The ISC Int'l Journal of Information Security

ISeCure

January 2017, Volume 9, Number 1 (pp. 73–91)

http://www.isecure-journal.org

An Automatic Test Case Generator for Evaluating Implementation of Access Control Policies

Marzieh Safarzadeh [1], Mahboubeh Taghizadeh [1], Bahman Zamani [2,*], and Behrouz Tork Ladani [1]

[1] *Department of Software Engineering, University of Isfahan, Isfahan, Iran*

[2] *Model-Driven Software Engineering Research Group, Department of Software Engineering, University of Isfahan, Isfahan, Iran*

ARTICLE INFO.

Article history:

Received: 5 November 2016

Revised: 14 January 2017

Accepted: 27 January 2017

Published Online: 31 January 2017

Keywords:

Access control policies, Automated testing, Model based technique, Implementation of access control, XACML.

ABSTRACT

One of the main requirements for providing software security is the enforcement of access control policies which aim to protect resources of the system against unauthorized accesses. Any error in the implementation of such policies may lead to undesirable outcomes. For testing the implementation of access control policies, it is preferred to use automated methods which are faster and more reliable. Although several researches are conducted for automated testing of the specification of access control policies at the design phase, there is not enough research on testing their implementation. In addition, since access control is amongst non-functional requirements of the system, it is not easy to test them along with other requirements of the system by usual methods. To address this challenge, in this paper, we propose an automated method for testing the implementation of access control in a system. This method, as a model based technique, is able to extract test cases for evaluating the access control policies of the system under test. To generate test cases automatically, a combination of behavior model of the system and the specification of access control policies are used. The experimental results show that the proposed approach is able to find the failures and cover most of the code that is related to access control policies.

© 2017 ISC. All rights reserved.

Archive of SID

1 Introduction

Access control means to ensure that the users can only access what they are authorized to have access to. Access control policies define high-level rules that determine under which conditions, who are allowed to access what resources. To better manage access control, the access control policies are specified using a policy language such as XACML (eXtensible Access Control Markup Language) [1]. In practice, after specifying the access control policies with a policy language and ensuring that there is not any error or inconsistency in this specification, we implement them in the system by hard coding.

Defects in the access control may lead to serious threats such as unauthorized access and escalation of privileges. Therefore, validation of implemented access control policies through testing is both important and inevitable [2]. Access control testing is performed in two levels, including ensuring the correctness of the specification of access control policies, as well

* Corresponding author.

Email addresses: Safarzadeh@eng.ui.ac.ir (M. Safarzadeh), m.taghizadeh@eng.ui.ac.ir (M. Taghizadeh), zamani@eng.ui.ac.ir (B. Zamani), ladani@eng.ui.ac.ir(B. Tork Ladanai)

ISSN: 2008-2045 © 2017 ISC. All rights reserved.

www.SID.ir

as ensuring the correctness of their implementation. First, we should ensure that the specification of access control policies match precisely to the intent and purpose of the system. Also, we ensure that there is not any replication and incompatibility in the rules and policies. Second, it should be checked that there is not any inconsistency between the specification of policies (at design level) and their implementation (at development level), and they have full compliance. In fact, during the implementation phase, for various reasons, e.g., misunderstanding of the specification of access control policies, forgetfulness, or carelessness in coding, the programmers may develop some of these access control policies incorrectly. There exist powerful and automated verification techniques for the first step [3–5]; however, the second step is still a challenge. In other words, although several researches that have been conducted to automate the verification of access control implementation, there is not a fully automated approach for this purpose that has good code coverage. A fully automated method could help in decreasing the cost, time and effort, as well as increasing the speed and repeatability of test process, and easier maintenance of the test suite, along with more complete test and more coverage of code.

As mentioned earlier, the implementation of access control may be incorrect for various reasons, therefore, evaluation of the implemented access control policies, is one of the key issues in the development of secure software. To address this issue, in this paper using model-based testing (MBT) technique, an automated method has been proposed, which is able to extract test cases to evaluate access control policies in given software.

MBT is a popular method for test automation which generates a set of test cases using a model that describes the behavior of the system under test (SUT) [6]. This method uses the behavioral model that represents the functional requirements of the system. Such requirements correspond to the functionalities that end-user expects from the system. However, the non-functional requirements which deal with the quality of the system are neglected. In the proposed approach, we use MBT technique and customize it for access control domain. This way, we are able to generate appropriate test cases for validating the implementation of access control policies in a given system. The test cases are executable and use real data during the test process.

Main contributions of this paper are as follows:

- Proposing an automated MBT method to generate test cases that are able to evaluate the implementation of access control policies.
- Automatic generation of both negative and positive test cases.
- Automatic generation of efficient and real test data through using both database and Microsoft Solver Foundation as constraint solver [7].

In fact, by combining the behavioral model of the given software and the XACML specification of access control policies, we produce a model that is appropriate for generating test cases. The generated test cases include both positive and negative ones and are capable of verifying the implementation of access control policies. Generation of negative test cases along with positive ones will increase the possibility of detecting unknown errors. This is one of the superiorities of the proposed method compared with similar approaches that generate only positive test cases. Furthermore, by automatic extraction of test paths from a model and generating test data via a constraint solver, the proposed method becomes a fully automated one. The proposed method is implemented as a tool called "ACP Test Generator" which is able to automatically generate a set of executable test cases based on the behavioral model of the software and XACML specification of access control policies. The generated test suite is able to evaluate a SUT with a user interface. Also, it is appropriate for testing form applications. Form applications are among systems that are vulnerable through access control faults. Our evaluations that are performed in a case study on a library management system (LMS) with a diverse set of access control rules show that the extracted test suite is able to discover most of the related defects.

The rest of this paper is organized as follows. Section 2 discusses the related work. In Section 3 some preliminary concepts are described. This background is required in the rest of the paper. Section 4 presents the motivation example. Section 5 presents the proposed approach. Section 6 shows our results, and Section 7 presents the conclusions and discusses the future work.

2 Related Work

Access control is an important aspect of a software system, i.e., existence of an error in this part may lead to unwanted outcomes. For this reason, various researches are conducted to ensure the correctness of access control. In the following, most related researches are reviewed.

Masood *et al.* [8] have studied the test of role based access control (RBAC). They use finite state machines (FSM) to model the access control policies. They have also used a fault model to identify user-role assignment, user-role activation, and permission-role assignment faults. Clearly, access control testing is beyond the identification of mentioned faults. Moreover, it is just

Archive of SID

www.SID.ir

able to cover behavioral aspects of the system; while, test data is the main part of a test case and is not captured in this model. Also, Masood *et al.* proposed another approach with FSM [9] and focus on time constraints of access control policies. Mallouli *et al.* [10] have used extended finite state machine (EFSM) to specify the behavior of the system. They have proposed algorithms for the integration of security rules within the system specification. However, this method lacks a mechanism for generating test data.

Xu *et al.* [2] have presented a model-based approach for automated testing of access control implementation. They have modified policies at design level by adding new rules before modeling them and so deal with a complete set of rules. In their approach, each test case is interpreted as a sequence of fired transitions from a Petri Net that describes the access control rules. In this method, the test data are provided manually and are placed in Petri Net as initial marking, and then the transitions use these data to fire.

Mouelhi *et al.* [11] have tried to reduce the required effort to test the security policies. Their idea is to reuse and automatically transform existing functional test cases for testing the security mechanisms. This is possible because each security policy is exercised at least once by functional test cases. However, from a functionality testing point of view, the issue is to determine the correctness of function, not how to access and corresponding security policies. In this method, test generation is not automated. Julliand *et al.* [12] also generate access control tests from functional test model.

Kalam *et al.* [13] have introduced the OrBAC model and in this context it is considered more than RBAC, because it is not only able to model static permissions but also includes contextual rules related to permissions, prohibitions, obligations, and recommendations. OrBAC is used in [10, 14–16] to model access control policies under test. In [14] a methodology is proposed for test case selection, not to generate and implement it. Thus, access control test has been studied at a high level of abstraction. Pretschner *et al.* [15] have focused particularly on the automatic generation of abstract test cases, and they have presented three strategies for this purpose. Strategies which are introduced in [15] increase the rate of test coverage but have ignored the optimality principle. It is possible to examine all states in small systems that have a short hierarchy of roles, permissions and context, but it is not efficient in large systems. Li *et al.* [16] were looking for test purpose generation from Or-BAC rules, and then they generate test cases from these test purposes. In this work, executable test cases are not generated because of using a model at high level of abstraction. In addition, there is no way to generate test data.

Hughes *et al.* [3] and Martin *et al.* [17] have used XACML to specify access control policies. In fact, the automatic generation of test cases is not the goal in [3] but, it automatically checks that there is no conflict between the XACML access control policies. Also, [17] presents a novel framework based on change-impact analysis to generate a set of tests in the form of request-response pairs to evaluate the XACML access control policies. In these works, the evaluation of security policies is performed at the design level. In addition, there exist other studies that have addressed the assessment of XACML policies at design level [3–5, 18].

Masood *et al.* [9] specify access control policies using temporal role based access control (TRBAC) model but as mentioned above, it focuses on time constraints of security policies. Hu *et al.* [19] have introduced verification and conformance testing as complements, since both of them is necessary to validate the system. They have proposed an approach for integrating formal verification and conformance testing for access control model in assurance management framework (AMF). In this method, access control model and its constraints are expressed using Alloy modeling language [20]. The Alloy analyzer, using the assertion expressions, generates the counterexamples that violate constraints of the model. These counterexamples are the positive and negative test cases to be run on the system. In comparison, our approach intends to generate real test data, automatically. Mouelhi *et al.* [21] have presented an interesting and useful framework for specifying, deploying and testing security policies but test cases are not automatically generated.

3 Background

In the following sections we present the background information including the XACML which is a language for describing access control policies, EFSM that is used to model the behavior of SUT, the modified condition/decision coverage (MC/DC) as a code coverage criterion that is used in the proposed approach, and the M2C/DC criterion that is a modified version of MC/DC and is used to generate tests to cover some parts of the system that MC/DC is not able to cover. Note that M2C/DC criterion is in fact a partial contribution of our research published earlier in [22].

3.1 XACML

"XACML is a language specification standard that is designed by OASIS [1] . It can be used to express domain-

[1] Organization for the Advancement of Structured Information Standards

Archive of SID

www.SID.ir

specific access control policy languages as well as access request languages" [23]. The main components of XACML model language include rule, policy, and policy set [1]. A set of rules forms a policy, and each policy set contains several policies. Policy set (also policy, and rule) includes a target that determines the applicability of policy set (policy, rule) for a request. The target is expressed as constraints on the subject, action, resource, and environment. If a request satisfies the target of a policy set (policy), then, that policy set (policy) is applicable for the request.

A rule contains the condition and the effect as well as the target. Condition represents a Boolean expression that determines the applicability of the rule as well as its target. In other words, if a request satisfies the target and condition of a rule, then that rule is applicable for that request. The effect of the rule indicates the result of the application of that rule for a request. Two values are allowed: "Permit" and "Deny". More than one rule (policy) in a policy (policy set) may be applicable to a given request. To resolve conflicting decisions from different rules (policies), a rule combining algorithm (policy combining algorithm) can be specified to combine multiple rules (policies) decisions into a single decision [23].

3.2 EFSM

Extended finite state machine (EFSM) is a modeling approach that has been used to model both the data aspect and the control aspect of a system. In our approach, the behavior of each role in SUT is described as an EFSM.

Definition. An EFSM M is a 5-tuple $M = (S; T; I; O; s_0)$ where S is the finite set of states, T is the finite set of transitions, I and O are the finite sets of input and output respectively, and $s_0 \in S$ is the initial state. Each transition $t \in T$ is a 6-tuple $(s_1; s_2; i; o; c; a)$ where $s_1, s_2 \in S$ are the initial and final states of the transition respectively, $i \in I, o \in O$ are the input and output symbols respectively, c is the constraint (a Boolean expression), and a is the action statements to be performed if the transition fires.

3.3 MC/DC

The modified condition/decision coverage (MC/DC) is a code coverage criterion that helps us to select an appropriate finite set of test cases, among all potential test cases. "The essence of this criterion is that each condition must be shown to independently affect the outcome of the decision, i.e., one must demonstrate that the outcome of a decision changes as a result of changing a single condition" [24]. In the following, we explain the details of MC/DC via an example.

Assume that we want to test the following code excerpt:

if $((i > 2 \,||\, j < 3)\ \&\&\ k > 1)$ { /*$instructions$*/ }

A condition is a Boolean expression that cannot be decomposed into simpler Boolean expressions; hence, in the above code there are three conditions $(i > 2), (j < 3) and (k > 1)$ that will be represented by symbols A, B and C, respectively. A decision is the combination of some conditions, using logical operators AND, OR, and NOT. Hence, if we represent the above decision by symbol S, we have $S = ((A \,||\, B)\ \&\&\ C)$.

The MC/DC extends the condition/decision criterion requiring that each condition should affect the outcome of the decision, independently. In fact, in the test suite that is produced based on the condition/decision criterion, for each condition or decision some test cases should exist such that both $true$ and $false$ values of elements can be considered. However, in the MC/DC criterion, the independent effect of each condition on the outcome of the decision must also be investigated. Independence of a condition is shown by proving that only one condition changes at a time. The requirement of independence ensures that the effect of each condition is tested considering the other conditions. In order to ensure MC/DC, for each individual condition in a decision, one pair of test cases is required, in which the Boolean value of that specific condition has changed while all the other conditions have the same Boolean values. These two test cases produce both $true$ and $false$ values in the outcome of the whole decision. For example, in the above excerpt code, MC/DC guarantees that:

- There is at least one test case, in which the condition A is true and so the outcome of decision is $true$.
- There is at least one test case in which the condition A is false and so the outcome of decision is $false$.
- In the above test cases, all other conditions (B and C) have the same values.
- For other conditions (B and C), some test cases are produced in a similar method.

For a decision with n conditions, we have to find at least $n + 1$ tests in order to be able to ensure MC/DC [25]. As there are three conditions (A, B, C) in our example, we can choose the following four test cases:

1. $A = true;\quad B = false;\ C = true \quad \rightarrow \quad S = true$
2. $A = false;\quad B = true;\ C = true \quad \rightarrow \quad S = true$
3. $A = false;\quad B = false;\ C = true \quad \rightarrow \quad S = false$
4. $A = true;\quad B = false;\ C = false \quad \rightarrow \quad S = false$

Archive of SID

www.SID.ir

Table 1. Test Case Pairs for Condition A.

True	False
TFT	FFT

Table 2. Test Case Pairs for Condition B.

True	False
FTT	FFT

Table 3. Test Case Pairs for Condition C.

True	False
TTT	TTF
TFT	TFF
FTT	FTF

Table 4. Selected Test Cases and Their Coverage.

Test case	Coverage
TFT	A(T), C(T)
FTT	B(T), C(T)
FFT	A(F), B(F)
TFF	C(F)

The procedure to find test cases that ensure MC/DC is as follows[26]:

- Create a table for condition A, which includes two columns. Column one contains all possible test cases, in which condition A is *true* and so the outcome is *true*. Column two includes the test cases that all conditions have the same value as column one except condition A, in addition the outcome should be evaluated to $false$.
- The same procedure is done for other conditions (B and C).
- Select the minimum number of test cases from the tables, such that they cover all conditions (in both *true* and $false$ values) and satisfy the above mentioned rules.

In the above example, we should create three tables for three conditions (Table 1, 2 and 3). Test cases are shown with a string including Ts and Fs, in which T or F means setting the value of corresponding condition to *true* or $false$. For example, the string "TFT" indicates a test case in which condition A evaluates to *true*, B evaluates to $false$, and C evaluates to *true*. Also, the title of each column shows the value of the corresponding condition. For example in Table 1, column "$A(F)$" means that the value of A is $false$. Table 4 shows the selected test cases and their coverage (final result to ensure MC/DC).

3.4 M2C/DC

As mentioned earlier, by using the MC/DC criterion we can produce a finite and appropriate set of test cases. However, it is possible to see some insoluble test cases in the final collection of test cases. In fact, such tests are not logically soluble since there are some inconsistent constraints in their conditions. To solve this problem, we can replace an insoluble test by a soluble one from the set of tests that are previously generated; but if there is not any test to replace, we should generate a new one. The new and old test should have similar properties. In the other words, they should cover same condition and show the independent effect of same condition on the outcome. So, we proposed the M2C/DC idea for solving this problem. In fact, M2C/DC is an improvement of MC/DC and should be applied after using MC/DC criterion, if some insoluble tests are generated. The detail of M2C/DC is proposed in [22], but in the following we will give a brief description.

The generation of insoluble tests is due to some restrictions of the MC/DC criterion. The problem is that in MC/DC, in the pair of test cases that show the independent effect of a condition, other conditions should not be changed. But if the conditions have common variables, this restriction may cause to produce some insoluble tests. In fact, maybe two tests are produced, that in one of them the common variable has a value that is inconsistent with that variable in the other one. So, the main idea for solving this problem is to lose this restriction, if we need to do so. In the other words, in M2C/DC, besides the condition that we considered it to produce appropriate test cases, we can change some other conditions, if by fixing them the produced tests are insoluble. In addition, changing the other conditions should be based on a principle and we cannot change them arbitrarily. In fact, after these changes, the outcome of considered test should be the same as the new one and also the value of the condition that is the subject of the current test to show its independent effect should be the same. Therefore, we can say that the old test could be replaced by the new one.

We can divide the subscription of the variables in the conditions into two groups as follows.

1 The conditions with common variables are fully inconsistent; i.e., the conditions negate each other and each initialization of the variables in these conditions results in inconsistency. For example, in the conditions $x > 0$ and $x < 0$, x is the common variable and only one of the conditions can be true at a time.
In this situation, if only one appropriate pair of test cases is produced for one of these conditions, the other condition will be covered and we do not need to produce a test for it. In fact, because of complete opposition of the conditions,

Archive of SID

www.SID.ir

the test cases that are produced to cover the true/false effect of one of the conditions will cover the $false/true$ effect of the other one as well. So, with one pair of test cases both conditions are covered.

2 The conditions with common variables do not have full inconsistency, but only some initialization of the variables in these conditions causes the inconsistency. For example, if in a conditional expression we have the conditions $x > 30$ and $x < 0$, x is a common variable and the conditions are not always opposite. But, in test cases that both conditions are true, there is not any appropriate initialization for x. So, we cannot produce test cases to cover $x > 30$(T) and $x < 0$(T).
In this situation, if there are not any alternative tests to cover these conditions, we should produce new soluble one by the proposed algorithm. For this purpose, we need to use the equivalent tree of conditional expression. Then, we find the main condition and the other one that is inconsistent with it, and consider the first common sub-tree of these conditions. After that, we change the value of the inconsistent condition to the opposite value. For example, if the value is T $(true)$, the new value is F $(false)$. After that, we should check the outcome of the part of conditional expression that is related to the common sub-tree. According to this outcome and also this new test, four situations may happen:

1 If the outcome is not changed and the new test is soluble, we accept it instead of the old insoluble one.
2 If the outcome is not changed but the new test is insoluble, we should apply our algorithm for it so that a new soluble one is produced.
3 If the outcome is changed and the new test is soluble, we should change the outcome by changing the value of the node that is at the same level of the tree or ate the higher level.
4 If the outcome is changed and the new test is insoluble, at first we should invoke our algorithm to be soluble (similar to situation 2), then we should change the outcome by changing the value of node that is at the same level of the tree or at the higher level (similar to situation 3).

The result of these changes is a set of soluble test cases with same properties as the old one.

4 Motivation Example

We use a library management system (LMS) as a case study to explain our approach. This LMS is almost similar to the one which is used in [14]. We assume that this system is used in a university and has four types of users. Admin manages the accounts of LMS and can create, edit, and delete them. Secretary manages books and is able to create, edit, delete, and classify them. Students and teachers can borrow or return the books. This system has a database that contains the information of accounts and books. In the following, some access control rules are listed for this LMS.

- Students can only borrow non-reference books.
- Students can borrow books for up to two weeks.
- Teachers can borrow non-reference books and reference books if there are more than two copies of them.
- Teachers can borrow books for up to one month.
- Students and teachers who are obligor, are unable to borrow books, until their debt is paid.
- Students can borrow up to four books.
- Students should borrow at least two books from the library of their faculty.
- Student can reserve up to two books for two days.
- Teachers can reserve up to three books for three days.
- Students can extend their books once.
- Teachers can extend their books twice.

In the rest of the paper, we will explain more details about this LMS and illustrate the proposed methods using a series of examples based on this LMS.

5 Proposed Approach

In this section, we present our approach for automatic access control test case generation, which is depicted as a framework in Figure 1. This framework includes four parts. The first part (Input) is a set of inputs that are used in the process. In the second part (Algorithm), the proposed algorithm is shown. The third part (Output) includes abstract and executable test cases as outputs of the algorithm. The fourth part (Application) describes how to use the specified output to demonstrate and correct the defects. In the following sections, with the help of the motivation example (LMS), we describe the proposed framework in more details.

5.1 Input

Input is one of the most important parts of the proposed framework. This part includes SUT model, XACML policies, rule under test (RUT), database, and adaptor. The SUT model is the behavioral model of the SUT and clarifies the functions of that system. Using XACML policies, one can express the access control constraints of SUT. RUT is the rule that is going to be tested and we want to generate test cases to test it. In fact, this rule and also the other constraints of the policy that contains this rule are combined with

Archive of SID

www.SID.ir

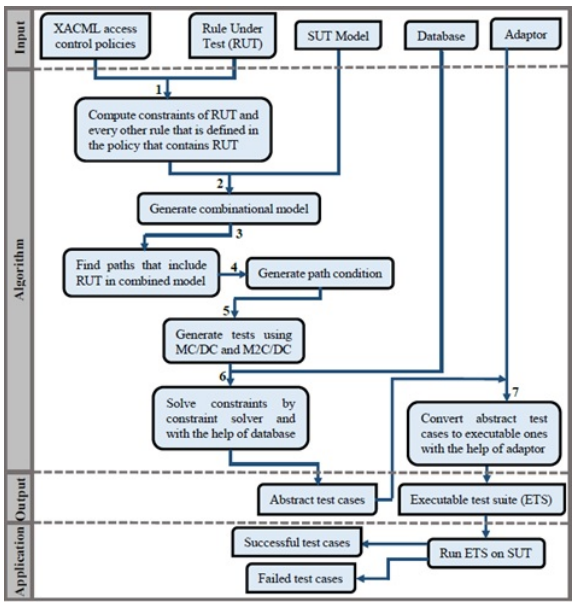

Figure 1. The Proposed Framework

the behavioral model of SUT, and then this combined model is used as one of the inputs of the test process. Database is a collection of tables that stores the data related to SUT. Due to the importance of test data and using the database in most systems, we have to indicate database as an input of our algorithm, so that we are able to generate real data. Finally, the adaptor, as an important input, is used to convert the abstract test cases to the executable ones. In the following, we will describe each of these inputs in more details for the motivation example.

5.1.1 XACML Access Control Policies

As mentioned earlier, XACML is an XML-based standard language for the specification of access control policies. There exist several open source editors for XACML, e.g., UMU-XACML-Editor [27]. In our proposed approach, after writing the specification of access control policies using XACML and ensuring that they are correct, we save them as an XML file and input it to our tool.

Example 1. Figure 2 shows an excerpt of the access control policies for the borrower in LMS. As indicated in this figure, the constraints of access control are written based on the users role. For this purpose, for each role in the system, we considered a policy set and we wrote the constraints related to each role only in its policy set. For example, in this figure we have a role with id "Teacher" that a policy with id "Borrower" is defined for it. In this policy we determined the rules of access control that are related to borrowing a book by teacher role. According to the rule that is shown in this figure, teachers can borrow non-reference books, or reference books with more than two copies.

```
<PolicySet PolicySetId="Teacher" PolicyCombiningAlgId="first-applicable">
  <Policy PolicyId="Borrower" RuleCombinationAlgId="first-applicable">
    <Target>...</Target>
    <Rule RuleId="BorrowBook by Teacher" Effect="Permit">
      <Target>
        <Subjects>...</Subjects>
        <Resources> <AnyResource/> </Resources>
        <Actions>
          <Action>
            <ActionMatch MatchId="equal">
              <AttributeValue>Borrow</AttributeValue>
              <ActionAttributeDesignator AttributeId="actionid"/>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
      <Condition FunctionId="or">
          <Apply FunctionId="equal">
            <SubjectAttributeDesignator AttributeId="bookT.reference"/>
            <AttributeValue>0</AttributeValue>
          </Apply>
          <Apply FunctionId="and">
            <Apply FunctionId="equal">
              <SubjectAttributeDesignator AttributeId="bookT.reference"/>
              <AttributeValue>1</AttributeValue>
            </Apply>
            <Apply FunctionId="greater-than-or-equal">
              <SubjectAttributeDesignator AttributeId="bookT.available"/>
              <AttributeValue>2</AttributeValue>
            </Apply>
          </Apply>

      </Condition>
    </Rule>
    <Rule RuleId="BorrowBook by obligored Teacher" Effect="Deny">
      <Target>
        <Subjects>...</Subjects>
        <Resources> <AnyResource/> </Resources>
```

Figure 2. Partial Access Control Policies of LMS

5.1.2 Rule Under Test (*RUT*)

It should be noted that the access control specification that is described in XACML, has three levels including policy set, policy, and rule. In the proposed approach, we consider the rules and produce some test cases for each of them when test process is performed. In fact, by choosing a rule as *RUT*, we can generate a test suite that is able to evaluate *RUT*. In addition, when we want to generate test suite for a *RUT*, we should consider other rules that are covered with the policy that is related to this *RUT*, so that the effect of those rules on the *RUT* are considered too.

5.1.3 SUT Model

Modeling the behavior of system is based on the formalism that is presented in Section 3.2. The behavioral model of each role only includes functional requirements, i.e., there is not any non-functional requirement such as access control in this model.

Example 2. The behavioral model of the teacher in LMS is shown in Table 5.1.4. The dash symbol (-) in the transitions represents the absence of input, output, constraint, or action. After entering the information of account, the teacher can log into the system (transition from s_0 to s_1). Then she should enter the book id and the number of days to borrow the book. If the book is available and the number of days is greater than zero, the book can be borrowed successfully (transition from s_1 to s_2). If there is not any available copy of the requested book, the teacher can reserve it (transition

Archive of SID

www.SID.ir

Table 5. Student Table in the Database.

FName	LName
Jack	Smith
Ruby	Davis

from s_1 to s_3). Teachers who have borrowed books should return them in less than a month; otherwise they must pay a fine. Due to lack of space, we do not explain other transitions.

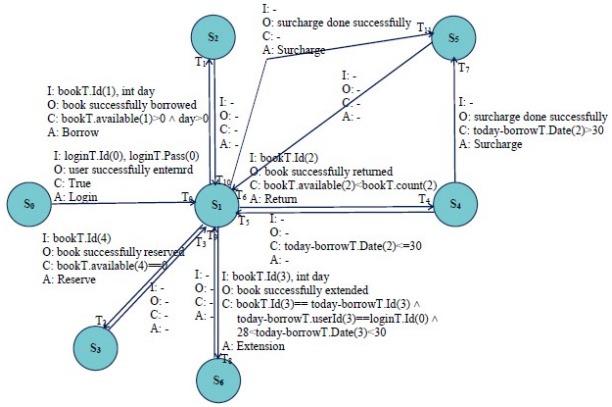

Figure 3. Behavioral Model of the Teacher in LMS

5.1.4 Database

In many cases, according to the input received from the user, the system has to connect to a database to retrieve information. For example, LMS will receive a username and password, and if there is such user, it allows other operations to be performed by that user according to his/her role. In many of the constraints that are used in access control policies, the domain of variables is a set of values extracted from the database. To distinguish these variables from others in the model and in the XACML policies, we use a naming convention as "T.F(index)". This name indicates that the domain of variable is the set of values of column F of table T. The index is a parameter that determines correlation between variables in the model and in the XACML policies. Example 5.1.4 clarifies the reason of using index parameter.

Example 3. Assume that we want to test part of SUT in which we need the first name and the last name of a student. The domain of these variables is table Student in the database. Table 5 shows this table.

Since both first name and last name variables are used in the model and in the XACML policies, we use "Student.FName" to refer to first name and "Student.LName" for last name. In fact, this naming convention indicates the exact domain of the variables. This way of referring to first name and last name, may assign Jack to Student.FName variable and Davis to Student.LName variable, which is not correct and the first name and last name must refer to the same person. To solve this problem, we add the index parameter to the phrases, i.e., Student.FName(1) and Student.LName(1). The same index shows that the value of two variables should be selected from the same row of the table. Also, other variables from other tables that are related to this student, should be named with index "(1)", for example the students score in table Course.

Generating the test data is an important part of a test process. For this purpose, we rely on a constraint solver; we use the Microsoft constraint solver for simple data and also some appropriate queries for database data. In fact, we can generate data for simple variables (non-database ones) by Microsoft constraint solver. However, due to the fact that variables with the domain of database are used in many systems, we should be able to generate appropriate values for such variables too (that it is not supported by Microsoft constraint solver). Hence, in the proposed approach, we generate data for these variables by using appropriate queries. In addition, according to the complexity of constraints that are specified with access control policies and SUT, the queries that should be used to choose the related variable, may be in various forms. Thus we are not able to generate these queries automatically. So in our approach, we must use the queries that are used in the SUT as input. With the help of these queries, we can generate appropriate data for variables with database domain. For example, if C is a constraint on T.F(index) variable, then the query "Select F from T where C" will solve it. In other words, for T.F(index) variable those values of column F of table T are chosen that satisfy the condition C. If more than one result is returned, one of them is chosen randomly.

Example 4. There are variables with names "bookT.Id(1)" and "bookT.available(1)" in the model that is shown in Figure . The same table and the same index for these variables show that both inputs are associated with the same row of the table "bookT". These inputs are related to the fields Id and available, respectively. The constraint of transition that contains these inputs is "bookT.available(1)>0". Constraint solver generates data so that this constraint is satisfied with the following query:
"select bookT.Id, bookT.available from bookT where available>0"

5.1.5 Adaptor

Adaptor maps the abstract actions to SUT [28]. In fact, it maps each component in the model level to its equivalent element in the implementation level. Figure shows the process of how to convert an abstract test

Archive of SID

www.SID.ir

suite into an executable one by the adaptor.

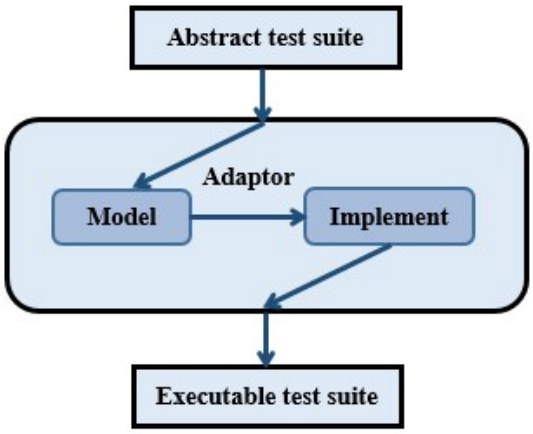

Figure 4. Overview of the Adaptor

We have implemented the adaptor only for form application programs in the format of an xml file. Figure shows an excerpt of the LMS adaptor that indicates how to map the login and borrow actions to the components of the user interface of SUT. As indicated in the figure, an adaptor can contain several action elements. Each action element has a name and includes several sub-action elements. Each sub-action element has a name, equivalent control to perform the sub-action, title of the control, and the form that contains that control. It includes input or output elements that each of them has one name, control to receive input or display output, the title of the control in user interface (controls with no title are identified with their index), and the form that contains the control.

5.2 Algorithm

In this section we will elaborate on the main part of our framework which is labeled as "algorithm" in Figure 1. This algorithm contains 8 steps. As mentioned in Section 3.2, one of the inputs of this algorithm is an EFSM model such as $M = (S; T; I; O; s_0)$ in which every transition $t \in T$ is written as $t = (s; s_0; i; o; c; a)$. Another input to this algorithm is the access control specification of SUT which is described using XACML and includes some *policy sets* with id PS_i that contain some *policies* with id P_j. Each P_j contains some *rules* with id R_k. Constraints identified as PSC_s, PSC_r, PSC_a, and PSC_e are constraints on *subject*, *resource*, *action*, and *environment* in PS_i (constraints on the target element of PS_i). PC_s, PC_r, PC_a, and PC_e are constraints on *subject*, *resource*, *action*, and *environment* in P_j (constraints on the target element of P_j). RC_s, RC_r, RC_a, and RC_e are constraints on *subject*, *resource*, *action*, and *environment* in R_k (constraints on the target element of R_k) and RC_g is a general condition (constraints on the condition element of R_k). In the following, we explain each step of this algorithm.

```
<fsm name="LMS-Adaptor">
  <Actions>
    <action name="Login">
      <subAction name="login" form="Login"
      control="Button" title="Log">
        <Inputs>
          <input name="loginT.Id(1) " form="Login"
          control="TextBox" title="1"/>
          <input name="loginT.Pass(1) " form="Login"
          control="TextBox" title="0"/>
        </Inputs>
      </subAction>
    </action>
    <action name="Borrow">
      <subAction name="select issue" form="LMS"
      control="Button" title="Issue"/>
      <subAction name="do issue" form="IssueTeacher"
      control="Button" title="Issue">
        <Inputs>
          <input name="bookT.Name(1)" form="IssueTeacher"
          control="TextBox" title="2"/>
          <input name="bookT.Id(1) " form="IssueTeacher"
          control="TextBox" title="3"/>
          <input name="int day" form="IssueTeacher"
          control="TextBox" title="0"/>
        </Inputs>
        <Outputs>
          <output name="book successfully borrowed"
                  form="IssueTeacher" control="MessageBox"
                  title="Success Issue"/>
        </Outputs>
      </subAction>
    </action>
  </Actions>
</fsm>
```

Figure 5. A Part of Adaptor for LMS

Step 0: Consider one role and then select one rule to test. In our approach, we assume that in SUT there are several roles that according to their access levels, different access control constraints are defined. Hence, for testing SUT, we should test all rules of each role, separately. In other words, we run our algorithm according to the role of users and the rule that is going to be tested. In the access control specification in XACML, we assume that each policy set is related to only one role. This policy set contains some policies that each of them can contain some rules. At the beginning of the test process we should determine the role of the user (the policy set). Then, we select one policy of this role. After that, we choose one of the rules that this policy contains. In fact, we want to test this rule as RUT and generate test cases for validating it. By using the set of test cases that are produced for all rules of all roles, we are able to test the access control part of SUT.

Step 1: Compute the constraints of *RUT* and all constraints that are indicated for each other's rules defined in the policy containing *RUT*. Let R be RUT, then its constraint is computed as following:

$$RC = PSC_s \wedge PSC_r \wedge PSC_e \wedge PC_s \wedge PC_r \wedge PC_e \wedge RC_s \wedge RC_r \wedge RC_e \wedge RC_g \quad (1)$$

A policy can be applied to a request when the request satisfies the target of the policy and policy set con-

Archive of SID

www.SID.ir

taining that policy. A rule can be applied to a request when the request satisfies the target and condition of rule and target of policy that contains that rule. In Formula 1, we have ignored the constraints of action because it will be checked in step 2.

In addition, for each of other rules that are mentioned above, we should compute a conditional expression similar to Formula 1.

Example 5. Consider the following rule in LMS: *"Teachers can borrow non-reference books or reference books that there exists more than three copies of them, for up to one month."*
According to Formula 1 and rule with id "Borrow Book by Teacher" in XACML policies which are described in Figure 2, the constraint of this rule is computed as follows:

$PS =$ "LMS"
$P =$ "$Borrower$"
$R =$ "$Borrow\ Book\ by\ Teacher$"
$PSC_s = True$
$PSC_r = True$
$PSC_a = True$
$PSC_e = True$
$PC_s = True$
$PC_r = True$
$PC_a =$ "$actionid == Borrow$"
$PC_e = True$
$RC_s =$ "$loginT.role(0) == 'teacher'$"
$RC_r = True$
$RC_a = True$
$RC_e = day \leq 30$
$RC_g =$ "$bookT.reference(1) == false \vee (bookT.reference(1) == true \wedge bookT.available(1) \geq 3)$"

$RC =$ "$(loginT.role(0) == teacher) \wedge (day \leq 30) \wedge (bookT.reference(1) == false \mid (bookT.reference(1) == true \wedge bookT.available(1) \geq 3))$"

Step 2: Generate combinational model. In this step, the model that contains access control information is generated. This model is combination of the behavioral model and all constraints that are indicated by RUT and also other rules of that policy that contains RUT, is the start point of the test process. This step is performed in two sub steps as follows:
2.1. Search the model to find transitions with action value as follows:

$$RC = PSC_a \wedge PC_a \wedge RC_a \tag{2}$$

We find those transitions of model that are related to RUT and also all other rules of the policy that contain RUT. In fact, we should consider the actions of these rules and find the transitions that have one of these actions in their action.

2.2. Update the conditions of the found transitions as follows:

$$t = (s, s_0, i, o, c \wedge RC, a) \tag{3}$$

In each transition t, RC is the constraint of the related rule that is obtained from formula 1.

In fact, in this step the transitions that contain action under test in RUT and also the actions of other rules in the related policy, are found and their condition element are equipped with these access control constraints. In addition, we need to store the transitions that are related to RUT for the next step.

Example 6. Consider the behavioral model of the teacher that is shown in Table 5.1.4. According to this figure, step 2 can be performed as follows:
2.1. Search the model and find transition t_1 that contains action under test which is computed according to Formula 2:

a1 = "actionid == Borrow"

2.2. Update the constraint of t_1 as follows:

$c1 = (bookT.available(1) > 0 \wedge day > 0)$
$\wedge ((loginT.role(0) == teacher) \wedge (day <= 30)$
$\wedge (bookT.reference(1) == false$
$\mid (bookT.reference(1) == true$
$\wedge bookT.available(1) >= 3)))$

Step 3: Find the paths that include RUT in the combined model. In this step, we want to find all paths that their source is the initial state of the model and their destination is the destination of a transition that its action is the action of RUT. For this purpose, we set the source state of the transitions that are found in the previous step and are related to RUT, as the start point and move backward in the model until we reach S_0 (the initial state). Thus, for each transition that is related to RUT, one or more paths are identified in the model. Each path is a sorted sequence of transitions. If in M the set of transitions is $T = (t_0, , t_n)$, then each path is defined as Formula 4 in which t_0 is a transition with source of initial state s_0 and t_i is the transition that is related to RUT.

$$RP = t_0, ..., t_i \tag{4}$$

For each of these identified paths, the next steps of algorithm should be repeated separately.

Example 7. In Table 5.1.4, t_1 with initial state s_1 is the only found transition in step 2. In step 3, we move back from state s_1 in the model until we reach state s_0. Thus, $RP = t_0, t_1$ is determined as the only path which is found in this step.

Step 4: Generate path condition. If the path of RUT is in the form of Formula 4, then the path

Archive of SID

www.SID.ir

Table 6. Test Case Pairs for Condition "bookT.available(1) > 0".

True	False
TTTTTTF	FTTTTTF
TTTTTFT	FTTTTFT
TTTTTFF	FTTTTFF
TTTTFTT	FTTTFTT

Table 7. Test Case Pairs for Condition "day>0".

True	False
TTTTTTF	TFTTTTF
TTTTTFT	TFTTTFT
TTTTTFF	TFTTTFF
TTTTFTT	TFTTFTT

condition can be computed as Formula 5, where c_k is the constraints of t_k in the combinational model.

$$PC = c_0 \wedge ... \wedge c_i \tag{5}$$

Because of using the combinational model, in which the condition elements of transitions were updated in step 2, PC contains constraints of RUT too.

Example 8. Assuming $RP = t_0; t_1$ and according to update constraint of t_1 in Example 5.1.3, PC is calculated as follows:

$$\begin{aligned} PC = c_0 \wedge c_1 = True \wedge (&bookT.available(1) > 0 \\ &\wedge day > 0) \wedge ((loginT.role(0) == \text{`}teacher\text{'}) \\ &\wedge (day <= 30) \wedge (bookT.reference(1) == false \\ &|(bookT.reference(1) == true \\ &\wedge bookT.available(1) >= 3))) \end{aligned}$$

Step 5: Generate tests using MC/DC and M2C/DC. By considering PC as a decision which contains several conditions, we can generate tests using MC/DC criterion. With the help of this criterion described in Section 3.3, the impact of changes of any condition is examined independently. As mentioned earlier, if PC contains n conditions, then $n+1$ tests are generated using MC/DC. So in the running example, 8 tests are selected, because PC contains 7 conditions. Example 9. PC that was generated in Example 5.1.4, is analyzed according to MC/DC criterion, and Table 6 to 12 show tests which can examine conditions with two values true and false.

Table 13 shows set of all tests that are generated in Table 6 to 12 and their coverage list. For example, consider FTTTTTF. The meaning of this test is as follows:

Table 8. Test Case Pairs for Condition "loginT.role(0)== teacher".

True	False
TTTTTTF	TTFTTTF
TTTTTFT	TTFTTFT
TTTTTFF	TTFTTFF
TTTTFTT	TTFTFTT

Table 9. Test Case Pairs for Condition "day<=30".

True	False
TTTTTTF	TTTFTTF
TTTTTFT	TTTFTFT
TTTTTFF	TTTFTFF
TTTTFTT	TTTFFTT

Table 10. Test Case Pairs for Condition "bookT.reference(1) ==false".

True	False
TTTTTTF	TTTTFTF
TTTTTFT	TTTTFFT
TTTTTFF	TTTTFFF

Table 11. Test Case Pairs for Condition "bookT.reference(1) ==true".

True	False
TTTTFTT	TTTTFFT

Table 12. Test Case Pairs for Condition "bookT.available(1) >=3'.

True	False
TTTTFTT	TTTTFTF

$$\begin{aligned} FTTTTTF = &(bookT.available(1) < 0) \wedge (day > 0) \\ &\wedge (loginT.role(0) == teacher) \wedge (day <= 30) \\ &\wedge (bookT.reference(1) == false) \\ &\wedge (bookT.reference(1) == true) \\ &\wedge (bookT.available(1) < 3) \end{aligned}$$

We should choose minimum number of tests from Table 13 such that all criteria are covered and both true and false independent effect of all conditions on the outcome of PC can be shown. For this purpose, we select tests using "size of coverage list" column in Table 13. In fact, any test with maximum size of coverage is selected and its coverage list in "coverage" column will be removed from the coverage list of other tests. Therefore, with each test selection, "coverage list" of other tests will be updated. If more than one test have the maximum "size of coverage list", then one of them which is compatible with selected tests should be selected (to maintain MC/DC guarantees). This procedure continues until all criteria are covered and

Archive of SID

www.SID.ir

so under these circumstances, size of all coverage lists is zero. If constraints in a selected test is not logically soluble (it is not true logically, e.g., TTTTFFT where the book is both reference and non-reference), then we should choose another test to cover that criterion, which is not covered by other selected test, and replace the previous insoluble test with it. This will be done in the next step, where constraints must be solved. In some cases maybe there is no test with soluble constraint to cover special criteria. In these situations, we use M2C/DC that is introduced in Section 3.4.

Step 6: Generate abstract test cases. Each abstract test case contains test input data, test path, and expected output. In this step, each test which is selected in the previous step is converted into an abstract test case. Algorithm 1 shows generating the abstract test suite. According to this algorithm, the constraint solver that is able to solve numeric, string, char, and database constraints, receives the selected test and generates data which satisfies the constraints. These data are combined with the test path and provide abstract test cases. If the generated data satisfy PC (i.e., by putting the generated data in PC, it is evaluated to True), then the test case is a positive one and should be added to positive test suite (PTS), otherwise, it is a negative one and is added to negative test suite (NTS).

Algorithm 1 Abstract Test Suite Generation

Input: RP (a Rule Path), CS (constraint set resulted from Analyzing PC), DB (SUT's DataBase), PC
Output: Abstract Test Suite

1: $TC = \emptyset$; $PTS = \emptyset$; $NTS = \emptyset$; $TS = \emptyset$; $RPcopy = RP$; $D = Solve(CS, DB)$; {D is a set of collection, each collection is set of data assigned to variable}
2: **while** $D \neq \emptyset$ **do**
3: choose $d \in D$; {d = {(variable,data) | data assigned to variable according to CS solving by constraint solver }}
4: $D = D - \{d\}$;
5: **while** $RP \neq \emptyset$ **do**
6: choose $t \in RP$; $\{t = (s_1, s_2, i, o, c, a)\}$
7: $tc = action(t) + input(t, d) + output(t)$;
8: $TC = TC \cup tc$
9: $RP = RP - \{t\}$
10: **end while**
11: **if** validate(PC, TC) = **true then**
12: $PTS = PTS \sup TC$;
13: **else**
14: $NTS = NTS \cup TC$;
15: **end if**
16: $TC = \emptyset$
17: $RP = RPcopy$;
18: **end while**
19: $TS = PTS \cup NTS$;
20: **return** TS;

Example 10. According to Figure 4, the constraint solver section receives the selected tests in Example 5.2 and generates the data after it solves them. For example, TTTTTFT is a selected test according to Table 13. Constraint solver should generate data so that the following constraints are satisfied: *bookT.available(1) > 0, day > 0, loginT.role(0) == teacher, day <= 30, bookT.reference(1) == false, bookT.reference(1) != true, bookT.available(1) >= 3* In this example, except day which is a numeric constraint, other are solved by database constraint solver

Table 13. Final Selected Constraints

Test	Size of coverage list	Coverage	Selected
TTTTTTF	5	bookT.available(1)>0(T) day>0(T) loginT.role(0)=='teacher'(T) day≤30(T) bookT.reference(1)==false(T)	No
TTTTTFT	5	bookT.available(1)>0(T) day>0(T) loginT.role(0)=='teacher'(T) day≤30(T) bookT.reference(1)==false(T)	Yes
TTTTTFF	5	bookT.available(1)>0(T) day>0(T) loginT.role(0)=='teacher'(T) day≤30(T) bookT.reference(1)==false(T)	No
TTTTFTT	6	bookT.available(1)>0(T) day>0(T) loginT.role(0)=='teacher'(T) day≤30(T) bookT.reference(1)==true(T) bookT.available(1)≥3(T)	Yes
FTTTTTF	1	bookT.available(1)>0(F)	No
FTTTTFT	1	bookT.available(1)>0(F)	No
FTTTTFF	1	bookT.available(1)>0(F)	No
FTTTFTT	1	bookT.available(1)>0(F)	Yes
TFTTTTF	1	day>0(F)	No
TFTTTFT	1	day>0(F)	No
TFTTTFF	1	day>0(F)	No
TFTTFTT	1	day>0(F)	Yes
TTFTTTF	1	loginT.role(0)=='teacher'(F)	No
TTFTTFT	1	loginT.role(0)=='teacher'(F)	No
TTFTTFF	1	loginT.role(0)=='teacher'(F)	No
TTFTFTT	1	loginT.role(0)=='teacher'(F)	Yes
TTTFTTF	1	day≤30(F)	No
TTTFTFT	1	day≤30(F)	No
TTTFTFF	1	day≤30(F)	No
TTTFFTT	1	day≤30(F)	Yes
TTTTFTF	2	bookT.reference(1)==false(F) bookT.available(1)≥3(F)	Yes
TTTTFFT	2	bookT.reference(1)==false(F) bookT.reference(1)==true(F)	Yes
TTTTFFF	1	bookT.reference(1)==false(F)	No

Archive of SID

www.SID.ir

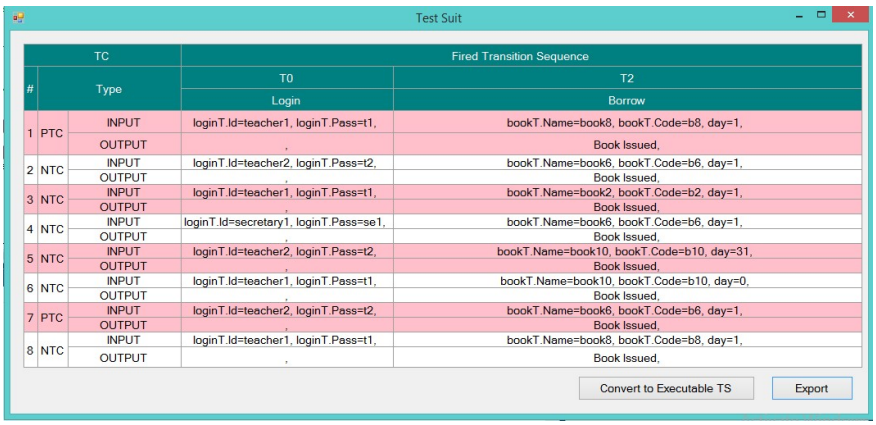

Figure 6. Generated Abstract Suite in ACP Test Generator Tool

(using the appropriate queries on system). Generated data are combined with the test path in Example 5.1.4 and provide abstract test cases. Abstract test suite for our case study is depicted in Figure 6. The first test case is resulted from TTTTTFT, where generated data satisfy the mentioned constraints, because "book8" is a non-reference book and 4 copies of this book exist in the library. The tool that implements the framework presented in Figure 1 is called "ACP Test Generator". It is explained in Section 6.

In final step, abstract test cases should be converted into test cases which are executable on SUT. In other words, using an adaptor, for each abstract test case a test script is created. We can run these test scripts on SUT and then check the results to find the errors of SUT.

Step 7: Generate executable test cases. With the help of the algorithm that is depicted in 2, each abstract test case is converted into an executable one. The test script that represents the executable test case (ETC) starts with a special header file (HF). Then each action in the abstract test case (ATC) is selected sequentially and its sub actions are added to ETC. In order to add each sub action to ETC, after determining the form containing sub action based on what is specified in the adaptor, its inputs are replaced in the corresponding controls. Then sub action is executed and outputs are placed in their corresponding controls according to adaptor. We used UI automation interface and thread libraries in c# to perform these actions automatically. If these outputs match with the output in the ATC, then positive test case has succeeded and negative test case has failed, and vice versa.

Example 11. Figure 7 shows part of test script (executable test case) corresponding to test case 1 in Figure 6. This test script is related to enter user name and password and then click on login button.

Algorithm 2 Algorithm convert Abstract Test Case to Executable Test Case

Input: ATC (Abstract Test Case), Adaptor, HF (Header File)
Output: ETC (Executable Test Case)

$ETC = HF$;
$AC = Actions(ATC)$ {ordered sequence of actions in ATC}
while $AC \neq \emptyset$ **do**
  choose $a \in AC$;
  $AC = AC - \{a\}$;
  $SAC = Subactions(a)$; {ordered sequence of subctions in a}
  while $SAC \neq \emptyset$ **do**
    choose $s \in SAC$;
    $SAC = SAC - \{s\}$
    $ETC = ETC + Form(s, Adaptor)$; {the choice of the form of subaction}
    $IN = Inputs(s)$;
    while $IN \neq \emptyset$ **do**
      choose $i \in IN$;
      $IN = IN - \{i\}$;
      $ETC = ETC + Control(i, Adaptor)$; {placement test input at the corresponding corntrol}
    end while
    $OUT = Outputs(s)$;
    while $OUT \neq \emptyset$ **do**
      choose $o \in OUT$;
      $OUT = OUT - \{o\}$;
      $ETC = ETC + Control(o, Adaptor)$; {compare the expected output with corresponding control}
    end while
  end while
end while
return ETC;

5.3 Output

The output of our algorithm is a test suite that contains both positive and negative test cases. By executing each of these test cases, we are able to test the SUT and find its errors. In addition, using database data generator in our constraint solver section, leads to generate actual and appropriate test data for SUT. It is important to note that our approach not only checks the correctness of defined operations for access

Archive of SID

www.SID.ir

```
using System;
using System.Windows.Automation;
using System.Diagnostics;
using System.Threading;
using System.IO;
namespace TestScenario{
  public class Program{
    public static void Main(){
      string tcResult = "";
      try{
        Console.WriteLine("\nBegin WPF UIAutomation test run\n");
        Process p=Process.Start(@"D:\LibraryManagement System\Library_Software
                          \Library_Software\bin\Debug\Library_Software.exe");
        Thread.Sleep(2000);
        AutomationElement aeDesktop = AutomationElement.RootElement;
        AutomationElement aeForm00 = null;
        int numWaits00 = 0;
        do{
          aeForm00 = aeDesktop.FindFirst(TreeScope.Children, new
                   PropertyCondition(AutomationElement.NameProperty, "Login"));
          Console.WriteLine("Looking for Login");
          ++numWaits00;
          Thread.Sleep(100);
        }while (aeForm00 == null && numWaits00 < 10);
        if(aeForm00 == null)
          throw new Exception("Failed to find Login"); tcResult = "Fail";
        else Console.WriteLine("Found it!");
        Console.WriteLine("Finding all user controls");
        AutomationElementCollection  aeAllTextBoxes00 =
        aeForm00.FindAll(TreeScope.Children,new PropertyCondition(
        AutomationElement.ControlTypeProperty,ControlType.Edit));
         AutomationElement aeTextBox00 = aeAllTextBoxes00[1];
         Console.WriteLine("\nSetting TextBox-1 to 'teacher1'");
         ValuePattern vpTextBox00=
         (ValuePattern)aeTextBox00.GetCurrentPattern(ValuePattern.Pattern);
         vpTextBox00.SetValue("teacher1");
         AutomationElement aeTextBox01 = aeAllTextBoxes00[0];
         Console.WriteLine("\nSetting TextBox-0 to 't1'");
         ValuePattern vpTextBox01=
         (ValuePattern)aeTextBox01.GetCurrentPattern(ValuePattern.Pattern);
         vpTextBox01.SetValue("t1");
         AutomationElement aeButton00=aeForm00.FindFirst(TreeScope.Children,
         new PropertyCondition(AutomationElement.NameProperty, "Log"));
         Console.WriteLine("Clicking on Log button");
         InvokePattern ipClickButton00=
         (InvokePattern)aeButton00.GetCurrentPattern(InvokePattern.Pattern);
               ipClickButton00.Invoke();
```

Figure 7. A part of test script corresponding to test case 1 in Figure 7

control, but also with the help of negative test cases can detect those operations, which should not be in SUT. So it is possible to discover more errors in the implementation of access control policies in SUT.

5.4 Application

With regard to the production of positive and negative test cases in the proposed approach, it is required to apply test oracle at application level. Test oracle is a mechanism to determine success or failure of test cases. In this discussion, a successful run of SUT is equal to the success of positive test cases, while success of negative test case shows running SUT with failure. Because negative test cases violate the access control conditions, test oracle receives actual outputs from SUT and compares them with the expected outputs, which are specified in adaptor. If the actual and the expected outputs are the same, positive test case succeed and the negative one failed; and vice versa. Each failure of test indicates a flaw in the SUT or model. Accordingly, the flaw should be modified and then the tests should be repeated.

6 Results

In this section we will evaluate the proposed solution and the corresponding tool. Evaluation of a testing method means to assess its ability to detect errors and unknown defects in a system. For this purpose, we use two test adequacy criteria. The first one is based on mutation analysis and the second one is the level of code coverage. In the following, each of these criteria and the related results will be investigated.

It should be emphasized that, in the proposed approach we assume that the correctness of the specification of access control policies have been checked already and we only want to test the implementation of them in the SUT. For evaluating our approach with the mutation analysis method, at first we should inject some vulnerabilities into the access control part in hard code of SUT, then run the test cases that are previously generated, on this vulnerable SUT. After that, we should check the results of these runs and determine the ability of test cases to detect the injected vulnerabilities with the help of mutation score in Formula 6. In this formula "killed mutations" are the mutations that are recognized by test cases.

$$MutationScore = (killedmutation/allmutation) \times 100 \quad (6)$$

The vulnerabilities that are injected into SUT are similar to what are produced by the mutant operators of the mutation analysis technique that are used to test the correctness of the specification of access control policies. In this technique, by using mutation operators some unknown vulnerabilities are injected into the specification of access control policies, then given a set of requests, this approach evaluates each of these requests on both original policy and the mutant one. After evaluation of each request, two responses will be produced; one response for the request on original policy and the other one for the request on mutant policy. If these responses are different, then it means that the mutant policy is killed by this request, otherwise it is not killed [29]. So in our evaluations, for a vulnerability that is injected into the specification of policies, we inject a similar one in the hard code of SUT; then we run our test cases on the new vulnerable SUT and check the results.

Given that in the proposed approach, XACML is used to describe access control policies, we should use the mutants of such policies. In [30] a tool that is called XACMUT is developed to create mutations in XACML policies. This tool has some mutation operators to create different mutations in policies. In our evaluations, we choose 12 appropriate operators among of those. In Table 14 these operators are presented. For more information about others refer to [30].

The following procedure was performed to assess the capability of ACP Test Generator in detecting vulnerabilities:

Archive of SID

www.SID.ir

Table 14. Selected Mutation Operators in XACMUT.

Operator	Abbreviation
CRE	Change Rule Effect
RTT	Rule Target True
RTF	Rule Target False
RCT	Rule Condition True
RCF	Rule Condition False
RPT	Rule Parameter Type
ANR	Add New Rule
RER	REmove Rule
ANF	Add Not Function XACML
RNF	Remove Not Function XACML
CLF	Change Logical Function XACML
CCF	Change Comparison Function XACML

1. Start
2. Define three variables AllMu, KillMu, AliveMu with initial values of zero.
3. Select a mutation operator from XACMUT operators list that has not been reviewed so far, or jump to step 12 if there is not any unexamined operator.
4. Generate mutated access control specification by XACMUT based on the selected operator.
5. Select a mutation that has not been studied so far, or jump to step 10 if there is not any unexamined mutation.
6. Apply mutation in SUT and increment AllMu.
7. Run the executable test suite, which is already generated, on the vulnerable SUT and review the results.
8. If at least one failure is reported, then the mutation is killed, otherwise it is alive. Accordingly increment KillMu or AliveMu.
9. Delete mutant from SUT and return to step 5.
10. Calculate the mutation score based on the values of variables, according to Formula 6.
11. Reset variables to zero and return to step 3.
12. Stop

Figure 8 shows the mutation scores for selected XACMUT operators in LMS, which are obtained using the above procedure. Due to using M2C/DC to generate alternative tests for insoluble ones and also the database constraint solver that is used in our proposed approach, the generated test cases cover most of the access control part of the system and explore most of the mutations that are injected into them. Nonetheless, we cannot find alternative tests for some of insoluble tests with the help of M2C/DC. So some part of SUT may not be covered and then the related mutations are not killed. In our case study such mutants are produced by mutant operators CRE, RER, and CCF. Also, most of other mutants can be killed by other soluble tests. Furthermore, another mutation operator that we used is ANR. This operator adds some new rules to access control part. These rules are generated by using the properties of other rules that previously existed in this part, like the action and subject. In fact, a new rule is a combination of some properties from several existing rules. Since respecting to the action of new rule is one of the actions that exist in SUT, this mutant can be detected in test process. In fact, in this mutant we change part of the system that is related to the action of new rule. Also according to the access control rules of SUT, for this action we already produced some test cases. So with the help of these test cases we can detect the mutant that is injected into SUT by the new rule. As it is indicated in Figure 8, some of the mutants that are injected into SUT by ANR operator were not detected. This is because of the new rules, which their actions are not any of the SUT actions. So we cannot detect these mutants with our test cases.

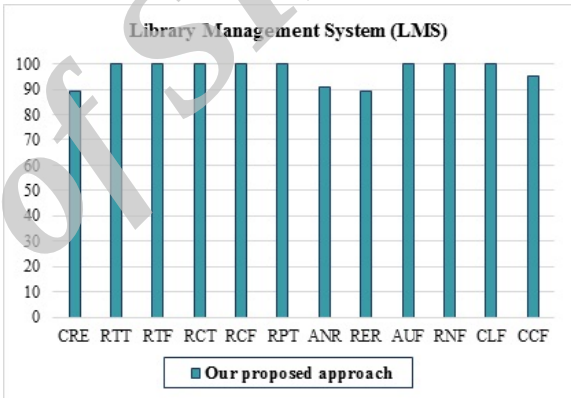

Figure 8. Mutation Score of LMS for XACMUT Operators

6.1 Output

In addition, some mutations like those that are generated by RTF and RCF can only be killed by the negative test cases. For this reason, other approaches which are limited to generate positive test cases cannot discover these mutations. For example, the following rule in LMS has been investigated in [2]:
"Borrowing available books are permitted on working days, prohibited on holidays, and undefined on maintenance days."
The test suite which is shown in Figure 9 is generated based on [2] to test this rule. In our approach, the mentioned rule is described in terms of four rules in XACML (for the roles of teacher or student):

- Borrowing books on working days (WD) by teacher

$$PC = (loginT.role(0) == \text{“}teacher\text{”}) \wedge (day == \text{“}WD\text{”}) \wedge (bookT.available(1) > 0)$$

Archive of SID

www.SID.ir

Table 15. Constraints of Test Cases for First Rule.

TC#	Test	loginT.role(0)	day	bookT.available
TC0	TFTT	teacher	WD	> 0
TC1	FFTT	!*teacher*∧!*student*	WD	> 0
TC2	FTFT	student	! WD	> 0
TC3	TFTF	teacher	WD	<= 0
TC4	FTTT	student	WD	> 0

Table 16. Constraints of Test Cases for Second Rule.

TC#	Test	loginT.role(0)	day	bookT.available
TC0	FTFTT	teacher	MD	> 0
TC1	FFTFT	!*teacher*∧!*student*	HD	> 0
TC2	TFFFT	teacher	!MD∧!HD	> 0
TC3	TFTFT	teacher	HD	> 0
TC4	TFFTF	teacher	MD	<= 0

- Borrowing books on holidays (HD) or maintenance days (MD) by teacher

$$PC = (loginT.role(0) == \text{“}teacher\text{”}) \wedge (day == \text{“}HD\text{”} | day == \text{“}MD\text{”}) \wedge (bookT.available(1) > 0)$$

- Borrowing books on working days (WD) by student

$$PC = (loginT.role(0) == \text{“}student\text{”}) \wedge (day == \text{“}WD\text{”}) \wedge (bookT.available(1) > 0)$$

- Borrowing books on holidays (HD) or maintenance days (MD) by student

$$PC = (loginT.role(0) == \text{“}student\text{”}) \wedge (day == \text{“}HD\text{”} | day == \text{“}MD\text{”}) \wedge (bookT.available(1) > 0)$$

Table 15 and 16 show the selected tests for first and second rule, respectively. After solving the constraints by constraint solver, abstract test cases are generated.

Now, consider the mutation, in which the programmer forgets to check the availability of books before borrowing. TC3 in Table 15 can kill this mutation; however, test cases in Figure 9 does not have this capability because in all test cases, only an available book has been used.

Another evaluation criterion that we used is the level of code coverage. It should be noted that access control is only a part of code under test; therefore, we cannot use code coverage measurement tools. But we assume that each access control rule is a decision in SUT which has several conditions, then we use the following method to calculate the percentage of coverage.

- Condition Coverage (CC)
  1. Define two variables Condition and CoveragedC with initial values of zero

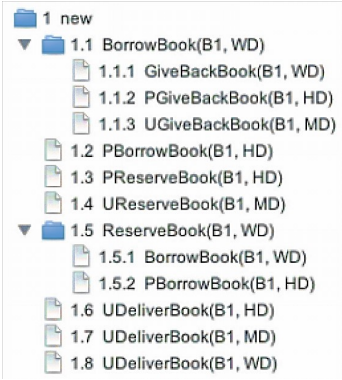

Figure 9. Partial Test Tree [2]

  2. Assign the total number of conditions to Condition variable
  3. Repeat steps 4 and 5 for each condition
  4. Increment CoveragedC, if there is at least one test case which contains positive condition
  5. Increment CoveragedC, if there is at least one test case which contains negative condition
  6. Compute condition coverage percentage using Formula 7.

$$CC = (CoveragedC/(Condition \times 2)) \times 100 \quad (7)$$

- Decision Coverage (DC)
  1. Define two variables Decision and CoveragedD with initial values of zero
  2. Assign the total number of decisions to Decision variable
  3. Repeat steps 4 and 5 for each decision
  4. Increment CoveragedD, if there is at least one test case which contains positive decision
  5. Increment CoveragedD, if there is at least one test case which contains negative decision
  6. Compute decision coverage percentage using Formula 8.

$$DC = (CoveragedD/(Decision \times 2)) \times 100 \quad (8)$$

- Condition/Decision Coverage (C/DC)
  1. Compute condition/decision coverage percentage using Formula 9.

$$CDC = ((coveragedC + coveragedD)/((condition \times 2) + (decision \times 2))) \times 100 \quad (9)$$

- Modified Condition/Decision Coverage (MC/DC)
  1. Generate the test suite with modified condition/decision coverage criterion based on the method which is presented in step 5 of Section 5.2. Then, determine the percentage of those test cases which are generated by ACP Test Generator.

The percentage of different coverage criteria for LMS is shown in Figure 10. As indicated in the figure,

Archive of SID

www.SID.ir

the test cases that are generated for LMS cover most part of SUT. Because of using M2C/DC to generate alternative tests instead of insoluble tests and also generating appropriate test data with the help of database constraint solver, the generated tests are good and covers about 95% of the code.

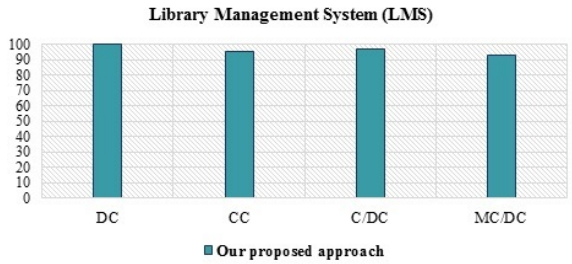

Figure 10. Code Coverage Percentage in the Proposed Approach

7 Conclusion and Future Work

We proposed a new approach to automate testing of access control. This approach is implemented as a tool that is called ACP Test Generator. Producing negative tests along with positive ones increases the possibility of detecting unknown faults. Due to use of constraint solver which is capable of solving database constraints as well as other kinds of constraints, actual data for the test are provided. In this tool, not only unit tests but also integration and system tests can be produced. Results of evaluation of our approach show that this method is able to detect about 95% of access control failures for LMS. In our proposed tool, the generation of executable tests is limited to local form applications. Therefore, in future works we intend to improve this shortcoming so that we are able to generate tests for checking the access control of other kinds of systems, e.g., web applications. Also, it is better to use a dynamic modeling method to model the behavior of SUT. In fact, by using such model, in the test path variable can be dynamic and their value can be changed during passing of the test path, if it is required.

References

[1] OASIS. extensible access control markup language (xacml) version 3.0. docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html . Accessed 6/22/2013.

[2] Dianxiang Xu, Lijo Thomas, Michael Kent, Tejeddine Mouelhi, and Yves Le Traon. A model-based approach to automated testing of access control policies. In *Proceedings of the* 17^{th} *ACM symposium on Access Control Models and Technologies*, pages 209–218. ACM, 2012.

[3] Hasan Qunoo and Mark Ryan. X-policy: Knowledge-based verification tool for dynamic access control policies. *Security & Its Applications*, 7(2):89–104, March 2013.

[4] Karthick Jayaraman, Vijay Ganesh, Mahesh Tripunitara, Martin Rinard, and Steve Chapin. Automatic error finding in access-control policies. In 18^{th} *ACM conference on Computer and Communications Security*, pages 163–174. ACM, 2011.

[5] Graham Hughes and Tevfik Bultan. Automated verification of access control policies using a sat solver. *Software Tools for Technology Transfer (STTT)*, 10(6):503–520, 2008.

[6] Alexander Pretschner Mark Utting and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, August 2012.

[7] Microsoft. Microsoft solver foundation. http://flo.livezon.com/2013/02/how-and-why-use-microsoft-solver-foundation-for-net. Accessed 6/5/2016.

[8] Ammar Masood, Arif Ghafoor, and Aditya Mathur. Scalable and effective test generation for access control systems that employ rbac policies. *Technical Report SERC-TR-285, Purdue University*, 2006.

[9] Ammar Masood, Arif Ghafoor, and Aditya Mathur. Conformance testing of temporal role-based access control systems. *IEEE Transactions on Dependable and Secure Computing*, 7:144–158, 2010.

[10] Wissam Mallouli, Jean Marie Orset, Ana Cavalli, Nora Cuppens, and Frederic Cuppens. A formal approach for testing security rules. In *Proceedings of the* 12^{th} *ACM symposium on Access control models and technologies*, pages 127–132. ACM, 2007.

[11] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. Transforming and selecting functional test cases for security policy testing. In *Proceedings of the* 2^{nd} *international conference on Software Testing, Verification, and Validation (ICST 09)*, pages 171–180. IEEE, 2009.

[12] Jacques Julliand, Pierre Alain Masson, and Regis Tissot. Generating security tests in addition to functional tests. In *Proceedings of the* 3^{rd} *international workshop on Automation of software test, New York, NY, USA*, pages 41–44. ACM, 2008.

[13] Anas Abou El Kalam, RE Baida, Philippe Balbiani, Salem Benferhat, Fredric Cuppens, Yves Deswarte, Alexandre Miege, Claire Saurel, and Gilles Trouessin. Organization based access control. In *Proceedings of IEEE* 4^{th} *International Workshop on Policies for Distributed Systems*

Archive of SID

www.SID.ir

and Networks (POLICY 2003), Lake Como, Italy, pages 120–131. IEEE, 2003.

[14] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing security policies: going beyond functional testing. In *The* 18^{th} *IEEE International Symposium on Software Reliability (ISSRE 07)*, pages 93–102. IEEE, 2007.

[15] Alexander Pretschner, Tejeddine Mouelhi, and Yves Le Traon. Model-based tests for access control policies. In 1^{st} *International Conference on Software Testing, Verification, and Validation (ICST 08)*, pages 338–347. IEEE, 2008.

[16] Keqin Li, Laurent Mounier, and Roland Groz. Test generation from security policies specified in or-bac. In 31^{st} *Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 2, pages 255–260. IEEE, 2007.

[17] Evan Martin and Tao Xie. Automated test generation for access control policies via change-impact analysis. In *Proceedings of the* 3^{rd} *International Workshop on Software Engineering for Secure Systems*, pages 5–11. IEEE, 2007.

[18] Antonia Bertolino, Marianne Busch, Said Daoudagh, Francesca Lonetti, and Eda Marchetti. A toolchain for designing and testing access control policies. In *Engineering Secure Future Internet Services and Systems*, volume 8431, pages 266–286. Springer, 2014.

[19] Hongxin Hu and GailJoon Ahn. Enabling verification and conformance testing for access control model. In *Proceedings of the* 13^{th} *ACM symposium on Access control models and technologies*, pages 195–204. ACM, 2008.

[20] Hongxin Hu and GailJoon Ahn. Alloy: a lightweight object modelling notation. 11(2):256–290, April 2002.

[21] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In 11^{th} *International Conference on Model Driven Engineering Languages and Systems (MoDELS 08), Toulouse, France*, pages 537–552. Springer, 2008.

[22] Evan Martin, Tao Xie, and Ting Yu. Defining and measuring policy coverage in testing access control policies. In *Proceedings of the* 8^{th} *International Conference on Information and Communications Security*, pages 139–158. Springer, 2006.

[23] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering*, 9(5):193–200, 1994.

[24] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. A practical tutorial on modified condition/decision coverage. *Technical Report SERC-TR-285, Purdue*, 2001.

[25] James A Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software*, 29(3):195–209, 2003.

[26] Marzieh Safarzadeh, Behrouz Tork Ladani, and Bahman Zamani. Improvement of modified condition/decision coverage criterion in model based testing technique. In *7th international conference on information and knowledge technologies*. Iran 2015. (In persian).

[27] Gins DOlera. Umu-xacml-editor v1.3.2. http://umu-xacmleditor.sourceforge.net/. Accessed 9/10/2013.

[28] Jan Tretmans and Ed Brinksma. Torx: Automated model-based testing. 1^{st} *European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, 2003.

[29] Evan Martin, JeeHyun Hwang, Tao Xie, and Vincent Hu. Assessing quality of policy properties in verification of access control policies. In *Proceedings Annual Computer Security Applications Conference (ACSAC)*, pages 163–172. IEEE, 2008.

[30] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, and Eda Marchetti. Xacmut: Xacml 2.0 mutants generator. In 6^{th} *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 28–33. IEEE, 2013.

Marzieh Safarzadeh received her B.S. and M.S degrees in Computer Software Engineering from University of Isfahan, Isfahan, Iran, in 2011 and 2016, respectively. Her research interests include software testing and software security.

Mahboubeh Taghizadeh received her B.S. in Computer Engineering from University of Shiraz, Shiraz, Iran, in 2011, and M.S. in Computer Software Engineering from University of Isfahan, Isfahan, Iran in 2014. Her research interests include software testing and model-driven software engineering (MDSE).

Archive of SID

www.SID.ir

Bahman Zamani received his B.S. from the University of Isfahan, Isfahan, Iran, in 1991, and M.S. from the Sharif University of Technology, Tehran, Iran in 1997, both in Computer Engineering (Software). He obtained his Ph.D. degree in Computer Science from Concordia University, Montreal, QC, Canada in 2009. From 1998 to 2003, he was a researcher and faculty member of the Iranian Research Organization for Science and Technology (IROST) - Isfahan branch. Dr. Zamani joined the Department of Computer Engineering at the University of Isfahan in 2009, as an assistant professor. His main research interest is model-driven software engineering (MDSE). He is the founder and director of MDSE research group at University of Isfahan.

Behrouz Tork Ladani received a B.S. in Software Engineering from University of Isfahan, Isfahan, Iran, in 1996, and M.S. in Software Engineering from AmirKabir University of Technology, Tehran, Iran, in 1998, and a Ph.D in Computer Engineering from Tarbiat-Modarres University, Tehran, Iran, in 2005. He is currently an associate professor and head of Department of Software Engineering in University of Isfahan. He is also the managing editor of Journal of Computing and Security. Dr. Ladani is also member of the Iranian Society of Cryptology (ISC). His research interests include software security, cryptographic protocols, formal verification, and computational trust.

Archive of SID

www.SID.ir

The ISC Int'l Journal of Information Security
ISeCure

January 2017, Volume 9, Number 1 (p. 98)
http://www.isecure-journal.org

Persian Abstract

مولّد خودکار موارد آزمون برای ارزیابی پیاده‌سازی سیاست‌های کنترل دسترسی

مرضیه صفرزاده[۱]، محبوبه تقی‌زاده[۱]، بهمن زمانی[۲] و بهروز ترک لادانی[۱]

[۱] دانشکده مهندسی کامپیوتر، دانشگاه اصفهان، اصفهان، ایران

[۲] گروه پژوهشی مهندسی نرم‌افزار مدل‌رانده، دانشکده مهندسی کامپیوتر، دانشگاه اصفهان، اصفهان، ایران

یکی از مهم‌ترین نیازمندی‌های سیستم‌های نرم‌افزاری امنیت است. از جمله اصلی‌ترین نیازمندی‌های مرتبط با تأمین امنیت، کنترل دسترسی است که گاهی از آن به‌عنوان قلب امنیت یاد می‌شود. هدف اصلی سیاست‌های کنترل دسترسی حفاظت از منابع سیستم در برابر دسترسی‌های غیرمجاز است. با توجه به این‌که وجود خطا در پیاده‌سازی سیاست‌های کنترل دسترسی ممکن است نتایج نامطلوبی در بر داشته باشد، لازم است که با روش‌هایی از درستی پیاده‌سازی این سیاست‌ها اطمینان حاصل نمود و بهتر است که این روش‌ها به‌صورت خودکار باشند. در واقع روش‌های خودکار، قابلیت اطمینان بیشتر و سرعت بالاتری دارند. تاکنون نیز پژوهش‌های متعددی در زمینه‌ی خودکارسازی آزمون توصیف سیاست‌های کنترل دسترسی انجام شده است، اما بیشتر این تحقیقات مربوط به مرحله‌ی طراحی سیاست‌های کنترل دسترسی است و تعداد پژوهش‌های انجام‌شده در زمینه‌ی پیاده‌سازی سیاست‌ها بسیار کم است. به‌علاوه، به‌دلیل آن‌که کنترل دسترسی در دسته‌ی نیازمندی‌های غیرکارکردی سیستم قرار می‌گیرد، نمی‌توان آن را با روش‌های معمول و همراه با نیازمندی‌های کارکردی مورد آزمون قرار داد. برای رفع این مشکل، در این مقاله یک روش خودکار برای آزمون پیاده‌سازی سیاست‌های کنترل دسترسی سیستم‌های نرم‌افزاری ارائه شده است. این روش که مبتنی بر مدل است، قادر به استخراج موارد آزمون برای ارزیابی سیاست‌های کنترل دسترسی سیستم تحت آزمون است. برای تولید خودکار موارد آزمون، از ترکیبی از مدل رفتاری سیستم و توصیف سیاست‌های کنترل دسترسی، که با زبان XACML نوشته شده است، استفاده می‌شود. نتایج نشان می‌دهد که روش پیشنهادی، قادر به کشف خطاهای پیاده‌سازی سیاست‌های کنترل دسترسی و پوشش کدهای مربوطه است.

واژه‌های کلیدی: کنترل دسترسی، خودکارسازی آزمون، آزمون مبتنی بر مدل، پیاده‌سازی سیاست‌های کنترل دسترسی، XACML.

Archive of SID

www.SID.ir