test requests that have high fault detection ability for XACML policies. The efficiency and effectiveness of XPTester lie in following facts: First, XPTester is useful in reducing the time spent in requests generation. Second, the generated requests cover the policy adequately and have strong fault detection capability. In addition, XPTester generates relatively small number of requests while achieving the same policy coverage and fault detection capability comparing to the state-of-the-art tools, such as X-CREATE.

### V. Related Work

There are many research projects on deploying and testing access control for mobile devices. El-Aziz and Kannan propose a framework that uses XACML to support access control over the data access of handheld devices [2]. Ullhveit-Moe and Oleshchuk study mobile security with location-aware role-based access control by implementing access control policy using XACML [3]. Abdunabi et al. propose a new spatio-temporal role-based access control model that considers time of access and location of the mobile device users [4]. To avoid modifying the operating system, Android specifically, while enhancing a security policy, Xu et al. propose a method that is able to repackaged applications to enclose the policy to facilitate runtime monitoring of application behaviors [6]. An Android runtime security policy enforcement framework is proposed by Banuri et al. that validates the behavior of an application through its permission exercising patterns [18]. In terms of securing the scenario of Bring Your Own Device (BYDO), Rmando et al. propose a security framework for mobile devices to restrict the installation of applications to assure the security of the resources under protection [19]. These research demonstrates the need of access control for mobile security and XACML is becoming an important deployment candidate for policy specification and enforcement.

For security policy testing, Mouelhi et al. and Le Traon et al. propose a technique that transforms the test cases for functional testing into test cases for security policy testing [20], [21]. There are several techniques proposed and tools developed for XACML policy testing. Bertolino et al. propose a test strategy to generate XACML requests from the XACML Context Schema that leads to X-CREATE [8]. Martin and Xie develop Targen [9], in which the policy under test is converted into a tree structure representation. It generates XACML requests by combining all the possible combinations of attributes values satisfying the constraint along the tree’s paths. X-CREATE and Targen, however, generate too many redundant test requests due to the randomness nature of their request generation approaches. Another tool called Cigr...
also developed by Martin and Xie [10]. Cigl utilizes the change-impact analysis capability of Margrave [22] to generate XACML requests. However, change-impact’s efficiency depends on the size of the policy [10].

The most similar work compared with ours is proposed by Yu et al. [11], in which they also propose a security policy testing approach based on code generation. They utilize concolic testing [23] to generate test inputs for program module by first converting the policy under test into program module like JAVA code. The concolic testing tool they use is JCUTE [24]. After obtaining test inputs from JCUTE, the inputs are translated into access requests for testing purpose.

While both leverages code generation, Yu’s work has several limitations. First, the program module generation process fails to take into comprehensive consideration of the complex semantics of the policy language like various Functions defined in XACML. Second, as far as we know, JCUTE does not offer any programming interfaces thus the test input generation and translation must be done with manual effort, thus the approach cannot be completely automated. Finally, the tester has to audit the evaluation results of the test requests manually which is a tedious and error prone exercise.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose an automated testing approach that employs symbolic execution and could be used to test XACML access control policy. We creatively implemented an automatic method to convert the static access control policies into runnable C programs. Then by properly leveraging symbolic execution, our approach is able to efficiently and effectively generate sound test requests. The experiment results confirmed the applicability of our approach. Moreover, this approach is able to deal with large scale policies.

In the future, we will continue our work in three directions. First, we plan to extend our approach to support more access control policy languages. Second, the approaches for testing PEP or obligations are yet to be developed. Third, we will work on mobile security, for example, testing mobile apps running on mobile devices, whose access control policies are specified with XACML.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No.91118007, No.61328491,No.61170066), and by the National 863 High-Tech Program of China (No.2012AA011205).

REFERENCES