Testing Access Control Policy through Change Rule and Swap Rule Algorithm (CRSR)

Patricia Ghann, Ju Shiguang, Conghua Zhou
Jiangsu University
China
pghann@gmail.com, jushig@ujs.edu.cn, chzhou@ujs.edu.cn

ABSTRACT: *We propose an algorithm for generating mutant policies based on XACML Context Schema, known as Change Rule and Swap Rule Algorithm (CRSR). Compared to other testing techniques and tools for testing access control policies, where policy set or policy is evaluated first, our algorithm focuses on the rule and target of a policy set or policy. Our approach represents policy as a vector of bits. A boolean variable 1 represents the applicability of a policy to a request and a boolean variable 0 represents the non-applicability of a policy to a request. Correct policy evaluates to 1: indicating that all the elements, attributes ID and their values are correct. This is done using the XACML Context Schema for a policy and request. We identify and extract the rule and target from the policy and generate request by applying the proposed algorithm. The rule and target are evaluated first on the assumption that policy set specifies what policies may be applicable to a request, while a policy specifies the rules that are required for a policy to be applicable to a request. Mutants generated based on XACML Context Schema for policies using the proposed algorithm is compared with mutants generated by using mutation testing where specific mutant operators are applied.*

Keywords: Policy testing, Original Policy, Bit Policy, Mutation testing, XACML Context Schema, Mutant Policy

Received: 18 October 2014, Revised 24 November 2014, Accepted 29 November 2014

1. Introduction

Access control is a widely recognized security mechanism used in computer systems. Access control systems prevent actions that would lead to violation of access control policy. This is achieved by the use of a reference monitor and an authorization system. The authorization system is a database that contains all allowed subjects, resources and actions mapped unto each other. Specifying and managing correct access control policy is therefore critical and challenging.

Policy testing is an important means of increasing confidence in the correctness of specified policies and their implementation. Policy testing is of two types. In the first type, the artifacts under test are policy specification and the objective of testing is to assure the correctness of policy specification. In the second type, artifacts under test are policy implementations, and the testing objective is to assure conformance between policy specification and implementation. Policy specification and implementation must undergo rigorous verification and validation through systematic testing to achieve these objectives. This is done generally by mutation testing.

Mutation testing uses specified mutant operators to seed a fault into a policy and then test requests against mutant policy and

then test requests against mutant policy and original policy. If the response of these two policies is different from each other, then a fault is detected also termed as killed. The question is what is a correct policy and how is a correct policy determined? In view of this question, our contributions in this chapter include: an algorithm that represents policy as a vector of bits: where a correct policy always evaluates to a boolean value 1. By this we assume that a correct policy is made up of a series of 1bits in a uniquely organized order. We also assume that a request based on the XACML Context Scheme will consist of a series of 1 bit that matches the rule and the target in a policy in a specified order. We use our algorithm on the rule and target of a policy to generate mutant polices. These mutant policies are compared with mutant policies generated using mutant operators and tested on both original and mutant policies.

The rest of this chapter is presented as follows: Section 1.1 presents related work. Section1.2 gives a background on XACML policy specification language and Section 1.3 describes mutation testing. Section 1.4 is on generating mutant policies and request based on XACML Context Schema for a policy and a request. We present our algorithm and its framework in section 1.5. Section 1.6 is on request generation and experiment using the proposed algorithm. Section 1.7 gives a discussion of results. We conclude this chapter in section 1.8.

1.1 Related work

The aim of policy testing is to assure the correctness of policy specification, as well as conformance between policy specification and implementation. To help achieve these, researchers and practitioners have proposed various algorithms to verify general access control properties. Various tools have also been proposed to verify properties for XACML policies. The Alloy analyzer by Hughes and Bultan [3] was used to translate XACML policies to Alloy language. The properties are then verified using Alloy analyzer. Margrave designed by Fisler et al [4, 5] uses multi-terminal binary decision diagram to verify user-specified properties and in the absences of specified properties performed change-impact analysis on two versions of policy. Change-impact analysis is performed automatically by generating specific requests that reveal semantic difference between two versions of a policy. E. Martin and Xie [6-8] developed a fault model for verifying access control properties that used a minimal cover concept for reducing the number of request generated. Zhang et al [9-10] proposed a model checking algorithm and a tool to evaluate access control policies written in RW language, which can be converted into XACML.

Various request generation techniques have also been proposed. The Targen tool [5] derives a set of request by possible combination of the truth-values of attribute id values pairs in a target. The Cirg tool [7] generates test requests based on Change-impact analysis. The simple and multiple combinatorial testing strategies are use to derive a request for each simple and multiple combinations of policy values [10]. Random test generation tool, which analyzes a policy under test and then generate request by randomly selecting requests from the set of all possible combinations of attribute id-values found in a policy [11]. The XPT-based testing strategy [12] for generating requests by using the structures that are obtained by applying the XPT strategy to XACML context schema and an improvement on XPT: Incremental XPT [13-15] capable of reducing the number of requests generated.

The distinguishing features of our proposed algorithm compared to previous work include: our approach represents policy as a vector of bits. A boolean variable 1 represents the applicability of a policy to a request and a boolean variable 0 represents the non-applicability of a policy to a request. Thus the algorithm considers only two responses to a policy; true or false since an indeterminate response results in no action performed by the Policy Enforcement Point (PEP). Instead of seeding pre-defined faults into policy, the values of elements and their attributes are changed in relation to the rule or rules of the policy. This results in the generation of mutant policies and test suits based on XACML Context Schema that ensures high coverage of the various elements in a policy. Our algorithm evaluates the target and the rule first in a policy. This is because we assume that a policy set or policy is a statement that can evaluate to 1 or 0 based on the attribute- values of the element within the policy set or policy. In order words the applicability of a policy to a request implies that the rule or rules when evaluated against the target in the policy matches that of a request according to the order specified within a policy. We generate a number of requests by dynamically manipulating the attribute-values of the target with the specified rule or rules. Mutant policies generated using this algorithm is compared with mutants generated using mutant operators.

1.2 Specifying Access Control Policies using XACML Policy Language

XACML is an OASIS (Organization for the Advancement of Structured Information Standards) standard that describes both a policy language and an access control decision request/response language all written in XML [1, 2]. The policy language is used to describe general access control requirements, and has standard extension points for defining new functions, data types,

combining logic, etc. The request/response language allows a query as to whether or not a given action ought to be allowed, and interpret the result. The response includes an answer about whether the request should be allowed using one of these four values: Permit, Deny, Indeterminate (an error occurred or some required value was missing, so a decision cannot be made) or Not Applicable (i.e. request can't be answered by this service). An XACML policy consists of a Policy set, a Policy, Rule, a Target, and a Condition.

The target specifies the subjects, resources, actions and environments to which a policy can be applied. The subject, resource, action and environment each contain an attributed and data type, and an attribute value that specifies the value associated with it. When a request satisfies a target of a policy, the set of rules of the policy is checked otherwise skipped. The rule of a policy is composed of target, which specifies the constraints of the request to which the rule applies. The rule also contains a condition, a boolean function evaluated whenever a rule is applicable to a request. If a condition evaluates to true, its rule decision is returned. Since there can be one or more policies and rules, combining algorithm exist to help reconcile conflicts. An access request consists of subject, resource, action and environment attributes.

In a typical setup, a request is made to access a resource in a system. The request is sent to what protect that resource such as a file system or a web server, known as Policy Enforcement Point (PEP) illustrated in figure 1.2. The PEP form a request based on a requester's attributes the resource in question, the action, and other information pertaining to the request. The PEP then sends this request to a Policy Decision Point (PDP), which looks at the request and some policy such as the simplified sample policy in figure 1.1 stored among other policies in a repository, which applies to the request, and come up with an answer about whether access should be permitted. The answer or response is returned to the PEP, which decides whether to permit or deny access to the requester. Both PEP and PDP might be contained within a single application, or might be distributed across several servers. In addition to providing request/response and policy languages, XACML also provides other pieces of relationship, namely finding a policy that applies to a given request and evaluating the request against that policy to come up with a yes or no answer.

```
1.    <Policy Id = "univ" RuleCombAlgId = "first-applicable">
2.    <Target>
3.    <Subjects> <AnySubjects/> </Subjects>
4.    <Resources> <AnyResources/> </Resources>
5.    <Actions> <AnyActions/> </Actions>
6.    </Target>
7.    <Rule RuleId = "1" Effect = "Permit">
8.    <Target>
9.    <Subjects> <Subject> Faculty</Subject> </Subjects>
10.   <Resources> Grades </Resources>
11.   <Actions> <Action> Write </Action>
12.   <Action> View </Action> </Actions>
13.   </Target>
14.   </Rule>
15.   <Rule RuleId = "2" Effect = "Deny">
16.   <Target>
17.   <Subjects><Subject> Student </Subject></Subjects>
18.   <Resources> Grades </Resources>
19.   <Actions><Action> Write </Action></Actions>
20.   </Target>
21.   </Rule>
22.   </Policy>
```

Figure 1. 1 Simplified Sample XACML Policy (source [24])

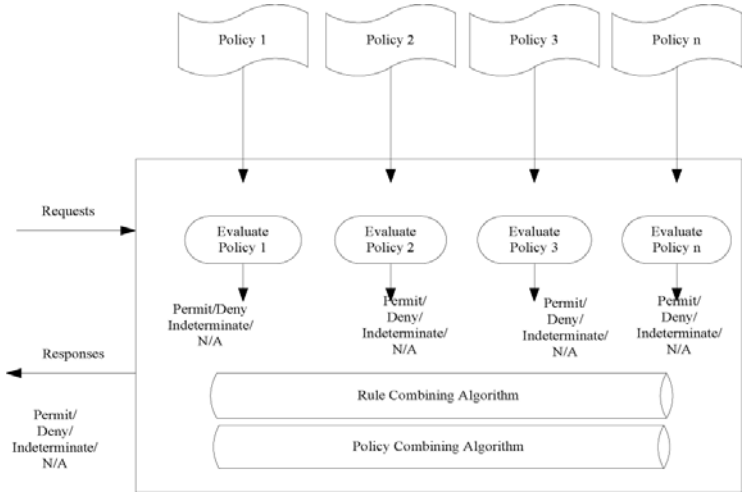

Figure 1. 2 XACML Engine

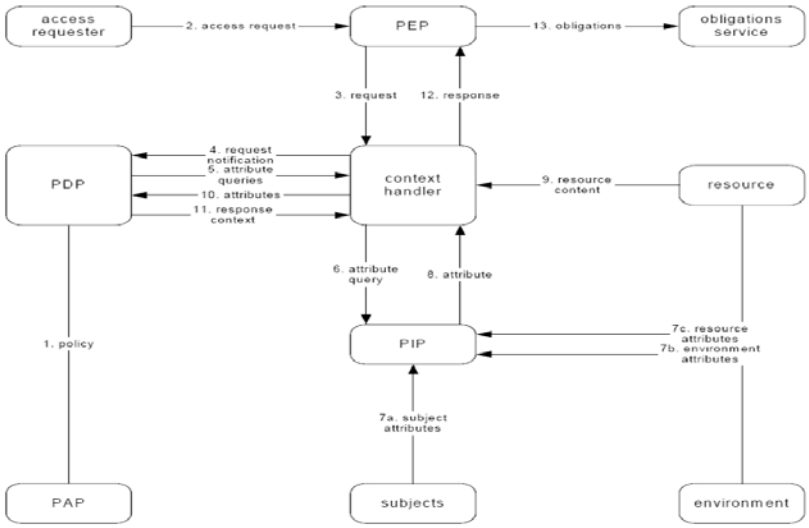

Figure 1. 3 Data-flow diagram (source [2])

Figure 1.3 is a data flow diagram of a XACML engine or model. The order in which the XACML engine functions is described below:

1. The policy administration point (PAP) write policies and policy sets and make them available to the policy decision point (PDP). These policies or policy sets represent the complete policy for a specified target.
2. The access requester sends a request for access to the policy enforcement point (PEP).
3. The PEP sends the request for access to the context handler in its native request format, optionally including attributes of the subjects, resource, action, environment and other categories.
4. The context handler constructs an XACML request context and sends it to the PDP.
5. The PDP requests any additional subject, resource, action, environment and other categories attributes from the context handler.
6. The context handler requests the attributes from a policy information point (PIP).
7. The PIP obtains the requested attributes.
8. The PIP returns the requested attributes to the context handler.
9. Optionally, the context handler includes the resource in the context.
10. The context handler sends the requested attributes and (optionally) the resource to the PDP and the PDP evaluates the policy.

11. The PDP returns the response context (including the authorization decision) to the context handler.
12. The context handler translates the response context to the native response format of the PEP and then returns the response to the PEP
13. The PEP is responsible for fulfilling obligations associated with request.
14. If access is permitted, then the PEP permits access to the resource; otherwise, it denies access.

1.3 Mutation Testing

Mutation testing is renowned in software testing. Historically mutation testing has been used in software design to detect faults in software. It involves the iteration of a program under test in order to produce mutant programs; each having exactly one fault [16-20]. This is done by the use of mutant operators, to generate mutants programs slightly different from the original program under test. Both original program and mutant program are then tested. If the output of these two tests differs, then mutants are said to be killed otherwise alive. To test access control policies, mutation testing is leveraged. In policy testing, the program that is under test is the policy while the test input and output represent the request and response respectively. Table 1 [6-8, 24] is an index of the various mutation operators that are used in testing access control policies. From table 1 mutation operators are not defined individually for subject, resource, action and environment, which are the main components of a request. However they are defined for the target, which encapsulate subject, resource and action and environment. The limitation of mutation testing is that, it does not consider the relationship of a mutant operator with each of the elements of a target. For example the mutant operator PSTF (policy set target false) only set the target to be false. We propose an algorithm for testing access control policies based on the XACML Context Schema for a policy and requests also based on the XACML request context. The algorithm considers the main components of an XACML policy and represents each of the components with a bit value of 0 or 1: a bit value of 0 indicates the absence of a component in a policy while a bit value of 1 indicates the presence of a component in a policy. Thus we represent a policy as a vector of bits. The length of a vector is equal to the different components in a policy. In the same manner we represent each of the attribute values of the components of a request by bit values of 0 and 1.

Type	ID	Description
Syntactic faults	**PSTF**	Policy Set Target False
	PTF	Policy Target False
	RTF	Rule Target False
	RCF	Rule Condition False
Semantic faults	**CPC**	Change Policy Combining Algorithm
	CRC	Change Rule Combining Algorithm
	CPO	Change Policy Order
	CRO	Change Rule Order
	CRE	Change Rule Effect
	PSTT	Policy Set Target True
	PTT	Policy Target True
	RTT	Rule Target True
	RCT	Rule Condition True

Table 1. Index of Mutation Operators

1.4 Generating Mutant Policies and Request using XACML Context Schema for a Policy and Request

The components of XACML policy consists of a policy set, policy, rule, target, condition and effect. In case of multiple policies or rules, a combing algorithm is used to render an authorized decision. A policy consists of a sequence of rules and rule combining algorithm: the procedure for combining decisions from multiple rules. Usually the rule combining algorithm reflects the effect of one of the rules which takes precedence over other rules in an authorized decision. A rule defines who can access what is been protected by a policy and it is applicable based on the target specification. For instance the subject, resource, action and environment set for a particular rule is represented by $S = \{s, s_2, s_3\}$, $R = \{r_1, r_2, r_3\}$, $A = \{a_1, a_2, a_3\}$ and $E = \{e_1, e_2, e_3\}$. The predicate corresponding to this rule is $p = (s_1 \vee s_2 \vee s_3) \wedge (r_1 \vee r_2 \vee r_3) \wedge (a_1 \vee a_2 \vee a_3) \wedge (e_1 \vee e_2 \vee e_3)$. A request set generated must satisfy all possible combination of truth-values for each of this independent clause. For example a request (Q) from $p = (s_1 \vee s_2 \vee s_3) \wedge (r_1 \vee r_2 \vee r_3) \wedge (a_1 \vee a_2 \vee a_3) \wedge (e_1 \vee e_2 \vee e_3)$ is given by:

$$Q \rightarrow \{s_1 \wedge r_1 \wedge a_1 \wedge e_1\}..... \quad (1)$$

An attribute of a rule is an effect. The rule may contain 0 or 1 condition and 0 or 1 target. A condition evaluates to true or false. A target specifies the combination of subject, resource, action and environment attributes to which a rule applies. The target may be specified at the policy set, policy, or rule level and it determines if a policy set, policy or rule is applicable to a request.

In order to use our proposed algorithm to represent a policy, we make the following assumptions:

1. A positively correctly specified policy would contain a string or vector bits of 1 based on the XACML Context Schema for a policy.
2. A negatively correctly specified policy would contain a string or vector bits of 1 and 0 based on the XACML Content Schema.
3. In case of multiple policies or rules, a combining algorithm is needed. The outcome of these combining algorithms include; deny-overrides (represented by the bit value of 0), permit-overrides (represented by the bit value of 1), first-applicable and only-one-applicable (represented by the bit values of either 0 or 1 depending on the order in which the rule is evaluated).
4. A policy in which the subject refers to 'any subject' has a bit value of 0.
5. A correctly specified request would contain a string of 1 bit: each bit representing the attribute value of the components in a request based on the XACML Context for a request. We denote a request and a target by the letter Q and T respectively.
6. The applicability of a policy set (PS), policy (P) and rule (R) to a request (Q) is denoted by A.
7. The evaluation of a correctly specified XACML policy must always be 1.
8. The number of bits in a policy set or policy increases as the policy components increases.

Table 2, represents the main components of XACML Context Schema for specifying access control policies: a policy set (PS) which consists of a policy set id, a target (T), policy id and a policy combining algorithm. A policy consisting of a policy id, rule id, a target (T) and a rule combining algorithm; a rule is made up of a rule id, an effect which can be either permit or deny, a target (T) and a condition. The condition of a rule is represented by a boolean value true or false. The AnyOf element contains a disjunctive sequence of AllOf elements. The AllOf element contains a conjunctive sequence of Match elements. The Match element compares its first and second child elements according to the matching function. The match is positive if the value of the first argument matches any of the values selected by the second argument.

The algorithm we propose does not consider AnyOf, AllOf and Match since their effect is captured in the rule and target.

From the assumptions made, XACML policy with all its components or elements must evaluate to 1 for a policy set, policy or rule to be applicable to a request. Additionally, all the various components with their attribute-values must individually evaluate to 1 to indicate their applicability within a positively specified policy. Based on these assumptions, a policy is represented with a string or vector of bits using the context schema for specifying a policy; these bits represent the attribute values of the main components of XACML policy. The applicability of a policy set or policy (A) is given by 1 as:

$$A \rightarrow \{ \textit{Policy set} \wedge \textit{Policy} \wedge \textit{Target} \wedge \textit{Rule} \wedge \textit{Condition} \wedge \textit{CombID} \} = 1 \quad (2)$$

$$A\ \textit{Policy}\ (P) \rightarrow \{ \textit{Rule} \wedge \textit{RCombID} \wedge \textit{Condition} \wedge \textit{Target} \} \quad (3)$$

$$A\ \textit{Target}\ (T) \rightarrow \{ \textit{subject} \wedge \textit{resource} \wedge \textit{action} \wedge \textit{environment} \wedge R \} = 1 \quad (4)$$

$$A\ \textit{Request}\ (Q) \rightarrow \{ \textit{subject} \wedge \textit{resource} \wedge \textit{action} \wedge \textit{environment} \} \quad (5)$$

Given a policy, we identify the number of rules and targets of the policy. This is to ensure that rules with their conditions are completely covered. We evaluate the rule and the target first, and then consider the policy to ensure complete policy coverage; the reason is we view a policy as a container that contains the rules that must be applied to a certain specified target to enable access. In the same vain a policy set is a container that contains various policies. Furthermore the order in which the rules in a policy are evaluated with respect to the target is very significant in our proposed algorithm; because of rule combining algorithm due to multiple rules. The solution we are proposing is, given a policy, applying the rule with its condition to the target must always satisfy a specified policy for a request to be permitted or denied otherwise. For instance, taking the simplified sample policy in figure 1, and using (2), our policy consists of two strings of 10 bits or vectors of 10 bits as shown in table 3a. Out of these 10 bits, at least 5 bits represent the target (T) which must match the request (Q) based on the Context Schema for a request.

In [21-22] when the TAXI tool is applied on XACML, $3^{Y}*2^{Z}$ intermediate instances can be derived. Where *Y* and *Z* represent the number of elements in the Schema with unbounded cardinality and Z represents the elements in the Schema with [0, 1] cardinality: thus for XACML Context Schema pertaining to request specification, a total of $3^{10}* 2^{2}$ = 236196 intermediate requests are generated. The algorithm proposed is similar to the TAXI tool. Its uniqueness is that, it considers the rule and its position in relation to target elements as most significant and evaluates the rule and the target before the policy or policy set.

Policy Set	$PS ::= PSid = \{T, (PSid \mid Pid)\ CombID\}$
Policy	$P ::= Pid = \{T, (Rid^{+}), CombID\}$
Rule	$R ::= Rid = \{Effect, T, C\}$
Policy Set	$PS ::= PSid = \{T, (PSid \mid Pid)\ CombID\}$
Policy	$P ::= Pid = \{T, (Rid^{+}), CombID\}$
Rule	$R ::= Rid = \{Effect, T, C\}$
Condition	$C ::= true \mid \{f^{boo} \mid (a_1, \ldots\ldots, a_n)\}$
Target	$T ::= null \mid \vee^{\varepsilon++}$
AnyOf	$\varepsilon ::= \vee^{A+}$
AllOf	$A ::= \wedge^{M+}$
Match	$M ::= Attr$ $CombID ::= po \mid do \mid fa \mid ooa$ Effect $::= p \mid d$
Attribute	$Attr ::=$ category (attribute_Value)
Request	$Q ::= (\text{Attr}\mid\text{error(Attr)})^{+}$

Table 2 XACML Policy Components

Components in a policy	**Bit representation of components in policy**
1. RCombAlg id = "*first –applicable*"	1, or 0
2. Rule id =1 *effect is* "*Permit*"	1
3. Subject = "*faculty*"	1
4. Resource = " *Grades*"	1
5. Action 1 for faculty = "*write*"	1
6. Action 2 for faculty = "*view*"	1
7. Rule id=2 effect is "*Deny*"	0
8. Subject = "*Student*"	1
9. Resource = "*Grades*"	1
10. Action = "*write*"	1

Table 3a. Bits Representation of Components in Simplified Sample Policy

a	0	1	1	1	1	1	0	1	1	1
b	1	1	1	1	1	1	0	1	1	1

Table 3b. Bit Policies

From table 3a, we have two original policies (a and b) in bits illustrated in table 3b. The first bit policy is (a) and the second bit policy is (b). Starting from the left in table 4.3b, the first bit represents the rule combining algorithm, which in our simplified sample policy is first applicable and can be either 1 or 0. The second bits represent the first rule id "1" with *permit* effect. This

is followed by the subject (faculty), resource (grade), action (write, view) respectively. The seventh bit represent the second rule id "2" with deny effect and it is followed by the subject (student), resource (grade), action (write) respectively. In the next section we describe how we use our algorithm to generate mutant policies and requests.

1.5 Framework

This section describes the proposed algorithm. The algorithm is based on XACML Context Schema for a policy and a request. The algorithm consists of the following steps. First we extract the rule and target from a policy under test. From our simplified sample policy, our policy is made of two strings of 10 bits or two vector bits of 10 with a bit representing each element in our simplified sample policy. Notice that the position and the order of each element in the policy are very significant. Unauthorized subjects deploy what we term the 'gambling method'; thus changing the order and manipulating the elements in a specified policy to identify a loop-hole in order to take advantage. As shown in table 3a, our sample policy contains two rules: rule id = '1' and rule id = '2' with '*permit*' and '*deny*' effects respectively. There are also two types of subjects; Faculty and Student. The policy also specifies two main types of actions that can be performed on the resource (grade); '*write*' and '*view*'. Although not explicitly stated, it is obvious from the policy that a student can 'view' the resource grade but cannot '*write*' to it. Our algorithm is used to generate mutant policies that cover this and many ambiguities that might exist in the original policy and which can lead to access leakage. The algorithm first identifies the rules; the shaded portion in table 3c and change their bit values to generate new policies; table 3d. The first mutant policies generated are:

a (0011111111) which state that the subject faculty should be denied access to either '*write*' or '*view*' the resource '*grade*'. And the subject student should be allowed access to 'write' to the resource '*grade*'. Access is however denied in both cases since the rule-combining algorithm is '*first-applicable* (i.e. rule Id '1' = 0(deny).

b (1011111111). This policy states that the subject, faculty should be denied access to '*write*' or '*view*' the resource 'grade' but the subject student should be permitted access to '*write*' to the resource 'grade' Both subjects are denied access since the rule-combining algorithm is 'first-applicable'(i.e. ruleId'1'= 0 (deny)). In other words the second part of these mutant policies disputes the intentions of the policy author. However since the rule Id '1' = 0 instead of 1, access is denied.

Next we identify the target (T) in our simplified sample policy. The target consists of two subjects (faculty and student), two types of actions (write and view) and finally one resource (grade). We change the values of each of the elements in the target and apply them to our rules in the mutant policies in table 3d.

a	0	1	1	1	1	1	0	1	1	1
b	1	1	1	1	1	1	0	1	1	1

Table 3c. Identifying the Rules in the Policy

a	0	0	1	1	1	1	1	1	1	1
b	1	0	1	1	1	1	1	1	1	1

Table 3d. Changing the Bit Values of Rules to Generate Mutant Policies

a	0	0	0	1	1	1	1	1	1	1
b	1	0	0	1	1	1	1	1	1	1

Table 3e. Changing the Values of Target Elements in Mutant Policies

Table 3e produces two different mutant policies where the subject (faculty) in the target of ruleId = 1 is 0 or any subject. Mutant policies a: 0001111111 and b: 1001111111 are thus generated. Notice that the ruleId = 1 is the second bit from the left and is 0 in our mutant policies in table 3d. Mutant policies a: 0001111111 and b: 1001111111 state that any subject can write to or view the resource grade and the subject student can write to the resource grade. Since the subject student can be any subject, these mutant policies dispute the intentions of the policy author. Nonetheless access to the resource, grade is denied in both cases because of the rule combining algorithm. We iteratively do this until each element have had its bit value changed to either 1 or 0 while the bit value of ruleId =1 is 0 illustrated in table 3a. In the next step, we generate mutant policies by changing the positions of the two rules; that is ruleId=1 and *ruleId* = 2. Thus *rule Id* = 1 is applied to the target student while *ruleId* = 2 is applied to target faculty.

a	b
0001111111	1001111111
0010111111	1010111111
0011011111	1011011111
0011101111	1011101111
0011110111	1011110111
0011111011	1011111011
0011111101	1011111101
0011111110	1011111110

Table 3f. Mutant Polices Generated

a	0	0	1	1	1	1	1	1	1	1
b	1	0	1	1	1	1	1	1	1	1

Table 3g. Swapping the Position of Rules in Simplified Sample Policy

Swapping the rules generate two different mutant policies; a (0011111111) and b (1011111111). Both policies specify that the subject, faculty should not be permitted to write to or view the resource grade, while the subject student, should be permitted to write to the resource grade which disputes the intention of the policy author. Fortunately both accesses are denied because of the rule combining algorithm. From these two mutant policies, we generate other mutant policies as we did in table 3a. From the algorithm we are also able to generate request simultaneously by changing the attribute values of the target in the policy. And this is one of the unique features about this algorithm compared to other test and request generation techniques; it can be used to generate mutant policies and request at the same time. Additionally it considers each element and their attribute values in a policy as a potential candidate for faulty policy. Furthermore it covers other techniques such as the simple combinatorial [22].

Algorithm

1. Identify the number of rules (R) in a policy and represent them with their bit values

2. If there is more than one policy or rule, then extract the combining algorithm as specified in the policy.

3. Identify the number of target elements (T) in a policy

4. Extract R and T from a policy according to the order specified in a policy

5. Change the bit value of R in the policy.

6. If R has a bit value of one, then change R's value and its effect. Thus if R is 0 with 'deny' effect then change R to 1 with 'permit' effect

7. Apply the changed value of R to the target elements(T) to generate mutant policy

For instance if $R = 0$

8. With $T = \{1, 1,1,1\}$ (i.e. a target with four elements)

9. Applying R to T will result in T_1

10. Thus $T_1 = [0, 1,1, 1,1\}$

11. With R = 0, change the value of each element in the target (T) one at a time to obtain all possible combination of elements with R to generate mutant policies.

12. For example int [ ] T_1 = {0, 1,1,1,1}

13. int [1] T_1 = [0]

14. For (int temp: T_1)

15. Print (temp) + R

Apply combine algorithm to mutant policies generated

16. If there are more than one rule, swap the position

(the order in which the rules appear in the original policy) and then apply them systematically to the target elements to generate mutant policies

17. Apply swapped rules to target by repeating steps described in (10)

18. end

1.6. Experiment

We perform experiment on mutant policies generated by using this algorithm on a set of policies, and compared the results with mutant policies generated by using mutation operators listed in table 1. Requests generated by using the algorithm are used to test the original policies and mutant policies generated by using our algorithm and mutant operators. The policies used in the experiment included a set of real policies such as university administration server policies implemented in a project known as the TAS3 [26], health care service policies (read-patient and dashboard) and three other policies (demo-5, demo-11 and demo-26) [7]. Due to the large number of mutants generated using our algorithm; we randomly selected 100 out of the mutants policies generated for each policy for the experiment and tested with 100 different requests. We did likewise for mutant policies generated using mutant operators. By using mutation operators, we specifically used mutant operators to introduce faults into the policies used in the experiment. Table 4 is a list of the policies with the number of policy set, policy, rule and condition in each policy. Table 5 illustrates the bit representation of the policies used in the experiment. We compare the mutant Killed using the proposed algorithm with that of using mutation operators. The results of the experimented conducted is presented in tables 6. The fault detection capability of applying our algorithm and using mutation operators is illustrated in table 4 .6.

Policy	#Rule	# Condition	#Subject	#Resources	#Action	#Function
University-admin-1	3	0	24	3	3	2
University-admin-2	3	0	24	3	3	2
University-admin-3	3	0	23	3	3	2
read-information-unit	2	1	0	2	1	2
read-patient	4	3	2	4	1	3
dashboard	6	5	3	3	0	4
demo-5	3	2	2	3	2	4
demo-11	3	2	2	3	1	5
demo-26	2	1	1	3	1	4
Student-application-1	2	0	5	2	2	2
Student-application-2	2	0	11	2	2	2

Table 4. Policies used in Experiment

Policy	# Bit in Policy
University-admin-1	35
University-admin-2	35
University-admin-3	34
read-information-unit	8
read-patient	17
dashboard	21
demo-5	18
demo-11	16
demo-26	12
Student-application-1	13
Student-application-2	19

Table 5. Bits Representation of Policies

1.7 Discussion of Results

Table 6 presents a summary of results obtained by using mutant operators and our proposed algorithm. From the experiment we observed that the number of elements in a policy results in the generation of higher number of mutant policies. Consequently to ensure and achieve structure coverage the algorithm has to be used to cover all elements in a given policy.

Bit Representation of Policies	Mutants generated by Algorithm			Mutants generated using mutant operators		
	Mutants	Requests	%Killed	Mutant	Requests	%Killed
35	100	100	99	100	100	82.4
35	100	100	100	100	100	78.99
34	100	100	99.5	100	100	67.87
8	100	100	100	100	100	90.21
17	100	100	90	100	100	85.76
21	100	100	95	100	100	79.78
8	100	100	92	100	100	90.21
16	100	100	95.8	100	100	68.73
12	100	100	92.65	100	100	87.54
13	100	100	99.45	100	100	89.77
19	100	100	99.89	100	100	85.67

Table 6. Results of Experiment

The use of mutant operators such as PSTT, PSTF, PTT, PTF, RTF, RTT,RCT and RCF only ensure that the target evaluate to either true or false, while CPC, CRC, and CRE manipulate the logic construct of the XACML policies. With the proposed algorithm, each element in a policy is regarded as a potential for faulty policy. Representing a policy as a vector of bits enables a wider coverage of policies (policy, rule and condition coverage). The large number of mutant policies generated by the proposed algorithm compared to that of mutant operator is due to the manipulation of individual elements in the policy and their relation and influence on other elements in the policy as a whole. It combines combinatorial and change-impact analysis to systematically generate mutant policies. Additionally, the algorithm simultaneously generate request by manipulating the target elements in the policy. From table 6, the percentage of mutants killed using our algorithm is higher compared to that of using mutant operators; indicating a high capability for fault detection as compared to using mutant operators. Additionally, policy with no condition resulted in almost 100 percentage mutants killed.

Conclusion

The correct specification of access control policies is very important and a difficult task. To ensure correctness and increase confidence in the specification of access control policies, various testing techniques and tools have been proposed. This chapter proposes an algorithm for testing acces s control policies and generating requests based on the XACML Context Schema for policy and request. The algorithm represents a specification of an access control policy as a vector of bits. It assumes that a correct policy must evaluate to one with all the components within the policy also evaluating to one. The algorithm focuses on the manipulation of the various vector bits in a policy by extracting the rules and target from a policy, to generate mutant policies and requests. These mutant policies are then compared with the mutant policies generated by using mutant operators. This algorithm is efficient and effective as it is able to achieve high structural coverage in terms of policy, rule and condition. Compared with using mutant operators, the mutant killed by using our algorithm is relatively higher resulting in a high fault detection capability.

References

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

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

[3] Hughes, G., Bultan, T. (2004). Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara

[4] Fisler, K., Krishnamurthi, S., Meyerovich, L. A., Tschantz, M. C. (2005). *Verification and change-impact analysis of access-control policies*. *In*: Proc. 27th International Conference on *Software Engineering*, p 196–205.

[5] Greenberg, M. M., Marks, C., Meyerovich L. A., Tschantz, M. C. (2005). The soundness and completeness of Margrave with respect to a subset of XACML, Technical Report CS-05-05, Department of Computer Science, Brown University.

[6] Martin, E., Xie, T. Yu. T. (2006). Defining and measuring policy coverage in testing access control policies. In: Proc. 8th International Conference on *Information and Communications Security*, p 139–158.

[7] Martin, E., Xie, T. (2006). Automated test generation for access control policies. In: Supplemented Proc. of ISSRE.

[8] Martin, E., Xie, T. (2007). Automated test generation for access control policies via change-impact analysis. In: Proc. 3rd *International Workshop on Software Engineering for Secure Systems*

[9] Martin, E., Xie, T. (2006) Inferring access-control policy properties via machine learning. In: Proc. International Workshop on Policies for Distributed Systems and Networks, p 235–238.

[10] Chadwick, G. R. D. W., Su, L., Laborde, R. (2009). Use of XACML request context to obtain an authorization decision. MIDDLEWARE, OGF GWD-I

[11] Zhang, N., Ryan, M., Guelev, D. P. (2004). Synthesising verified access control systems in XACML. In Proc. 2004 *ACM Workshop on Formal Methods in Security Engineering*, p 56–65.

[12] Zhang, N., Ryan, M., Guelev, D. P. (2005). Evaluating access control policies through model checking. In Proc. 8th *International Conference on Information Security*, p 446–460.

[13] Ostrand, T. J., Balcer, M. J. (1988). The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31 (6), 676-686.

[14] DeMillo, R. A., Offutt, A. J. (1991). Constraint-based automatic test data generation. IEEE *Trans. Softw. Eng.*, 17 (9):900–910.

[15] Choi, B., Mathur. A. P. (1993) High-performance mutation testing. *Journal of Systems and Software*, 20:135–152.

[16] Budd, T. A. (1980). Mutation Analysis of Program Test Data. PhD thesis, Yale University, 1980

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

[18] Wong, W. E. (1993). On Mutation and Data Flow. PhD thesis, Purdue University.

[19] Fleyshgakker, V. N. Weiss, S. N. (1994). Efficient mutation (analysis: A new approach. *In*: Proc. *International Symposium (on Software Testing and Analysis*, p 185–195.

[20] Horgan, J. R, Mathur, A. P. (1990) Weak mutation is probably strong mutation. Technical Report SERC-TR-83-P, Software Engineering Research Center - Purdue University, December

[21] Howden, W. E. (1983). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8 (4):371–379.

[22] Bertolino, A., Gao, J., Marchetti, E. Polini, A. (2007). Systematic generation of XML instances to test complex software applications. *In*: *Rapid Integration of Software Engineering Techniques* (p. 114-129). Springer, Berlin Heidelberg

[23] Bertolino, A., Gao, J., Marchetti, E., Polini, A. (2007). TAXI - a tool for XML-based testing. In: *Companion to the proceedings of the 29th International Conference on Software Engineering* (p. 53-54). IEEE Computer Society

[24] Johnson, D. S. (1974). Approximation algorithms for combinatorial problems. *J. Comput. System Sci*., 9:256–278.