# XPA: An Open Soruce IDE for XACML Policies

Roshan Shrestha
*Department of Computer Science*
*Boise State University*
Boise, ID 83725, USA
roshanshrestha@boisestate.edu

Shuai Peng
*Department of Computer Science*
*Boise State University*
Boise, ID 83725, USA
shuaipeng@boisestate.edu

Turner Lehmbecker
*Department of Computer Science*
*Eastern Washington University*
Cheney, WA, USA
edmfrosty@gmail.com

Dianxiang Xu
*Department of Computer Science*
*Boise State University*
Boise, ID 83725, USA
dianxiangxu@boisestate.edu

***Abstract*—This paper presents XPA (XACML Policy Analyzer), an open source IDE (Integrated Development Environment) for testing, debugging, and mutating XACML 3.0 policies. XACML is an OASIS standard for specifying attribute-based access control policies. XPA provides a variety of new techniques for generating test cases from policies, localizing bugs in faulty policies, and repairing faulty policy elements. XPA has been applied to numerous XACML policies from the literature and real-world applications. These policies have been used to quantitatively evaluate the effectiveness of various testing and debugging methods. For system developers and administrators, XPA is a practical IDE for developing dependable XACML policies. For access control researchers, XPA offers a versatile toolkit for studying and evaluating new testing, debugging, and verification techniques.**

***Keywords—access control, XACML, testing, fault localization, debugging***

## I. Introduction

Attribute-Based Access Control (ABAC) is a new generation of access control techniques. It makes authorization decisions based on attributes of users, resources, actions, and environments [1]. Due to its fine granularity and high flexibility, ABAC is playing an increasing role in business and federal security domains. XACML (eXtensible Access Control Markup Language) is an OASIS standard for specifying ABAC policies in the XML format [2]. It has been integrated in major identity management products, such as Oracle Identity Manager and WSO2 Identity Server. Although these products allow user to edit and query XACML policies, there is a lack of tool support for policy testing, debugging, and evaluation.

The inherent complexity of real-world ABAC policies and the expressiveness of the XACML language indicate the likely existence of access control defects and the difficulty in finding them. The access control defects may result from omission or misunderstanding of access control requirements, unexpected interactions between security policy and business logic, and coding errors. These defects need to be uncovered and fixed before the system is deployed; otherwise they may lead to unauthorized access or denial of service. For quality assurance purposes, we argue that, similar to system and software development, policy development should follow a rigorous engineering process, including requirements analysis, design (e.g., decomposition and modularization), coding (e.g., in the XACML language), validation (e.g., testing and debugging), deployment and maintenance. Thus, an integrated development environment (IDE) is needed to provide computer-aided support for various activities in this engineering process.

This paper presents XPA (XACML Policy Analyzer), an evolving IDE for the development and implementation of dependable XACML policies. It consists of a variety of tools for editing, compiling, testing, debugging, and mutating XACML policies. The main features are: (1) coverage-based test generation using a constraint solver for XACML policies, (2) mutation-based test generation using a constraint solver for XACML policies, (3) coverage-based fault localization of XACML policies, and (4) mutation-based repair of XACML policies. The underlying technical approach of each feature implies substantial research effort and its elaboration requires a separate paper. The fault localization and repair methods for debugging XACML policies appeared in our previous work [3] [4], but their implementations have been improved for efficiency and user-friendliness. The policy mutator in XPA is currently the only one that supports XACML 3.0 and second-order mutation (i.e., application of two mutation operators). Other mutation tools for XACML [5][6] can only apply one mutation operator to XACML 1.0 and 2.0 policies.

The remainder of this paper is organized as follows: Section II gives a brief introduction to XACML policies. Section III presents the architecture of XPA, Section IV describes the mutation tool for XACML 3.0 policies. Sections V and VI present coverage-based and mutation-based test generators, respectively. Sections VII introduces fault localizer and policy repairer. Section VIII summarizes the evaluations of XPA. Section IX reviews and compares related work. Section X concludes this paper.

## II. XACML Policies

The first class entities in XACML are policy and policy set. A policy set consists of a policy set target, a policy-combining algorithm identifier, a list of policies or policy sets, an obligation expression, and an advice expression. Policy set target, obligation expression, and advice expression are optional. An obligation expression describes the string attached to the access privilege, whereas an advice expression describes an optional suggestion on the access. A policy comprises a policy target, a rule-combining algorithm identifier, a list of rules, an obligation expression, and an advice expression. A rule consists of a target, a condition, an effect (permit or deny), an obligation expression, and an advice expression. The rule target specifies the set of requests to which the rule is intended to apply. The rule condition refines the applicability of the rule established by the rule target. The target of a rule, policy, or policy set is a conjunctive sequence of AnyOf clauses. Each AnyOf clause is a disjunctive sequence of AllOf clauses, and each AllOf clause is a conjunctive sequence of match

This work was supported in part by US National Science Foundation (NSF) under grants CNS 1618229 and CNS 1461133.

DOI reference number: 10.18293/SEKE2018-027

predicates. A match predicate compares attribute values in an access request with the embedded attributes. Logical expressions for match predicates and rule conditions can apply a great variety of predefined functions and data types (such as string, Boolean, integer, double, time, and dates) to attributes. XACML provides four pre-defined categories of attributes: subject, resource, action, and environment. It also allows user to introduce additional attribute categories.

When an access request is fed to an XACML engine that is running a policy set or policy, the engine will return an access decision (permit, deny, not applicable, or indeterminate) per the policy set or policy. The decision may be attached with obligation or advice, depending on the policy or policy set. An access request consists of a list of attribute names, types, and values. In this paper, it is also call test input, specified in a text file. A complete test case is composed of both test input and expected access decision (i.e., oracle value). The oracle value for a test input is usually determined by the access control requirements of the system under development. When a policy set or policy is known to be correct (e.g., for experiment purposes), the actual response of the policy set or policy can be recorded and then used as the oracle value of the corresponding test input. In an evolving policy development process, the actual access decisions of test inputs from earlier policy versions can be recorded and then used as the oracle values of corresponding test inputs for testing the current or future versions if their correctness has been confirmed before. Given a test case for a policy set or policy, the actual response returned by the XACML engine depends on the evaluation of all policy elements. Consider a typical policy set with a list of policies, where each policy is composed of a list of rules. The final access decision per the policy set depends the evaluation results of the policy set target, access decisions of individual policies within the policy set, and the policy combining algorithm. The access decision of each individual policy depends on the evaluation results of the policy target, access decisions of individual rules in the policy, and the rule combining algorithm. XCAML3.0 provides 11 rule combining algorithms and 12 policy combining algorithms. The most commonly used combining algorithms are Deny-overrides, Permit-overrides, First-applicable, Deny-unless-permit, and Permit-unless-deny.

## III. THE ARCHITECTURE OF XPA

Figure 1 shows the architecture of XPA. The main components are: editor, test runner, fault localizer, policy repairer, policy mutator, mutation-based test generators, and coverage-based test generators. It is implemented in Java and AspectJ (an aspect-oriented extension to Java). The editor is adapted from the open source project UMU-XACML-Editor [7], which was originally developed for XACML 1.0 and 2.0. The XACML engine is Balana [8], the only open source implementation for XACML 3.0 when we started this project.

The test runner feeds a test suite to the XACML engine running a policy set or policy and reports the pass/fail result of each test. For a test case without an oracle value (expected response), the actual response is recorded. For a test case with an oracle value, the test runner also compares the oracle value with the actual response and makes a verdict of pass or fail. The failure of a test case indicates that the policy or policy set under test has one or more faults if the test input and the oracle value are both correct.

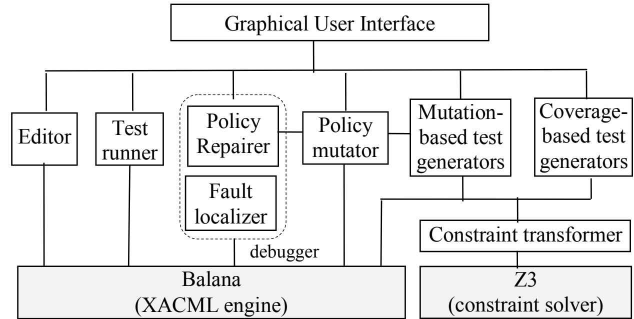


Fig. 1. Architecture of XPA

When there is a test failure, the fault localizer can be used to pinpoint the possible locations of faults (e.g., policy elements in the policy or policy set under test). It ranks all policy elements in the descending order of their suspicion scores calculated from the execution of the entire test suite. The user can then examine the top-ranked elements to determine whether they are faulty and how to fix them. Because Balana does not keep track of test execution information, we use an aspect-oriented instrumentation technique in AspectJ to monitor the evaluation result of each policy element when each test case is executed. This technique does not need to modify the source code of Balana.

The policy repairer takes a step further, aiming to repair a faulty policy automatically. It attempts to make a series of changes to the faulty policy, i.e., mutate the faulty policy, so as to make all test cases pass. The repair attempt may or may not be successful, depending on the faults. Note that automatic repair is a hard problem. To the best of our knowledge, our work is the first effort toward automatic repair of XACML policies and policy sets. The current repairer can fix a fault policy with no more than two simple faults. A simple fault is one that can be corrected by one mutation operation.

The policy mutator is a program that generates mutants of a given policy or policy set. Each mutant is a variation of the original policy or policy set. A first-order mutant is obtained by applying one mutation operator to make one change, whereas a second-order mutant is created by applying two mutation operators to make two changes. As the mutants of a correct policy contain different types of faults, they are commonly used to evaluate the effectiveness of a testing method, i.e., how many faults can be detected.

XPA also exploits policy mutation for test generation purposes. Given an original policy and its mutant, a mutation-based test generator produces an access request such that the two policies yield different responses. To do so, it first collects the constraint on attributes by comparing the two policy versions and then feeds the constraint to Z3 to find attribute values to satisfy the constraint, and converts the result into an access request. For a set of policy mutants, the mutation-based test generators can produce an optimal test suite that reveal all faulty mutants. This test suite can be used to test a policy or

policy set without knowing whether the policy or policy set is faulty. It can also be used to measure other testing methods. Z3 is an SMT (Satisfiability Modulo Theories) Solver from Microsoft Research [9]. It is worth pointing out that, although policy mutation is commonly used for evaluating testing methods in the literature, mutation-based test generation and mutation-based policy repair in our work are new.

Coverage-based test generators are a set of programs that generate a test suite from a given policy or policy set according to a chosen coverage criterion. The main coverage criteria are rule coverage, rule pair coverage, permit/deny rule pair coverage, decision coverage, MC/DC (modified-condition and decision coverage), non-error decision coverage, and non-error MC/DC. Each coverage-based test generator first collects the constraints on attributes according to the chosen coverage criterion, feeds each constraint to Z3, and converts the result into an access request. The coverage-based test generators focus on the extent to which the policy elements are exercised by tests, whereas the mutation-based test generators aim to produce tests that can reveal hypothesized faults. Both are useful for quality assurance of XACML policies.

## IV. Policy Mutator

The policy mutator creates mutants of a policy set or policy by applying mutation operators to the policy set or policy. Mutation operators are defined with respect to a fault model, which represents a comprehensive set of fault types in XACML. Table I shows the fault model (i.e., column 1) and mutation operators for each fault type. Application of one mutation operator may result in a number of mutants. For example, the rule combining algorithm of a policy can be changed to any of the other rule combining algorithms.

TABLE I. Fault Model and Mutation Operators

| Fault type | Mutation operator | |
|---|---|---|
| | Name | Mutation |
| Incorrect policy/ policy set target | PTT | set Policy/set Target True |
| | PTF | set Policy/set Target False |
| Incorrect rule/policy combining algorithm | CRC | Change Rule/Policy Combining algorithm |
| Incorrect rule effect | CRE | Change Rule Effect |
| Incorrect rule target | RTT | set Rule Target True |
| | RTF | set Rule Target False |
| Incorrect rule condition | RCT | set Rule Condition True |
| | RCF | set Rule Condition False |
| | ANF | Add Not Function in condition |
| | RNF | Remove Not Function in condition |
| Incorrect rule ordering | FPR | First Permit Rules |
| | FDR | First Deny Rules |
| Missing rule | RER | REmove a Rule |
| Missing target element | RPTE | Remove Parallel Target Element |

Mutation operators in Table I are named with respect to correct policy sets and policies. Mutants of a correct policy set or policy may or may not contain faults. It is possible that a mutant is functionally equivalent to its original version, i.e., they always yield the same access decision for any access request. Mutants of a correct policy set or policy are commonly used for evaluating the fault detection capability of a testing method in term of mutation score. A mutant is said to be killed if a failure is reported by any test case produced by the testing method. A mutant that is not killed may be equivalent to the original policy. Given a test suite produced by a testing method, its mutation score or mutant-killing ratio is as follows:

$$\frac{\textit{number of killed mutants}}{\textit{total number of mutants} - \textit{number of equivalent mutants}}$$

Note that mutation operators can be applied to a policy set or policy no matter whether the policy set or policy is known to be correct or faulty. In particular, XPA applies mutation to test generation (Section VI) and policy repair (Section VII). In these cases, the fault types in Table I do not represent the meanings of mutation operators.

## V. Coverage-Based Test Generators

The coverage-based test generators produce access requests from a given policy set or policy to satisfy a chosen coverage criterion. As policies are special cases of policy sets, we describe the coverage criteria with respect to policy sets.

***Rule coverage***: A test suite for a policy set is said to satisfy rule coverage of the policy set if, for each rule in each policy of the policy set, there is as least one test in the test suite that evaluates the rule to its specified effect (permit or deny).

***Decision coverage*:** A test suite for a policy set is said to satisfy decision coverage of the policy set if the test suite covers all three decisions (true, false, error) of each decision expression, including the policy set target, the target of each policy, the target and condition of each rule in each policy.

***Non-error decision coverage*:** A test suite for a policy set is said to satisfy non-error decision coverage of the policy set if the test suite covers all non-error decisions (true and false) of each decision expression, including the policy set target, the policy target of each policy, the rule target and condition of each rule in each policy.

***MC/DC*:** A test suite for a policy set is said to satisfy MC/DC of the policy set if the test suite satisfies MC/DC and covers the error condition of each decision expression, including the policy set target, the policy target of each policy, the rule target and condition of each rule in each policy.

***Non-error MC/DC*:** A test suite for a policy set is said to satisfy MC/DC of the policy set if the test suite satisfies MC/DC of each decision expression, including the policy set target, the policy target of each policy, the rule target and condition of each rule in each policy.

***Rule pair coverage***: A test suite for a policy set is said to satisfy rule pair coverage of the policy set if, for each pair of rules within each policy, the test suite has a test to make both rules evaluate to their specified effects.

The above coverage criteria can also be used to measure the coverage adequacy of a given test suite. Such a test suite may be produced by other testing methods when a policy is developed or represent actual access requests in operation.

Generally speaking, test suites of different coverage criteria have different levels of fault detection capabilities. The measurement of coverage adequacy provides important guidelines for the development of access control tests.

## VI. Mutation-Based Test Generators

Given a policy set or policy whose correctness is unknown, mutation-based test generators create access requests by comparing the policy set or policy with each of its mutants (i.e., a hypothesized fault). The mutants are obtained by applying the mutation operators in Table I. A mutation-based test generator with respect to a mutation operator tries to generate one access request for each mutant obtained by the mutation operator. For such an access request, the original version and the mutant will respond with different access decisions. Assuming that one version is correct and the other version is faulty, the idea of mutation-based test generation relies on the following fault detection conditions: (1) ***Reachability condition***: the access request must reach the mutated policy element, such as rule target, rule condition, rule effect, policy target, policy set target, and rule/policy combining algorithm. (2) ***Necessity condition***: the access request must make the mutated element and the corresponding element in the original version evaluate to different intermediate results; (3) ***Propagation condition***: the access request must make the mutant and the original produce different responses. Propagation condition largely depends on the rule and policy combining algorithms.

By comparing the two policy versions, the mutation-based test generator derives a constraint that is composed of all three conditions. Then it feeds the constraint to Z3. If the constraint is solved, the solution is converted into an access request; otherwise the two policy versions are considered to be equivalent, assuming Z3 is sound and complete.

The key challenge of mutation-based test generation is the formalization of reachability condition, necessity condition, and propagation condition for each kind of mutants. The idea originated from fault-based testing or constraint-based testing in the software testing community. However, practical mutation-based test generators for software remain to be seen unless for toy examples – it is difficult, if not impossible, to formulate the fault detection conditions because of the inherent complexity of software. Due to the special structure of XACML policy sets and policies, we have been able to automatically derive complete fault detection conditions of all mutants. The details will be described in a separate paper.

## VII. Automated Debuger

The automated debugger consists of the fault localizer and the mutation-based policy repairer. Fault localizer aims to identify which element of a policy set or policy is likely faulty if there is a failure when it is tested with a test suite. The basic idea is to build a correlation between the evaluation result (firing or not) of each policy element and the test verdict (pass or fail) for each test case. The correlation data is then used to rank all policy elements with a certain scoring method. A policy element with a high suspicion score has a high probability of having fault(s). XPA has implemented 14 scoring methods selected from the best-performing spectrum-based methods for software fault localization [10].

The policy repairer takes a step further to modify the policy set or policy so that no test in the test suite will fail. According to the suspicion rankings from the fault localizer, the repairer starts with the most suspicious policy element, mutates it to create a new policy set or policy, and runs the new policy set or policy to check if all tests pass. If there is no failure, the repair is successful; otherwise the repairer will try another mutation or another suspicious element. Because a policy set or policy may have a number of faults, the repairer exploits the notion of plausible fix. A plausible fix does not make all tests pass. Instead, it makes the debugging progressive, indicated by a decreased number of failed tests. The repairer allows user to set up the depth of mutation for repair, a scoring method for sorting suspicious elements, and select some or all of the mutation operators. If the repair attempt is successful, XPA presents the relevant policy elements of both original and repaired versions.

## VIII. Evaluation and Application

We have applied a number of XACML policies to XPA, as listed in Table II. All of them are available at the project website. Three policies, *continue*, *fedora*, and *itrust*, were obtained from the literature. They were originally coded in 1.0 or 2.0. We upgraded them to 3.0 without changing their semantics. We also created three variations of *itrust* (*itrust5*, *itrust10*, and *itrust20*) for studying scalability of testing and debugging methods. *itrustX* has *X* times as many rules as *itrust*. The new rules are created by replicating original rules with new attribute values. *HL7* is a real world policy set provided by an XACML developer. The system that uses the HL7 policy set is not available, though. GPMS (Grant Proposal Management System) is an open source Java project that we have developed as an exemplar application of XACML. The motivation behind GPMS was that there is no real-world XACML3.0 application whose policy files and application source code are publicly available. GPMS is a web-based application for an academic institution to manage the internal workflow for grant submissions. It uses XACML to implement a fine-grained access control of the workflow.

TABLE II. Sample XACML3.0 policies

| Policy | # of rules | # of lines of the XACML file |
|---|---|---|
| continue | 15 | 229 |
| fedora[1] | 12 | 227 |
| itrust[2] | 64 | 1,283 |
| itrust5 | 320 | 6,403 |
| itrust10 | 640 | 12,803 |
| itrust20 | 1,280 | 25,603 |
| HL7 | 19 | 809 |
| GPMS policy | 97 | 7,678 |

***Evaluation of the coverage-based test generators***: Six policies (*continue*, *fedora*, *itrust*, *itrust5*, *itrust10*, and *itrust20*)

[1] http://www.fedora.info

[2] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start

have been used to evaluate all coverage-based test generators. As they are considered to be the correct version, the oracle value of each test input is the actual response from the original policy. The fault detection capability of each test generator is assessed through mutation testing, where both first-order and second-order mutants were generated by the policy mutator. The results show that the MC/DC test suite has the highest mutation score, whereas the rule coverage test suite was only able to kill about 50% of the mutants. All test generators have acceptable time performance for all policies.

***Evaluation of the mutation-based test generators***: All policies in Table II have been applied to the mutation-based test generators. They are able to generate a test input for every non-equivalent first-order mutants.

***Evaluation of the fault localizer***: All policies except *itrust20* and the GPMS policy in Table II have been applied to the fault localizer. The first-order and second-order mutants of each policy are used as inputs to the fault localizer. The experiments show that the 14 scoring methods have varying accuracy. *Naish2* and *CBI-Inc* can accurately localize the faults regardless of the policy size. The actual faulty policy element is usually among a few top candidates that are suggested by the *Naish2* and *CBI-Inc* methods.

***Evaluation of the policy repairer***: All policies except *itrust20* and the GPMS policy in Table II have been applied to the policy repair. Both first-order and second-order mutants of each policy are used as inputs. All scoring methods were able to repair them. This indicates that the mutation operators for policy mutation and the mutation operators for policy repair are reversible. The *Naish2* and *CBI-Inc* methods have the best time performance to locate the faulty elements.

***Application to GPMS***: XPA was used to test the GPMS policy in the development process of GPMS. The mutants of the GPMS policy is currently being used to evaluate the fault detection capability of a model-based test method for GPMS.

## IX. Related Work

Several methods have been proposed to generate test inputs for XACML policies: Cirg [11] generates access requests from counterexamples produced by the change-impact analysis of two synthesized versions. The difference of the two versions of a policy targets a test coverage goal (e.g., rule, or condition). Because access requests are encoded in XML, they must conform to the XML Context Schema. Bertolino et al., have developed different test generation algorithms by considering the structures of the Context Schema, such as Preliminary XPT and Incremental XPT [12]. Li et al. [8] used symbolic execution technique to generate access requests by converting the XACML policy under test into semantically equivalent C Code Representation (CCR) and symbolically executing CCR to create test inputs and translating the test inputs to access requests. The coverage-based test generators in XPA are different from the above work except for the rule coverage. In addition to the new coverage criteria, XPA generates access requests for exercising error conditions. Policy mutation has been used to evaluate the above testing methods, but limited to 1.0 and 2.0 [5][6]. XPA also uses policy mutation for test generation and policy repair.

## X. Conclusions

We have presented a comprehensive toolkit, XPA, for editing, testing, debugging, and mutating XACML policies. It also provides an infrastructure for experimentation with new testing and debugging methods. For example, when mutation is used to evaluate a new testing method against a policy, XPA can apply the test suite to all mutants of all or selected mutation operators and produce a summary of test execution results.

Our future work will focus on tool support for access control requirements analysis and policy maintenance in the policy engineering process. We plan to develop a computer-aided approach for transforming access control requirements specification in a natural language (e.g., English) into XACML policies. We will also implement various refactoring methods to facilitate changes of XACML policies.

## References


[1] V. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, “Guide to attribute based access control (ABAC) definition and considerations,” NIST Special Publication 800-162, October 2013.

[2] OASIS, “eXtensible Access Control Markup Language (XACML) Version 3.0,” http://www.oasisopen.org/committees/xacml/, Jan. 2013.

[3] D. Xu, Z. Wang, S. Peng, N. Shen, “Automated fault localization of XACML policies,” Proc. of the 21st ACM Symp. on Access Control Models and Technologies (SACMAT’16), pp. 137-147, June 2016.

[4] D. Xu and S. Peng, “Towards automatic repair of access control policies,” Proc. of the 14th IEEE Conference on Privacy, Security and Trust (PST’16), pp. 485-492, Auckland, New Zealand, December 2016.

[5] E. Martin, and T. Xie, “A fault model and mutation testing of access control policies,” Proc. of WWW’07, pp. 667-676, May 2007.

[6] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, “XACMUT: XACML 2.0 mutants generator,” Proc. of 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 28-33, 2013.

[7] P. G. Morcillo, A. J. Lázaro, G. Tormo UMU-XACML-Editor. http://umu-xacmleditor.sourceforge.net/

[8] WSO2. “Balana: An open source XACML 3.0 implementation.” http://xacmlinfo.org/2012/08/16/balana-the-open-source-xacml-3-0-implementation/

[9] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” Proc. of the 14th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), LNCS volume 4963. 2008, Springer.

[10] X. Xie, T. Y. Chen, F. Kuo, and B. Xu. “A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization,” ACM Trans. on Software Engineering and Methodology (TOSEM), 22(4):31.

[11] E. Martin, and T.Xie. "Automated test generation for access control policies via change-impact analysis." Proceedings of the Third International Workshop on Software Engineering for Secure Systems. IEEE Computer Society, 2007, pp.5-11.

[12] A. Bertolino, S. Daoudagh, F. Lonetti, and E.marchetti. "The X-CREATE Framework-A Comparison of XACML Policy Testing Strategies." Proc. of the 8th International Conference on Web Information Systems and Technologies (WEBIST). pp.155-160.