A Fault Model for Testing the Access Control Policies using Classified Mutation Operator

Suraj Sharma[1] and Sanjay Kumar Jena[2]
[1]Computer Science & Engineering Department, National Institute of Technology Rourkela, Orissa, INDIA
[2]Computer Science & Engineering Department, National Institute of Technology Rourkela, Orissa, INDIA
[1]suraj.sine@gmail.com
[2]skjena@nitrkl.com

ABSTRACT

In today's scenario any multiuser system need to implement access control for protecting its resources from unauthorized access or damage. With the help of separate policy specification language we can specify these access control policies. However, it is challenging to specify a correct access control policy and so, it is common for the security of a system to be compromised because of the incorrect specification of these policies. There are many ways in which a policy can be checked for correctness like, formal verification, analysis and testing. In this paper, a testing framework called ACPC (Access Control Policy Checker) has been introduced; we choose to illustrate the above technique using XACML language. We conduct extensive experiments using nine policy sets to evaluate the effectiveness of the above technique. The experimental result shows that ACPC can effectively generate requests to achieve high structural coverage of policies and outperforms random requests generation in terms of policy structural coverage and fault-detection capability. We have used nine mutation operators to make the mutant policy for mutation testing. We found the better result by classify these mutation operator in to three classes. We got up to 98% of mutant killed by one class of mutation operator, these results shows that, above framework generates better request sets and the classification gives better performance in terms of computational cost.

Keywords

Access control policies, change-impact analysis, mutation operator, mutation testing, XACML

1. INTRODUCTION

Any multiuser system need to implement access control for protecting its resources from unauthorized access or damage. Access control [5] is one of the fundamental mechanisms for information system security and it is widely used in operating systems, databases, networks, etc. All these systems support different applications with multiple users and every activity performed by a user or a process must be checked to see if it is authorized. An access control system determines what principals can access what resources and when.

Access control is traditionally enforced by directly hard coding into a system. However, this is tedious and becomes difficult for a large system. Also, this makes it hard to accommodate changes of security requirements in a system.

Recently, access control system increasingly separate policy from mechanisms. That is, an access control policy is explicitly specified using certain policy languages. And a system dynamically consults the policy to determine whether an access request should be granted. The advantage of this is that by separating policy from mechanism makes it easier to specify the protection requirements to be enforced on the system independent of the underlying implementation details. Also, when the security requirements on the system change later on, it is possible to easily change the policy without affecting the underlying mechanism implementing it. The ACPC is designed for checking these access control policies specified for the access control systems. The ACPC has two section; one for generating request by changeimpact analysis tool and second for testing these policies with the generated request sets.

The rest of the paper is organized as follows: Section 2 presents some related work in this area. Section 3 describes the proposed framework. Simulation Results and comparison with the existing method is in Section 4.

2. RELATED WORK

There are various ways in which the quality of the policy can be assured like, formal verification, analysis and testing. Formal verification techniques can verify if a policy satisfies a particular security property [11]. However, a formal representation of the policy is not scalable and properties about a policy do not exist in practice. Analysis of policies can include semantic analysis like performing a change impact analysis between two policies [2]. Testing is one practical way for checking the correctness of a policy specification. Semantic analysis techniques can be used complementary to testing. Techniques [10] have been proposed to leverage mutation testing to automatically generation and/or reduce test sets for general purpose programming languages. L. J. Morell [8] gives the brief discussion in fault based testing for software that is used frequently now a day. In [9] Martin and Xie have discussed about test generation via Change –Impact Analysis.

Martin, Xie, and Yu [3] have developed a random test generation tool for XACML policies. The tests (requests) are generated as a set of all combination of attributes found in the policy. The tool represents this attribute as a bit vector and an attribute appears in the request only if the corresponding bit in the vector is set to 1. The number of requests to be generated can be user specified. To achieve adequate coverage, even in a small request set, they modify the random bit setting algorithm to ensure each bit is set at least once. This method, though simple to implement is not ensure that a policy is thoroughly tested.

In our approach to policy testing, we generate policy requests by the Chang-Impact analysis tool margrave [2] and mutation testing method for testing the access control policies.

3. PROPOSED FRAMEWORK

ACPC (Access Control Policy Checker) is the proposed model for testing the correctness of Access Control policies. This model will work for the policies written in XACML policy specification language and having two sections. First section generates the request sets and the testing performs in second section.

Figure 1 shows the testing framework called ACPC (Access Control Policy Checker) for testing the correctness of policy. The input to the framework is the access control policy that is to be tested. In the request generation process, this policy is converted into derived policies and performing the Change-Impact analysis. The output of this phase is the request sets. In the policy checker phase, the input is request sets, generated by request generation phase and the policy for testing. In the policy checker phase mutation operators is used for producing mutant (faulty) policy and compare the response of mutation policy and original policy against the same request. If the response is different we say mutant has been killed otherwise not killed. Higher the mutant killing rate higher the correctness of the policy under test, so we can say:

$$\boxed{\text{Mutant Killing rate(mk rate) } \alpha \text{ Correctness of the policy}}$$

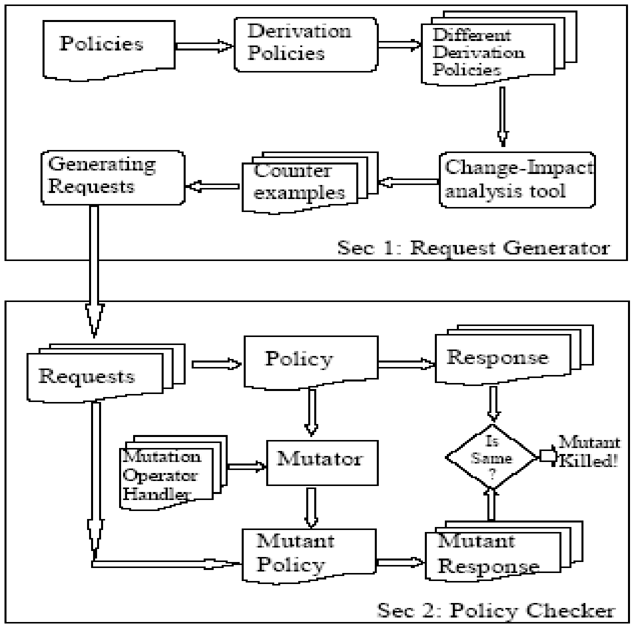

Figure 1: ACPC (Access Control Policy Checker) Model

3.1 Request Generation Process

To automatically generate high-quality test suites for access control policies, we develop a novel framework based on change-impact analysis [2]. We have referred different paper regarding change-impact analysis for developing Section 1. Figure 2 shows the Section 1 of the proposed framework. The framework receives a set of policies under test and outputs a set of request for policy authors to inspect for correctness. The framework consists of three major components: Derivation, Change-Impact analysis and Request Generation. The key notion of the framework is to derived 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 derived versions. A change-impact analysis tool can then be leveraged to generate counterexamples to witness these differences. Based on the generated counterexamples, the framework generates the request sets.

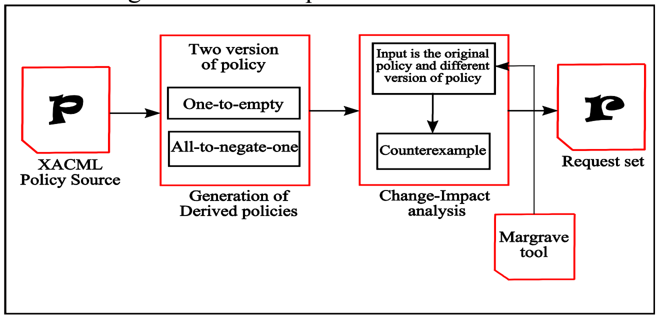

Figure 2: Request Generation Process Framework

3.1.1 Derivation

Given the policy under test, the derivation component derives the policy's versions, which are later fed to a changeimpact 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 below called one-to-empty and all-tonegate-one.

```
<Policy Id="demo" RuleCombAlgId="first-applicable">
<Rule RuleId="1" Effect="Deny">
<Target>
 <Subjects> <AnySubjects /> </Subjects>
 <Resources>
     <Resource>
          <ResourceMatch MatchId="equal">
          <AttrValue>demo:5</AttrValue>
          <ResourceAttrDesignator AttrId="objectid" />
          </ResourceMatch>
     </Resource>
  </Resources>
  <Actions>
     <Action>
          <ActionMatch MatchId="equal">
          <AttrValue>dissemination</AttrValue>
          <ActionAttrDesignator AttrId="actionid" />
          </ActionMatch>
     </Action>
  </Actions>
</Target>
</Rule>
<Rule RuleId="2" Effect="Permit" />
</Policy>
```

Figure 3: An example XACML policy

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 derivation variants with the case of a single policy p that contains n rules.

To further illustrate the framework, we provide a concrete example of XACML [12] policy in Figure 3. An XACML policy encodes rules in 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).

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 3. 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-negateone 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 3. 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.

3.1.2 Change-impact analysis

By supplying two versions of a policy, change-impact analysis tool i.e. Margrave's API [2,7] 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 request sets by iteratively manipulating the inputs to a change-impact analysis tool.

3.1.3 Request Generator

The counterexample generated by Margrave tool is in the form of 0 and 1 with respect to corresponding attribute values. The request generator generates the request with the corresponding attributes whose value is 1 in the counterexample. Exactly one request is generated from each counterexample.

3.2 Policy Testing Process

Mutation testing [8] has historically been applied to general purpose programming languages. The program under test is iteratively mutated to produce numerous mutants, each containing one fault. A test input is independently executed on the original program and each mutant program. If the output of a test executed on a mutant differs from the output of the same test executed on the original program, then the fault is detected and the mutant is said to be killed.

An overview of our Section 2 of the framework for policy mutation testing is illustrated in Figure 4.

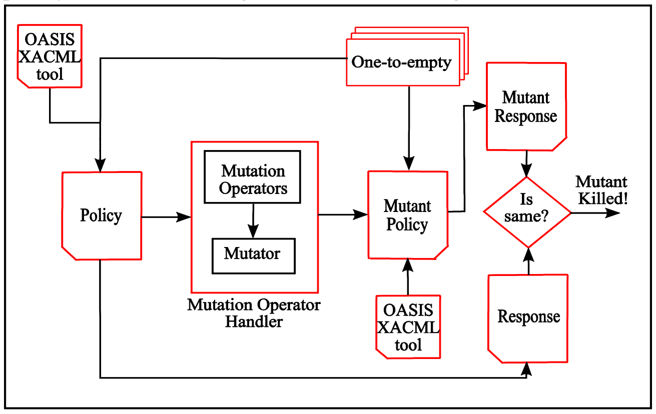

Figure 4: Policy Mutation Testing Framework

In the framework, we first define a set of mutation operators. Given a policy and a set of mutation operators, a mutator generates a number of mutant policies. Given a request set, we evaluate each request in the request set on both the original policy and a mutant policy. The request evaluation produces two responses for the request based on the original policy and the mutant policy, respectively. If these two responses are different, then we determine that the mutant is killed by the request; otherwise, the mutant is not killed. This framework also consists of three major components: Mutation Operator Handler, Mutant and Original Policy Testing, Difference Checker.

3.2.1 Mutation Operator Handler

Mutation operators describe modification rules for modifying access control policies to introduce faults into the policies. Previous studies [1] have been conducted to investigate the types and effectiveness of various mutation operators for general-purpose programming languages; however, these mutation operators often do not directly apply to mutating policies. This section describes the chosen mutation operators for XACML policies that implement our fault model. An index of the mutation operators is listed in Table 1 and their details are described below. The first eight mutation operators emulate syntactic faults because these mutation operators manipulate the predicates found in the target and condition elements. In particular, PSTT, PSTF, PTT, PTF, RTT, RTF, RCT and RCF emulate syntactic faults as simple typos in the policy set, policy, and rule target elements as well as the condition elements which result in the predicates found in those elements to always evaluate to true or false.

Table 1: Index of Mutation Operators

ID	Description
PSTT	The policy set is applied to all requests.
PSTF	The policy set is not applied to any requests.
PTT	The policy is applied to all requests.
PTF	The policy is not applied to any requests.
RTT	The rule is applied to all requests.
RTF	The rule is not applied to any requests.
RCT	The condition always evaluates to true.
RCF	The condition always evaluated to false.
CRE	The rule effect is inverted (e.g. permit for deny).

The last mutation operator CRE, emulate semantic faults because they manipulate the logical constructs of XACML policies.

Policy **S**et **T**arget **T**rue (PSTT): Ensure that the policy set is applied to all requests by removing the <Target> tag of each PolicySet element.

Policy **S**et **T**arget **F**alse (PSTT): Ensure that the policy set is never applied to a request by modifying the <Target> tag of each PolicySet element. The number of mutants created by this operator is equal to the number of PolicySet elements.

Policy **T**arget **T**rue (PTT): Ensure that the policy is applied to all requests simply by removing the <Target> tag of each Policy element.

Policy **T**arget **F**alse (PTF): Ensure that the policy is never applied to a request by modifying the <Target> tag of each Policy element.

Rule **T**arget **T**rue (RTT): Ensure that the rule is applied to all requests simply by removing the <Target> tag of each Rule element.

Rule **T**arget **F**alse (RTF): Ensure that the rule is never applied to a request by modifying the <Target> tag of each Rule element.

Rule **C**ondition **T**rue (RCT): Ensure that the condition always evaluates to True simply by removing the condition of each Rule element. **R**ule **C**ondition **F**alse (RCF): Ensure that the condition always evaluates to False by manipulating the condition value or the condition function.

Change **R**ule **E**ffect (CRE): Invert each rule's Effect by changing Permit to Deny or Deny to Permit. The number of mutants created by this operator is equal to the number of rules in the policy. This operator should never create equivalent mutants unless a rule is unreachable, a strong indication of an error in the policy specification.

These operators will pass in to Mutant module which take the original policy and convert it into mutant policy.

3.2.2 Mutant and Original Policy Testing

After getting the mutant policy generated by Mutation Operator Handler, we need the response against all the requests generated by Request Generation phase. Therefore, we need the OASIS XACML tool for verification and getting the response. The working principle of OASIS XACML tool is; it gives the response when we provide policy and request as input to the tool the output of the OASIS XACML tool is the response in XACML language. We use this principle for verifying the Mutant policy and Original policy by supplying the same request set for getting the response. This is very simple process and it worked just like Access Control System.

3.2.3 Difference Checker

The work of difference checker is very simple; it only compares the response of the Mutant policies and Original policy against same request set. It returns 1 if both the response is different otherwise return 0. The mutant killing rate is calculated by number of 1, if the number of 1 is more we say that the mutant killing rate is more. We can formulate the mutant killing rate as;

$$\boxed{\text{Mutant Killing rate(mk rate)} = \frac{\#\text{Mutant}}{\#\text{Mutant Policies}}}$$

The mk rate is easily calculated with the help of above formula and by multiplying by 100 we can find the mutant killing percentage (i.e. mk%).

4. SIMULATION RESULTS

We used nine XACML policies collected from three different sources as subjects in our experiment. Table 2 summarizes the basic statistics of each policy. The first column shows the subject names. Columns 2-5 show the numbers of policy sets, policies, rules, and conditions, respectively. Four of the policies, namely codeA, codeB, codeC, and codeD are examples used by Fisler et al. The remaining policies are examples of real XACML policies used by Fedora. Fedora is open source software that gives organizations a flexible service-oriented architecture for managing and delivering digital content.

Table 2: List of Policies used in the Experiment

Policy	#Policy Set	#Policy	#Rule	#Condition
codeA	5	2	2	0
codeB	11	5	5	0
codeC	8	4	4	0
codeD	11	5	5	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 1	2	12	10

As we mentioned in the beginning the result is in the form of comparison between the proposed approach (change-impact method) and the random method of generation of request. After that we proposed a classification of mutation operator based on iterative experiments which gives some better result. All this we will see one by one further.

4.1 Fault Detection Capability Comparison

Table 3 summarizes the two approaches of request generation method by its mutant killing ability. Column 2 shows the number of mutant policy generated by the Mutation Operator Handler. Columns 3-4 shows the mutant killed and mutant killed% by request set generated by random method, similarly columns 5-6 shows the mutant killed and mutant killed% by request set generated by change-impact analysis method.

Table 3: Mutant-kill result achieved by both methods

		Random Method		**Change-Impact analysis Method**	
XACML Policy	**# Mutant**	**# Mutant Kill**	**Mutant kill %**	**# Mutant Kill**	**Mutant kill %**
codeA	64	20	31.25%	29	45.31%
codeB	92	33	35.87%	42	45.65%
codeC	112	50	44.64%	53	47.32%
codeD	148	55	37.16%	69	46.62%
default 2	157	10	6.37%	85	54.14%
demo -11	22	16	72.73%	16	72.73%
demo -26	17	09	52.94%	09	52.94%
demo-5	23	17	73.91%	19	82.61%
mod-fedora	157	35	22.29%	82	52.23%
average			41.90%		55.50%

It observed from the table that, the average mutant killing% of change-impact analysis method is greater than the random method. Some time the request generated by random method is same as the change-impact analysis method, in the row of demo-11 and demo-26 the killing% is same. But in the case of default-2 there is the huge change, it varies from 6.37% to 54.14%. We can say that our method works well in all the cases and gives good result compare to random method. The graph between sample XACML policies and mutant kill percentage by both methods are shown in Figure 5, the difference between random and change-impact method is clearly visible in the graph.

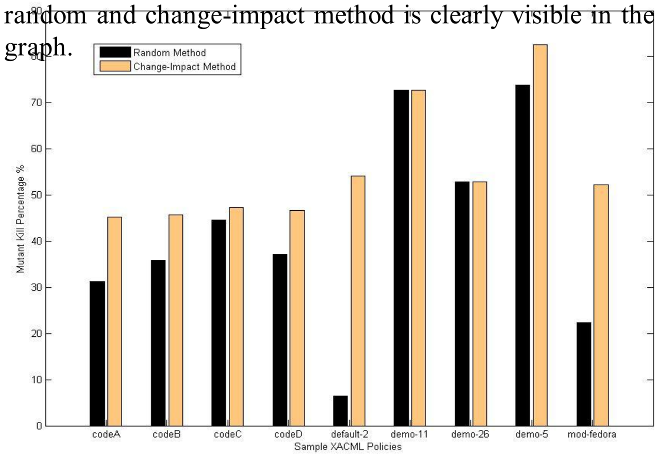

Figure 5: Comparison between Random and Change-Impact
method

4.2 Fault Detection by Individual Mutation Operators

Figure 6 shows the mutant kill % with respect to each mutation operator for all nine sample policies. In this result we found the variation in proposed method (Change-impact method) and random method. Now by doing the further experiment we found some better result after classify the mutation operator, describe in the further section.

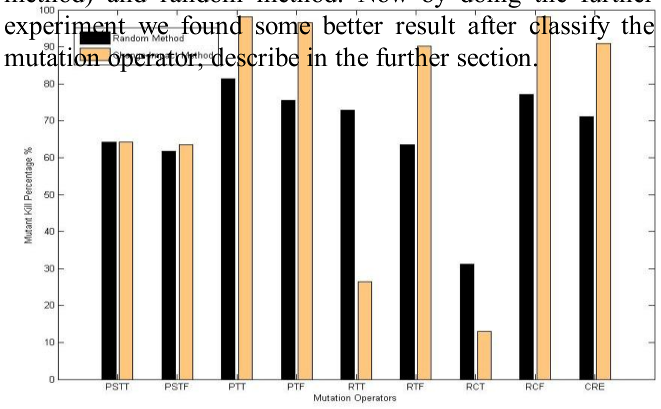

Figure 6: Fault detection of all policies by individual mutation operator

4.3 Classified Mutation Operators

The mutation operators can be classified based on the policy element on which the mutation operation is performed. They can be classified as, *Policy Set Mutation Operators:* These represent the mutation operations that can be done at the policy set level.

ID	Description
PTT	Policy Target True
PTF	Policy Target False

Policy Mutation Operators: These represent the mutation operations that can be done at the policy level.

ID	Description
PSTT	Policy set Target True
PSTF	Policy Set Target False

Rule Mutation Operators: These represent the mutation operations that can be done at the rule level.

ID	Description
PTT	Policy Target True
PTF	Policy Target False
RCT	Rule Condition True
RCF	Rule Condition False
CRE	Change Rule Effect

The result is in the form of graph shows Figure 7, the Rule mutation operator gives higher mutant killing percentage than Policy Set mutation operator and Rule mutation operator. The conclusion is that the mutation operator that we have chosen, that is the Rule mutation operator, works well and with the help of these five mutation operator we can perform the policy testing, which gives the approximate same result as with the combined mutation operator, but take less computational cost. In the graph we get up to 98.14% of mutant killed by the Rule mutation operator. These entire mutant kill percentages are for the request sets generated by the proposed method.

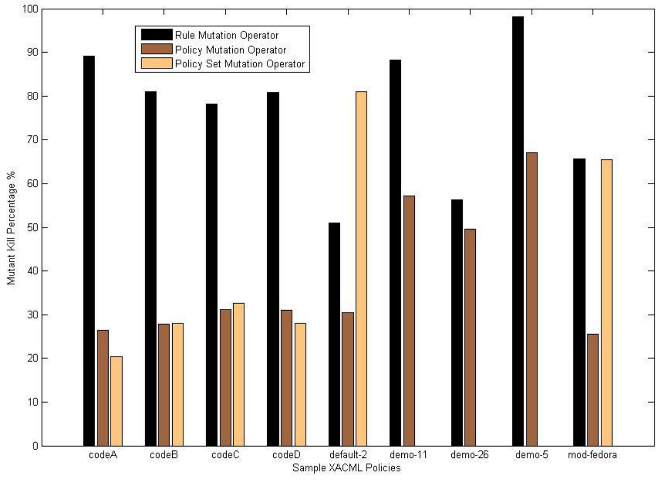

Figure 7: Mutant-killing ratio by different class of operators

5. CONCLUSION

We evaluate the proposed framework of policy testing with nine XACML policies. We perform mutation testing on the policy and the generated request set and compares our results with the existing technique. The mutant kill percentage of the proposed framework is as good as or better than existing techniques. Also, the results indicate that the mutants created by the rule mutation operators have more kill percentage than other classes of operators. This shows that the use of the policy program for generating test cases is able to capture fine errors created by mutants. We got up to

98% of mutant killed by the rule mutation operator and this classification gives better performance in terms of computational cost.

6. REFERENCES

[1] A. J. Offutt R. Geist and F. c. Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5): pages 55-558, 1992.

[2] L. A. Meyerovich K. Fisler, S. Krishnamurthi 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.

[3] T. Xie E. Martin and T. Yu. Defining and measuring policy coverage in testing access control policies. In Proc. 8th International Conference on Information and Communications Security, December 2006.

[4] R. S. Sandhu and P. Samarati. Access control: Principles and Practice. IEEE Communications, 32(9):pages 40-48, September 1996.

[5] Messaoud Benantar. Access Control Systems: security, Identity Management and Trust Models. Springer, 2008.

[6] R. Sandhu S. Osborn and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. ACM Transactions on Information and System Security, 3(2): pages 85-106, May 2000.

[7] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. IEEE Transaction Software Engineering, 17(9): pages 900-910, 1991.

[8] L. J. morell. A theory of fault-based testing. IEEE Transaction Software Engineering, 16(8): pages844-857, August 1990.

[9] E.Martin and T.Xie.Automated Test Generation for Access Control Policies via Change-Impact Analysis. In Proceeding of the 3rd International Workshop on Software Engineering for Secure Syetems (SESS 2007), Minneapolis, MN, pp 5-11, May 2007.

[10] X. Zhang T. Jaeger and A. Edwards. Policy management using access control spaces. ACM Transactions on Information and System Security (TISSEC), 6(3): pages 327-364, August 2003.

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

[12] OASIS eXtensible Access Control Markup Language XACML. http://www.oasisopen.org/committees/xacml, 2005.