Model-Based Tests for Access Control Policies

Alexander Pretschner, Tejeddine Mouelhi, Yves Le Traon

2008 International Conference on Software Testing, Verification, and Validation

Model-Based Tests for Access Control Policies[1]

Alexander Pretschner[a], Tejeddine Mouelhi[b], Yves Le Traon[b]
[a]*ETH Zurich, Switzerland*
[b]*IT-TELECOM Bretagne, Cesson Sévigné, France*

Abstract

We present a model-based approach to testing access control requirements. By using combinatorial testing, we first automatically generate test cases from and without access control policies—i.e., the model—and assess the effectiveness of the test suites by means of mutation testing. We also compare them to purely random tests. For some of the investigated strategies, non-random tests kill considerably more mutants than the same number of random tests. Since we rely on policies only, no information on the application is required at this stage. As a consequence, our methodology applies to arbitrary implementations of the policy decision points.

1. Introduction

The amount of digital data that is exchanged between individuals, businesses, and administrations continues to increase, and so does the number of applications that manage this kind of data. Fueled by the existence of many interfaces to such applications, including the Internet, there is a general agreement that the security of these applications is becoming ever more relevant.

One major security concern is control over the access to resources. Respective requirements are stipulated in so-called access control policies. An application for which such a policy is *specified* is supposed to also *implement* that policy. The part of the application that implements the policy is normally called the policy decision point (PDP). The PDP is a conceptual entity and can be implemented in many different forms, e.g., by dedicated software components that are called before each resource access, or by spreading the respective program logic over the code. An obvious problem then is to make sure that a PDP implements a given access control policy.

One approach to solving this problem is by design. Basin et al., for instance, propose to develop systems on the grounds of two related specifications, one design model and one access control model [3]. For different infrastructures, the respective code is generated. Assuming the correctness of code generators, this ensures that the application's PDP implements the policy. Legacy systems and systems that are built in development processes that do not rely on models require a different approach. One generally applicable strategy is that by analysis and, if necessary, subsequent modification of the PDP.

Problem Statement. In this paper, we study how to test access control requirements in arbitrary applications. This question entails three sub-problems, namely how to generate tests, how to assess their quality, and how to run them on an actual system.

Solution and Empirical Results. We propose to proceed in two steps. In a first step, we generate abstract tests (test targets, §3.1). Test targets represent classes of actual requests. They are generated (1) regardless of any policy, i.e., by only taking into account roles, permissions, and contexts; (2) by considering all the rules in a given policy, that is, the model; and (3) completely at random. Relying on a fault model that considers incorrect decisions of the PDP to be a consequence of *n*-wise interactions of rule elements, we use combinatorial testing for strategies (1) and (2) to automatically generate a test suite of manageable size. In a second step, we show how to derive actual tests (code) from the abstract test targets. Because this involves application-specific program logic and usually also a particular state of the application, this can in general not be fully automated. We discuss the issue of automation.

As to the problem of measuring the quality of the generated tests, we use mutation analysis to show that the tests that only use domain knowledge (that is, no policy-related information) perform as good as the same number of random tests. We also show that some of the generation strategies that use pair-wise testing and

[1] pretscha@inf.ethz.ch, {tejeddine.mouelhi, yves.letraon}@telecom-bretagne.eu. This work was done while the first author was on sabbatical leave at ENST Bretagne. Financial and organizational support is gratefully acknowledged.

978-0-7695-3127-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ICST.2008.44

that rely on the policy do indeed perform better than random tests, and that they do so with comparably few tests. This provides first experimental evidence that combinatorial testing of access control policies is a promising approach. Our generation procedure is stable and not subject to random influences, as suggested by a thirty-fold repetition of our experiments.

Contribution. The first contribution of this paper is a methodology and technology for automatically generating tests both without and on the grounds of access control policies. The second contribution is an experimental assessment of the effectiveness of the generated tests. We consider a special extension of role-based access control in this paper, but our methodology naturally generalizes to other access control models.

Overview. The remainder of this paper is organized as follows. We give the necessary background in terms of access control, combinatorial testing, and mutation testing in §2. Our test methodology, consisting of generating both abstract test targets and deriving concrete tests is the subject of §3. We describe experiments for assessing the quality of the generated test suites in §4, and also discuss the results there. After putting our work in context in §5, we conclude in §6.

2. Background

This section sets the scene. We present our running example, describe the access control model used in this paper, and provide the essence of combinatorial testing for test case generation as well as of mutation testing for test case evaluation.

2.1 Running Example

As a running example, we re-use a library management system (LMS, [12]). While it is not necessary to describe all constraints here, access conditions like the following are typical for many systems: books can be borrowed and returned by the users of the library on working days. When the library is closed, users can not borrow books. When a book is already borrowed, a user can make a reservation. When the book is available, the user can borrow it. The LMS distinguishes three types of users: public users who can borrow 5 books for 3 weeks, students (10 books for 3 weeks), and teachers who can borrow 10 books for 2 months.

The accounts of in the LMS are managed by an administrator. Books in the library are managed by a secretary who can order books and enter them into the LMS when they are delivered, repair damaged books, etc. Finally, the director of the library has the same access rights as the secretary and he can also consult the accounts of the employees.

2.2 Access Control

Access control is concerned with the protection of system resources against unauthorized access. In particular, it defines a process by which the use of system resources (1) is regulated according to an access control policy; and (2) is exclusively permitted by authorized entities (users, programs, processes, or other systems) according to that policy. While a multitude of access control strategies and models has been developed [15], we will concentrate on an extension of the so-called role-based access control (RBAC, [14]) model. Roughly, RBAC associates roles (sets of subjects) with permissions (pairs consisting of activities and resources).

In this paper, we use a simple extension of RBAC with contexts and strategies for conflict resolution. The model relies on hierarchical roles, hierarchical permissions that associate activities with resources, and hierarchical contexts. Requests consist of a subject instance, a resource instance, an action instance the subject wants to exercise on the resource, and a context instance. The PDP checks if a request corresponds to a rule such that the requester is an element of a descendant of that rule's role, the action is an element of a descendant of that rule's action, etc.

Definitions and Syntax. The policy meta model is depicted in Figure 1. Our domain consists of *role names*, *permission names* that are pairs of *activity names* and *resource names*, and *context names*. Roles associate role names with finite sets of *subject instances*. Similarly, activities associate activity names with finite sets of *action instances,* and resources associate resource names with finite sets of *resource instances*. Contexts associate context names with finite sets of *spatial and temporal constraints*. In the example of the LMS, the role names include borrower and administrator, and the subject instances include John Doe. The permission names include borrowBook (activity name: borrow, resource name: book), and permission instances include the method name borrow() as well as the electronic representation of a copy of Dumas's Count of Monte Christo. Context names include WorkingDays, and context instances include a representation of the fact that the current day is a working day.

Roles, permissions, and contexts are hierarchical, and we assume the existence of a dedicated root node for each hierarchy. In this way, for each policy, the universe of discourse is determined by the n:1 associations between policies and role names, permission names, and context names in Figure 1.

In the LMS, for instance, a BorrowerActivity is specialized by, among other things, BorrowBook and ReserveBook. Note that all nodes (as opposed to leaves only) can be mapped to non-empty sets of instances.

A *rule* is a quintuple consisting of a role, a permission, a context, a status flag indicating permission or prohibition, and a natural number that denotes a *priority*. In the LMS, an example of a rule is `prohibition(Borrower, BorrowBook, Holidays, 5)`. Finally, a *policy* is a set of rules together with a function that defines the hierarchies, and a default rule (see below).

Note that nothing prevents us from generating one type (role name, permission name, context name) per instance and add it to the hierarchies. In this case, the four instance classes in Figure 1could be omitted.

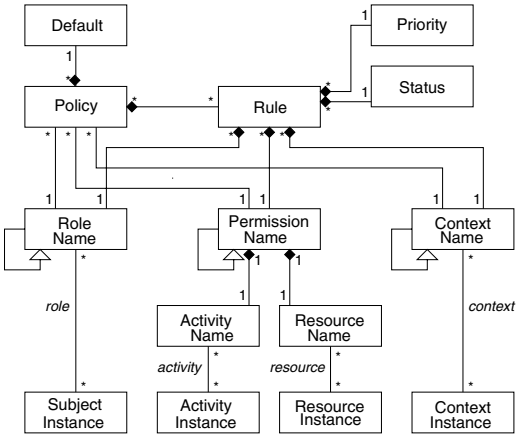

Figure 1

Semantics. The application of actions to resources is requested by a subject in a given context. Intuitively, for a request, the policy decision point checks if there is a rule that matches the request: the subject instance of the request is part of the role or its subroles defined in the rule; the (activity, resource) pair is part of the permission of the rule, and the context instance is part of the context defined in the rule. In that case, the rule is said to be *applicable* to the request. If there is no such rule, the PDP returns a default that is either permission or prohibition. Finally, if there is more than one rule, then the rule with the higher priority is chosen. If there are two applicable rules with the same priority and different status flags, then either a deny-override or permit-override strategy is applied. We omit the formal semantics here. While our approach to generating tests obviously relies on the semantics of a particular access control model when the oracle is consulted, it can nonetheless be applied to *any* access control model.

2.3 Combinatorial Testing

When testing configurations that consist of multiple parameters, one fault model consists of assuming that failures are a consequence of the interaction of only two (three, four, …) rather than all parameters. Combinatorial testing [16,6] relies on precisely this idea and aims at defining test suites such that for each pair (triple, quadruple, …) of parameters, all pairs (triples, quadruples, …) of values for these parameters appear in one test case in the test suite. If the assumption (the fault model) can be justified, the eminent benefit of this strategy lies in the rather small size of the test suite. For instance, without any further constraints, the number of tests necessary for pair-wise testing is the sum of the number of all possible parameter values in the system.

2.4 Mutation Testing

The original purpose of mutation testing [4] is to assess the quality of a test suite in terms of failure detection. The idea is to apply small syntactic changes to a program, e.g., replace a plus by a minus. The modified program is called a mutant. If a test suite is able to detect the deviation, the mutant is said to be killed, and the mutation score is the number of killed mutants divided by the number of mutants. If a mutant is not killed, this may be because the test suite is too weak or because the mutant is equivalent to the original program. This happens, for instance, if a +0 is replaced by a -0. As a consequence, mutation scores of 100% cannot be obtained. If mutation scores are used as test selection criteria [17], then this obviously poses a practical problem. Mutation testing assumes validity of the competent programmer's and the coupling hypotheses. The former assumes that programmers essentially introduce relatively simple faults, and the latter assumes a correlation between simple and more complex faults. Even though Andrews et al. have recently provided some empirical evidence that there is indeed such a correlation [2], both assumptions remain critical.

Mutation can also be applied not to the code but to access control policies [7,10]. If the code implementing a PDP can be configured by these policies, then test cases can be applied to PDPs that implement a mutated policy. By doing so, the quality of access control policy test cases can be assessed.

In this paper, we will use the mutation operators defined in earlier work [7]. Five operators are used to modify existing rules (replace prohibition by permission, replace permission by prohibition, replace role by any role, replace role by a descendant, replace context by different context, and replace the activity part of a permission by another permission). A sixth operator adds a new rule to the policy by picking a permission from the policy, and completing the rule by any role, any context, and any status flag. If a mutated rule is in conflict with other rules of the policy (which can be analyzed statically), then we assign it a priority that is higher than that of the other rules. Since the mutation

procedure operates on policies, and as such at a "semantic" level (as opposed to code), the first five operators generate no equivalent mutants, provided that there are no redundant rules in the original policy. Equivalent mutants generated by application of the sixth operator can easily be detected and removed.

3. Test Methodology

Recall that the problem we set out to solve is the generation of a set of test cases for assessing in how far an application implements a given access control policy. We use this section to describe the test methodology and the technologies involved. After providing the big picture, §3.1 describes two different ways of generating test targets; one regardless of any policy, the other depending on a policy. In §3.2 we explain how to concretize abstract test targets into concrete tests.

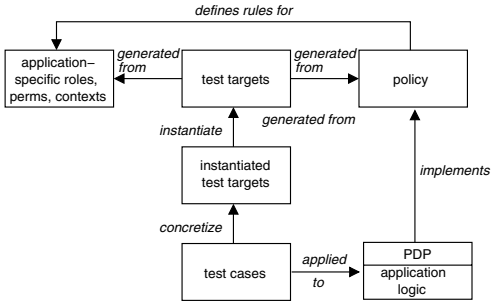

Figure 2

The overall process is depicted in Figure 2. As described in §2.2, policies are defined for a domain that consists of, among other things, application-specific *roles*, *permissions*, and *contexts*. That domain corresponds to the lower half of Figure 1. Different means are employed to generate *test targets* from policies and partial domain descriptions (§§3.1.1 and 3.1.2). A test target consists of a role name, a permission name, and a context name (§2.2). By picking a subject that is associated with the role name, a permission that is associated with the permission name, and a context that is associated with the context name, we obtain an *instantiated test target*. Test targets and instantiated test targets are defined at the level of abstraction of the policy and the partial domain description. They are hence too abstract to be directly executed on a real piece of software. *Test cases*, in contrast, are executed. Essentially, they are pieces of code that implement the instantiated test targets and that take into account the business logic of the application under test. The methodology that we define in this paper is about the generation of (instantiated) test targets and the subsequent manual derivation of test cases (see §6 for a discussion of the potential for automation).

Note that when generating both the PDP [3] and the test suite from the same policy, applying the tests to the application that contains the PDP is likely to reveal problems in the code generator rather than in the PDP plus associated application [13]. Our methodology hence applies to existing legacy systems and new systems without automatically generated PDPs [3].

3.1 Test Targets

The generation of test targets can be done in at least two ways. One is to generate them regardless of any policy. A second strategy takes into account the policy.

3.1.1 Ignoring the Policy

Roughly, the *generation of test targets without policy* only takes into account information on roles, permissions, contexts, and the respective hierarchies. Essentially, combinatorial testing is applied to all nodes of the three hierarchies (Figure 3).

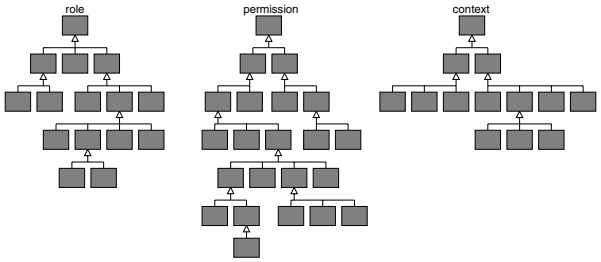

Figure 3

Instantiation. Random instantiation then takes place by picking one element of the respective instance sets. In case the number of instantiations of roles, permissions, or contexts is not prohibitive, combinatorial testing can even be applied at the level of instances (see the comment above on introducing one role name, permission name, or context name per respective instance).

3.1.2 Using the Policy

The *generation of test targets from access control policies*, in contrast, proceeds as follows. We will consider each single rule in turn. The part of the rule that is relevant for test case generation is the triple (r, p, c) that consists of a role name, a permission name, and a context name (priorities are required for the oracle). Each element of that triple is a part of one of the hierarchies defined in Figure 1, and they correspond to boxes with thick borders in Figure 4. Combinatorial testing is then employed to generate *n*-wise coverage

1. for all role names below r,
2. with all permission names below p,
3. with all context names below c.

All these nodes depicted as grey boxes in Figure 4, top. Since a rule is either a permission or a prohibition, it appears sensible to also test all those nodes which are not explicitly specified (this complementary set is depicted as grey boxes in Figure 4, bottom).

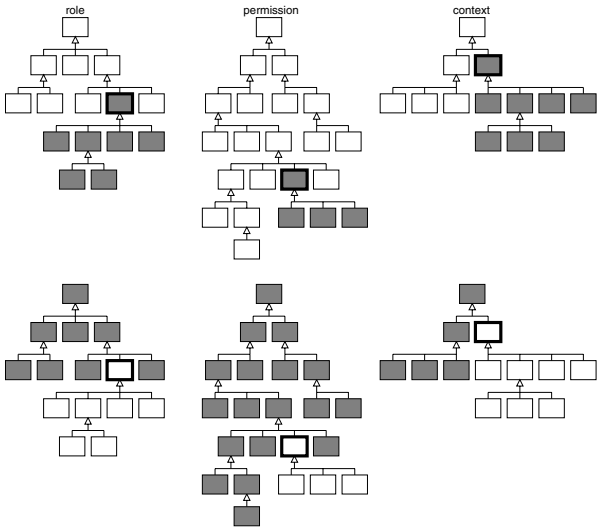

Figure 4

For the three elements of a rule, we can then specify whether or not the respective component of a test target should conform with the element of a rule. As an example, consider a rule that constrains role r, permission p, and context c (boxes with thick borders in Figure 5). The test targets that correspond to the grey boxes in that figure specify *a role different from r*, a *permission that derives from p*, and a *context that differs from c*. In sum, for each rule, this gives rise to eight combinations that can be used for the generation of test targets.

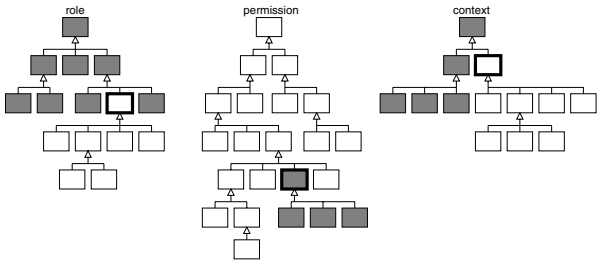

Figure 5

Instantiation. Similar to the first case, instances are then randomly chosen for each role name, permission name, and context name. Furthermore, also in accordance with the first case, if the number of instances is not prohibitive, combinatorial testing can even be applied to the respective instances. If, in this sense, one role name is introduced for each subject instance in the system, one permission name for each pair of action instance and resource instance, and one context name for each constraint, then the above procedure needs to be applied to the leaves of the hierarchies only, and no subsequent instantiation step is required.

3.1.3 Comparison

Not using the policy at all may, at first sight, appear surprising: if there is a policy, why wouldn't we use it for test case generation? Not using a policy seems reasonable because regardless of the policy, *all kinds of requests* should be tested, and these only depend on the roles, permissions, and contexts, but not on the policies. The essential difference between the two strategies lies in the number of tests that correspond to existing rules. In most cases, some permissions or prohibitions will not be defined explicitly in the policy. Respective decisions are then taken by referring to the default rule. In the first approach, corresponding test targets may be generated more or less at random. In contrast, many test targets are explicitly generated by the second approach for these implicit rules. Note that because our policies can express both permissions and prohibitions even within one policy, the two approaches do not differ in "better" testing permissions than prohibitions nor vice versa.

3.2 Concrete Test Cases

Concrete tests differ from abstract and instantiated test targets in that both targets do not take into account the application logic at all. The problem then is that specific actions cannot be applied to specific resources in all states. For instance, a book cannot be returned before it has been borrowed. When concretizing test targets, this information must be taken into account. The information, however, is likely available in the requirements documents, in the form of sequence diagrams or similar descriptions. The derivation of test cases then consists of writing a preamble that puts the system into a state where the access rule is applicable as far as the state of the application is concerned. We are currently working on generating the respective code from sequence diagrams, and integrating it with the generated tests, but this is not the subject of this paper and immaterial to our results that relate to test generation strategies. From a practical perspective, however, the automation of such procedures is of course highly useful—testing becomes a push-button technology.

The following example illustrates concretization. Consider a rule `prohibition(borrower, return_book, maintenanceDay, 4)` for the LMS. For any of the above generation strategies, assume that the **test target** prescribes a role `borrower`, a permission `return_book`, and a context `maintenanceDay`. A possible **instanti-**

ated test target is `prohibition(std1, book1, maintenanceDay, 4)`. This however, still is too abstract to be executed. Concretization leads the following code that consists of a preamble that puts the system into a desired state, execution of the actual test, and the evaluation of the test.

```
// test data initialization
// log in a student
std1 = userService.logUser("login1", "pwd1");
// create a book
book1 = new Book("book title");
// activity
// book needs to be borrowed before returned
borrowBookForStudent(student1,book1);
// context
contextMan-
ager.setTemporalContext(maintenaceDay);
// security test
// run test
try {
 returnBookForStudent(std1,book1)
 // security oracle
 // SecurityPolicyViolationException is ex-
pected because an SP rules is not respected
 // test failure
 fail(" SecurityPolicyViolationException ex-
pected,   returnBookForStudent with student =
" + std1 + " and book = " + book1);
}
catch( SecurityPolicyViolationException e) {
 // ok security test succeeded log info
 log.info("test success for rule :   prohibi-
tion(borrower,return_book,maintenanceDay)");
}
```

3.3 Implementation

With the exception of the generation of Java code, our test generation procedure is fully automated. The system takes as input a policy and the respective domain description (roles, permissions, contexts) together with the strategies to be applied. For each strategy, it returns the generated test targets. *N*-wise test generation—we concentrate on pairs in this paper—is performed by a tool that is publicly available at www.burtleburtle.net/bob/math/ jenny.html.

In terms of the experiments, mutants of the policy (not any respective implementing code—this would almost certainly lead to equivalent mutants) are generated by the procedure described in §2.4. This procedure has been implemented as part of earlier work [7, 12]. The mutated polices are translated into Prolog code (motorbac.sourceforge.net), and this code is used as the executable oracle.

4. Experiments

The question that we study is concerned with the quality of tests generated by the different above strategies. We consider the following case studies that have all been used in various other research projects. Both policies and systems were designed and implemented independently of the present study.

The **library management system** has already been described in §2.1. Its policy is defined via 41 rules on 7 roles, 10 permissions, and 4 contexts.

We also consider the access control policy in a **hospital** with different physicians and staff. It defines who manages the administrative tasks and who performs which medical tasks. Its policy is defined in 37 rules with 10 roles, 15 permissions, and 3 contexts. In contrast to the other systems, the hospital system was not implemented in an application. The policy, however, was defined in projects on security policies [1].

The **auction system** is inspired by the eBay auction sales management system. Sellers can create and sell products. Buyers place bids. During an auction, buyers and sellers can post comments. They can also give marks to the sellers and buyers. In addition, the application allows the personnel of the website to manage sales and user and personnel accounts. The system's policy is defined on 8 roles, 23 permissions and 4 contexts with 130 rules.

Finally, the **meeting management** system allows users to create and attend different types of meetings. It also allows the management to delegate and to transmit texts during the meeting. Meetings can be moderated by a specific user called the moderator and the number of attendees can be fixed. This application also allows the personnel to create and manage accounts for meeting users. Its policy consists of 106 rules defined over 8 roles, 18 permissions, and 3 contexts.

Procedure. We first generate tests with the two strategies described in §3.1. Our instantiation of *n*-wise testing is pair-wise testing. As a gold standard, we also generate tests purely at random. We count the number of generated test targets and measure the generation time. This time turns out to be negligible – less than a second – which is why we refrain from stating respective numbers here. In a second step, we assess the quality of the test suites by using mutation testing (§2.4).

Generation of test targets. The **first strategy** does not take into account policies but rather the domains only. Pair-wise testing is applied to roles, permissions, and contexts. In order to get a feeling for the influence of randomness when pairs are chosen, we perform the generation process thirty times for each strategy and for each case study.

The **second strategy** does take policies into account. For each of the eight sub-strategies, we generate test targets for each single rule (that is, each role-permission-context triple). We make sure that they are all independent from each other: we do not first generate two sets of role names, two sets of permission names and two sets of context names and then pick the

8 combinations from these buckets, but rather regenerate them in each turn. This experiment is also performed thirty times (hence 240 experiments per rule). In each of the thirty experiments, we remove redundant test targets from each of the eight test suites.

The **third strategy** also does not consider policies. However, test targets are chosen fully randomly, without pair-wise testing. In such a randomly generated test suite, we make sure there are no identical tests. This generation strategy is also applied thirty times.

Assessment of tests. The quality assessment proceeds as follows (we only consider the level of test targets here). We consider each of the four fixed policies in turn. We apply the first five mutation operators to every rule (§2.4). Replacing a prohibition by a permission and vice versa yields one mutated policy. When applying the other mutation operators, we generate all possible mutants rather than just one. For instance, when a role is replaced in a rule, we replace it by all other possible roles in the system rather than by just one other role. Furthermore, the application of the addition operator yields an entire set of mutated policies. Details of the mutation operators are described elsewhere [7, 12]. The generated test suites are then applied to all mutants, and we record the mutation score.

Observations and Discussion. Figure 6 to Figure 9 show box-whisker diagrams of the mutation scores for the four case studies. **Strategy 1** (no policy, only domain considered, pair-wise coverage) is labeled *no policy (N).* The test targets generated for **strategy 3** are labeled *random (N)*. In both cases, *N* denotes the number of generated test targets.

The eight sub-strategies of **strategy 2** are labeled *policy 000 (N_0)*, *policy 001 (N_1)*, …, *policy 111 (N_7)*. The N_i correspond to the number of test targets generated for each sub-strategy. The binary number rpc in the labels encodes the chosen sub-strategy: Let r denote whether, for each rule, roles are chosen from the specializations of the role provided in that rule (1), or from the complement of that set (0). Let p denote whether, for each rule, permissions are chosen from the specializations of the provided permission (1) or from the complement of that set (0). Finally, let c denote whether, for each rule, contexts are chosen from the specializations of the provided context (1) or from the complement of that set (0). Then, for instance, rpc=010 (Figure 5) corresponds to: (a) role *not* below the role that is specified in the rule, (b) permission from the set of all permissions that are below the permission provided in the rule, and (c) a context chosen from the set of contexts that are *not* sub-contexts of that provided in the rule. In the following, we will denote sub-strategy xyz (x,y,z are 1 or 0) of strategy 2 by 2.xyz.

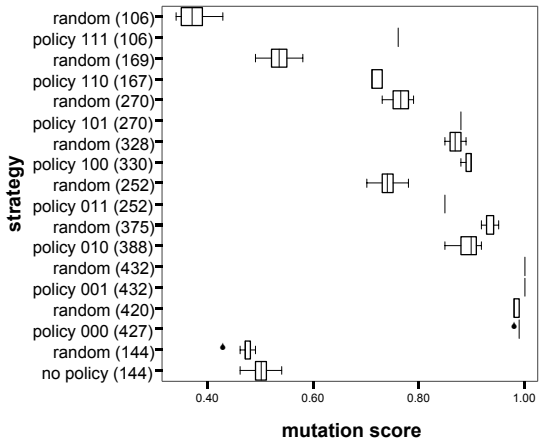

Figure 6: Meeting System (432 exhaustive tests)

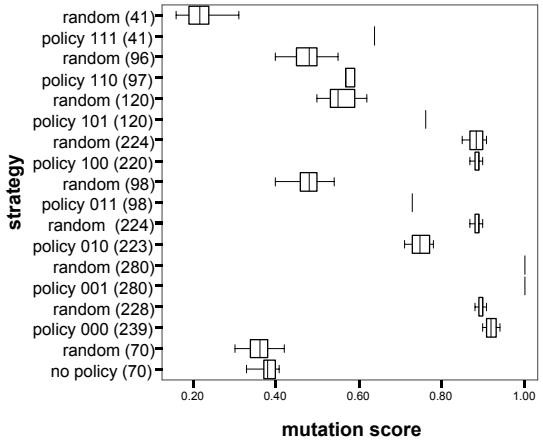

Figure 7: Library System (280 exhaustive tests)

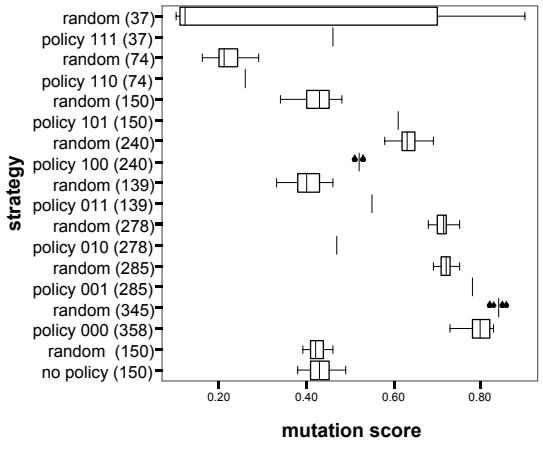

Figure 8: Hospital System (450 exhaustive tests)

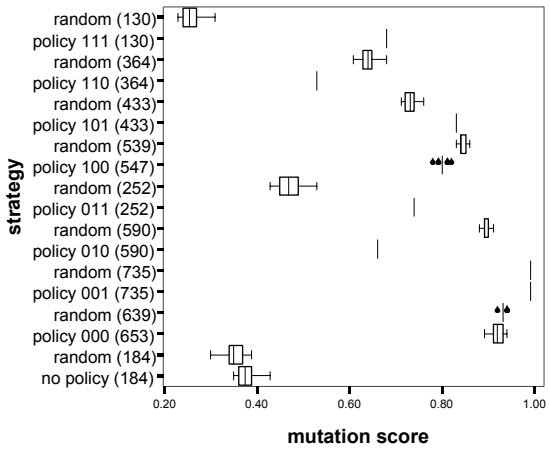

Figure 9: Auction System (736 exhaustive tests)

Overall, the variance of the mutation scores in all experiments is small. The number of tests for comparable strategies may slightly differ (e.g., for strategy 2.000 in all four experiments). This is a result of randomness in the pair-wise testing approach. However, as the variance is small, we may ignore this effect here.

Furthermore, the eight sub-strategies of strategy 2 result in different numbers of test targets. This is a consequence of the role, permission, and context hierarchies: the number of nodes above or below a node need not be identical. Consequently, pair-wise testing is applied to variables with different domains.

Strategy 1 turns out to be as good as random testing, and, with the exception of the hospital system, worse than all sub-strategies of strategy 2. The number of tests is comparatively low: an average 29% of the cardinality of the exhaustive test set. However, because of better strategies, pair-wise testing that does not take into account a policy is, in terms of our assessment criterion, a strategy that can safely be discarded. This result indicates that taking into account an access control policy when generating tests on the grounds of pair-wise testing is highly advisable. Because of the existence of default rules, we would have argued that, *in general*, *all requests* (or test targets) are equally likely to detect faults. The above is hence a result that we did not expect and that is probably due to the use of mutation testing for assessing tests.

With the exception of the hospital system, **strategy 2.001** yields exhaustive tests. This will, in general, not be the case for all policies. In our examples, however, hierarchies are rather flat (which means that the negation of roles and permissions yields, for every rule that is used for test target generation, almost all other roles and permissions, respectively). At the same time, the number of contexts is small. Taken together, this leads to a high probability of generating exhaustive tests (note that this is only almost the case for the auction system). The exception of the hospital system is explained by its policy and the mutant generator: for one role, there are no rules, and in contrast to the test target generator, the mutation generator creates mutants only for those elements that occur in any rule (and that are not only provided in the policy's domain description).

When comparing different strategies to random tests, we get the following aggregated results (bold rows indicate superiority of the respective strategy):

Strategy	Better than random	Equal	Worse
1	0	4	0
2.000	1	2	1
2.001	1	3	0
2.010	0	0	4
2.011	**4**	**0**	**0**
2.100	0	2	2
2.101	**4**	**0**	**0**
2.110	2	1	1
2.111	**3**	**1**	**0**

Strategies 2.011, 2.101, and 2.111 perform better than random tests**; strategies 2.000, 2.001, and 2.110** perform approximately like random testing; and **strategies 2.010 and 2.100** perform worse than random testing.

However, we also have to consider the number of tests that achieve these results. The following table shows the relative number (x 100) of test cases that achieved the mutation scores, i.e., number of tests divided by the cardinality of the exhaustive test set.

000	**001**	**010**	**011**	**100**	**101**	**110**	**111**	**System**
99	100	90	58	76	63	39	25	Meeting
85	100	80	35	79	43	34	15	Library
80	63	62	31	53	33	16	8	Hospital
89	100	80	34	74	59	50	18	Auction
88	**91**	**78**	**40**	**71**	**50**	**35**	**17**	**Average**

1. Strategies that take into account positive context definitions (i.e., do not make use of the complement set: strategies 2.001, 2.011, 2.101, 2.111) provide better or identical results than random tests. Out of these, 2.011 and 2.101 perform their results with 40%-50% of the tests, and 2.111 with only 17%.
2. Conversely, strategies that take into account the complement set of the contexts (strategies 2.000, 2.010, 2.100, 2.110) provide worse results than random tests. Three of them require 71%-88% of the tests; only strategy 2.110 requires a mere 35%.
3. Strategies that do not negate both roles and permissions at the same time and that negate contexts (strategies 2.010 and 2.100) perform worse than

random tests. This is with a comparably high number of tests: 71-78%.

4. With the exception of strategy 2.111, strategies that either negate or do not negate both roles and permissions perform as good as random tests (2.000, 2.001, 2.110). Strategies 2.000 and 2.001 require 88%-91% of the tests while strategy 2.110 (that may also be classified as performing better than random tests) only requires 35% of the tests.

In terms of the number of necessary tests—that tends to be relatively high when compared to the exhaustive test set—strategies 2.011, 2.101, 2.110, and 2.111 appear promising. In terms of the mutation scores, 2.001, 2.011, 2.101 and 2.111 appear promising. The intersection consists of **strategies 2.011, 2.101 and 2.111**.

When compared to random testing, **strategy 2.111** (positive role, positive permission, positive context) performs particularly well. Good mutation scores are obtained for rather small numbers of tests (17%). The number of test targets for strategy 2.111 equals the number of rules. The reason for this equivalence is the rather flat hierarchies in all example systems, and as it turns out (in hindsight), if a rule is defined for a non-leaf node of a hierarchy, then there are always complementary rules for all sub-nodes. Since redundant tests are removed in all strategies, this explains that exactly the number of rules is obtained. This result suggests that simply using one test per rule (possibly with exactly the elements that define the rule) provides surprisingly good results. We cannot really explain this finding, and we do not dare to generalize it. We will use it to scrutinize the mutation operators.

Summary. Because of the many degrees of freedom in the policy language of our examples (default rule, specification of both permissions and prohibitions, priorities), we think it is too early to draw generalized results in terms of which strategy is better. We also conjecture that this depends on the policies—ratio of permissions and prohibitions, and on the depth of the different hierarchies. However, we believe that our work, firstly, suggests that *using policies for test generation with pair-wise algorithms is preferable to only using domain knowledge* (roles, permissions, and contexts). Secondly, while it is too early to decide on the best strategies, *there are notable differences between the different strategies that use combinatorial testing*. This suggests that research into combinatorial testing for access control policies is a promising avenue of research. Thirdly, our generation procedure that uses pair-wise testing is stable in the sense that *it is not subject to random influences*, as suggested by the small variances in a thirty-fold repetition of our experiments.

Our conclusions are of course subject to several validity threats. The main problem with any generalization is obviously the small number of systems and the small number of roles, permissions, contexts, and rules for each policy. A set of four domain definitions with four hierarchies is unlikely to be representative of all possible hierarchies. Furthermore, as with all mutation testing, the relationship between mutants and actually occurring faults needs to be investigated [2].

5. Related Work

In earlier work, we have defined mutation operators and mutation-based coverage criteria for assessing tests for access control policies [7]. The concern of that work was not the automated generation of tests but rather their assessment. Martin and Xie [10] define a fault model and mutation operators for XACML policies. Their mutation operators may well lead to equivalent mutants which, because of the higher level of abstraction we use for mutation, can be avoided in our approach. This is crucial because with no equivalent mutants, we can measure the quality of a test suite in terms of failure detection. The respective mutation score is less significant when equivalent mutants exist. The same authors also generate tests [11] without relying on the ideas of combinatorial testing and measure, among other things, the mutation scores. These numbers are difficult to compare with ours because we work at a different level of abstraction and, as a consequence, employ mutation operators at the level of rules rather than at the level of XACML code.

Among other things, a tool [5] developed by Fisler et al. performs change-impact analysis on a restricted set of XACML policies. Given an original and a modified policy, the tool proposes requests that lead to different decisions for two PDPs that implement the two policies. This tool can be used for test case generation on the grounds of XACML which is not the level of abstraction that we target in this paper.

Several researchers have generated tests from access control policies given by various forms of state machines [8,9]. This work does not contain an evaluation of the generated tests.

6. Conclusions

We have presented a methodology for testing access control requirements, technology for automatically generating test targets, and an evaluation of the generation procedure. In sum, our results suggest, firstly, that using a policy for test generation is beneficial when compared to only using domain knowledge, i.e., the roles, permissions, and contexts (cf. the related controversy on partition-based testing). Given that policies are usually equipped with a default rule and that hence any request to a system is relevant for testing, this

comes as a surprise and requires further investigations. Secondly, we find that pair-wise testing yields, for some strategies, tests with higher mutation scores than purely random tests. Because we have considered only four examples of moderate size, we refrain from drawing generalized conclusions as to which strategy is the best. However, we believe that our results provide evidence that combinatorial testing for access control policies is a promising avenue of future research.

In addition to overcoming the threats to validity discussed in §4, there are many possibilities for future work. We deliberately only use the policies, and not any code, for generating and assessing tests (generation is simple and the assessment does not run into the problem of equivalent mutants). The concretization of the tests, in contrast, is a manual process today. We believe in the potential of automatically generating this code from requirements in the form of sequence diagrams. However, by their very nature, these sequence diagrams are likely to only modify one small part of the system's state space. Yet, tests (that is, requests for accessing a resource) should be run in many different states. This suggests that one test target should be concretized into *many* concrete tests. This is currently not considered in our approach. Because of the separation of test target generation from all application logics, once the problem of putting the system in "interesting" states is solved, however, it seems easy to integrate this with our test target generation procedure.

In our examples, the exhaustive number of test targets is rather low. Assuming that automation for generating and executing concrete tests is available, one might ask why not to perform exhaustive testing (exhaustive at the level of the policy, not the code). Test case minimization nonetheless reduces effort, and for huge policies, exhaustive testing may not be possible. The better of our generation procedures generate in-between 17% and 50% of the number of exhaustive tests. We do not know these numbers for larger policies.

In terms of the size of the test suite, the benefits of combinatorial testing become increasingly apparent if more than three parameters (roles, permissions, contexts) have to be taken into account. ORBAC [1], for instance, explicitly splits permissions into activities and resources, and adds the dimension of organizations, which gives a total of five parameters. We are currently conducting a respective study. The concretization of tests, however, is not entirely trivial because many activities are not defined for all resources.

If the reduction of test suites turns out to result in larger gains (say, of an order of magnitude), then one interesting application of our approach would be the manual validation of policies.

Finally, we will have to better understand the suitability of our mutation operators, and how they relate to actual faults in policies and the respective implementations. In other words, the respective fault model for policies needs to be investigated. In this vein, among other things, we will have to look into the difference between first-order and second-order mutants.

References

1. El Kalam A., Benferhat S., Miège A., El Baida R., Cuppens F., Saurel C., Balbiani P., Deswarte Y., Trouessin G.: Organization Based Access Control. Proc. Policy, pp. 120-131, 2003
2. Andrews J., Briand L., Labiche Y.: Is Mutation an Appropriate Tool for Testing Experiments? Proc. ICSE, pp. 402-411, 2005
3. Basin D., Doser J., Lodderstedt T.: Model-driven security: From UML models to access control infrastructure. ACM TOSEM 15(1):39-91, 2006
4. DeMillo R., Lipton R., Sayward F.: Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer 11(4):34-41, 1978
5. Fisler K., Krishnamurthi S., Meyerovich L., Tschantz M.: Verification and Change-Impact Analysis of Access Control Policies. Proc. ICSE, pp. 196-205, 2005
6. Grindal M., Offutt J., Andler S.: Combination Testing Strategies: A Survey. Technical Report ISE-TR-04-05, George Mason University, 2004
7. Le Traon Y., Mouelhi T., Baudry B.: Testing Security Policies: Going Beyond Function Testing. Proc. ISSRE, pp. 93-102, 2007
8. Li K., Mounier L., Groz R.: Test Generation from Security Policies in OrBAC. Proc. COMPSAC(2), pp. 255-260, 2007
9. Mallouli W., Orset J.-M., Cavalli A., Cuppens N., Cuppens F.: A Formal Approach for Testing Security Rules. Proc. SACMAT, pp. 127-132, 2007
10. Martin E., Xie T.: A Fault Model and Mutation Testing of Access Control Policies. Proc. WWW, pp. 667-676, 2007
11. Martin E., Xie T.: Automated test generation for access control policies via change-impact analysis. Proc. 3rd Intl. workshop on software engineering for secure systems, pp. 5-11, 2007
12. Mouelhi T., Le Traon Y., Baudry B.: Mutation analysis for security test qualification. Proc. Testing: Academic and Industrial Conference, pp. 233-242, 2007
13. Pretschner A., Philipps J.: Methodological Issues in Model-Based Testing. In Broy, M. et al. (eds.): Model-Based Testing of Reactive Systems, pp. 281-291, 2005
14. Sandhu R., Coyne E., Feinstein H., Youman C.: Role-based access control models. IEEE Computer 29(2):38-47, 1996
15. Sandhu R., Samarati P.: Access Control: Principles and Practice. IEEE Communications Magazine 32(9):40-48, 1994
16. Williams A., Probert R.: A Practical Strategy for Testing Pair-wise Coverage of Network Interfaces. Proc. ISSRE, pp. 246-254, 1996
17. Zhu H., Hall P., May J.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys 29(4):366-427, 1997