Access Control Enforcement Testing

Donia El Kateb[1,2], Yehia ElRakaiby[1], Tejeddine Mouelhi[1], Yves Le Traon[1,2]
[1]Security, Reliability and Trust
Interdisciplinary Research Center, SnT
[2]Laboratory of Advanced Software SYstems (LASSY)
University of Luxembourg
Luxembourg
{donia.elkateb, yehia.elrakaiby, tejeddine.mouelhi, yves.letraon}@uni.lu

***Abstract*—A policy-based access control architecture comprises Policy Enforcement Points (PEPs), which are modules that intercept subjects access requests and enforce the access decision reached by a Policy Decision Point (PDP), the module implementing the access decision logic. In applications, PEPs are generally implemented manually, which can introduce errors in policy enforcement and lead to security vulnerabilities. In this paper, we propose an approach to systematically test and validate the correct enforcement of access control policies in a given target application. More specifically, we rely on a two folded approach where a static analysis of the target application is first made to identify the sensitive accesses that could be regulated by the policy. The dynamic analysis of the application is then conducted using mutation to verify for every sensitive access whether the policy is correctly enforced. The dynamic analysis of the application also gives the exact location of the PEP to enable fixing enforcement errors detected by the analysis. The approach has been validated using a case study implementing an access control policy.**

***Index Terms*—Access Control Policies, PEP, PDP, Security Test Cases.**

I. Introduction

In policy-based software systems, access to services is regulated by an access policy specifying controls over subjects' access to services. Several access control models such as RBAC, MAC, DAC and OrBAC [11], [4], [7], [8] may be used to specify access control policies. To make access controls reconfigurable, the recommended standard architecture is based on the separation of the policy decision point (PDP), the security component where access decisions are taken, and the policy enforcement points (PEPs), the security mechanisms inside the business logic where access is controlled. More precisely, after an access request, a PEP queries the PDP. The PDP decides whether this access should be allowed or not. The PEP then enforces the decision taken by the PDP. This separation enables update and evolution of the policy since changes made to the policy in the PDP are directly reflected in policy enforcement at the PEP level. In practice, the PEPs are scattered in several places inside the code, the modules and the components of the system.

In such systems, debugging policy enforcement errors becomes a tedious task since PEPs are often manually implemented, specially in legacy systems. In this context, Policy enforcement errors may consist in a non-alignment between what has been specified in the policy and the security mechanisms enforcing the policy rules at the application level. For instance, a mis-configured PEP may lead to granting access while it should be denied according to the policy. Such unauthorized accesses resulting from incorrect policy enforcement might produce a heavy impact on an organization security. Thus having mechanisms that enable the detection and the correction of policy enforcement errors, is a crucial issue for the secure design of software systems.

In this paper, we propose an approach that automates testing and validation of policy enforcement in policy-based software systems. Our approach includes two steps. First, we use static code analysis in order to identify accesses which can be made from within the application. In particular, we consider the analysis of the application class diagram to determine the different accesses which may need to be regulated by the access control policy rules. This analysis also allows us to detect and remove non relevant rules at the policy level. In this analysis, we have proposed transformation rules that derive, from UML class diagram, the accesses that are relevant to the policy in an application.

The second step aims at helping fixing policy enforcement errors and at establishing a mapping between PEPs in the application and the rules at the policy level. For this purpose, we consider the use of dynamic analysis. In particular, we combine the execution of test cases exercising each selected access control rule with mutation of policy rules. This allows the detection of several policy enforcement problems. For example, it may reveal the absence of access control mechanisms or the incorrect enforcement of a policy rule. Our dynamic analysis also enables the localization of the PEP enforcing a policy rule inside of the application, simplifying the correction of policy enforcement when problems are detected.

The remainder of this paper is organized as follows: Section II introduces the context of this work. Section III presents the overall approach. Section IV presents the main results. Section V discusses related work and Section VI concludes the paper.

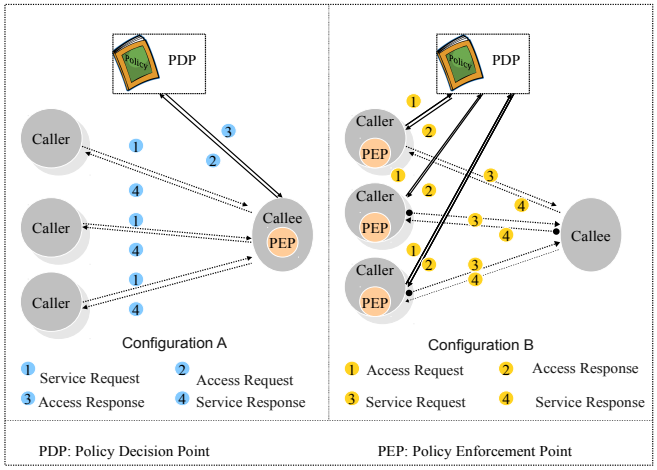

Fig. 1. Possible PEPs Configurations

II. Context

In this work, we assume that the evaluation of access control follows the standard access control PEP-PDP architecture. In this architecture, the PDP evaluates accesses and the PEPs enforce decisions taken by the PDP. A PEP-PDP architecture centralizes the location where the policy is evaluated. Therefore, it simplifies policy update and management. In the context of a Java application, a PEP often corresponds to a method M_p that is encapsulated in the business logic. Every M_p is generally associated with a set of services. Once a given service is requested, M_p is first triggered and the PDP is called. The PDP evaluates the current access control policy and allows or denies access accordingly. The decision taken by the PDP is enforced by the PEP. PEPs are typically organized according to two typical configurations. These configurations are shown in Figure 1 and may be described as follows.

- Access is enforced by a PEP on the callee's (the service) side: In this case, the PEP is encapsulated in the service implementation. Access control is more centralized since enforcement mechanisms reside inside the implementation code of the service itself. Configuration A in Figure 1 illustrates this scenario.
- Access is enforced by a PEP on the caller's (the client) side like shown in Configuration B of Figure 1: In such configuration, controls are more decentralized since the PEP is located at the application level of the client side.

Within an application, PEPs are typically organized according to one of the configurations above or using a mix of the two configurations depending on the security requirements. Testing access control correct enforcement can therefore be problematic since some PEPs are mis-implemented or missing.

III. Approach Description

The overview of our approach is described in Figure 2. The approach is two-fold: First, a static analysis of the application code is made in order to examine relationships between the application classes that are policy-relevant and to determine the set of sensitive method calls that can be made from within the target application. This set corresponds to accesses that might be regulated by the access control policy. The identification of this set therefore enables to detect misspecified security rules or errors in the implementation of the application security mechanisms. After the removal of the non relevant rules, access control tests are generated.

In the second step, a dynamic analysis of the target application is preformed using the tests produced from the first step. In particular, for every sensitive access, a permission is specified in the policy. The security policy mutation tool (MutaX) [2] is then used to generate a mutant of this security rule such that the access is no longer authorized. In our experiments, a simple mutation operator was considered where the rule type is changed, namely a permission is changed into a permission and vice versa. The test corresponding to this particular access is then run twice, one time when the original rule is included in the policy and another time when the original rule is replaced by its mutant.

When the policy is correctly enforced, i.e. the security mechanisms are correctly implemented, the traces generated from the runs of the tests differ at exactly the point where the PEP is located and the trace where access is denied shows the procedure executed after access denial, e.g. a security exception is thrown. Every other case reveals a problem in the enforcement of the policy. For example, two identical traces may mean that there is no policy enforcement mechanism or that the PDP is not queried correctly by the PEP. A trace where access denial procedure is executed when access is authorized or the non-execution of the denial procedure when access should be denied indicates the incorrect implementation of the PEP. To locate the implemented security mechanisms in these two latter cases, the execution trace is captured in the form of a structured tree of method calls and a trace comparison is made to locate the PEP in the target application. Consequently, the correction of the security mechanisms is simplified.

A. Static Analysis: Identification of Possible Accesses

An access control policy regulates accesses made by system users (subjects) to protected resources (targets). Subjects and resources correspond to class instances at the application level and policy roles correspond to classes. For the sake of simplicity, in what follows, we consider instances of a class are assigned a single role. Note that it is straightforward to generalize our approach to the case where instances of a class may be assigned multiple roles. In this context, possible accesses are interactions between classes representing system users and classes representing application services.

To clarify the approach, we consider a library management system comprising users such as professors, secretaries, etc. Protected resources in the system are personal accounts, books, etc. This application's model in the form of a class diagram, has been automatically generated using JaMoPP eclipse plug-in [1] and is shown in Figure 3.

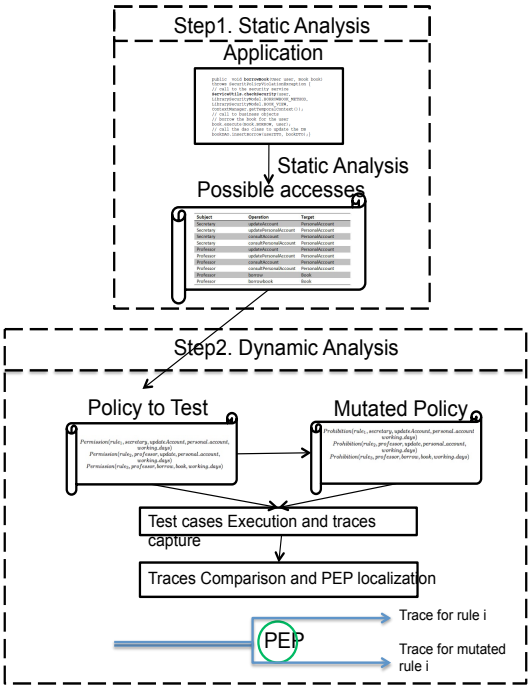

Fig. 2. The Overall Approach

Note that there are *implicit* security rules that are inherent to the implementation of the application. For example, through the UML design, a class *Secretary* cannot directly call *borrowBook* since the program does not have any direct reference from the class *Secretary* to *Books*. In other words, a call of *borrowBook* by an instance of the class *Secretary* will inevitably fail regardless of whether the specified access policy stipulates that this call is allowed or not.

This partially motivates the static analysis of the target application since it allows identification of method calls (accesses) which are executable given the application's model. The motivation behind using static analysis at the class diagram level is to allow generic verifications and analysis of the security constraints at the model level. This analysis can be easily be mapped to the implementation level. This is further discussed in Section III-B.

To identify accesses which can be made by users (subjects) to protected resources (targets), we analyze the application's class diagram. This class diagram is inferred from the application code and, thus, it shows the different associations and operations of the different application's classes. To identify possible accesses, interactions between the classes which represent subjects (Cl_S) and those which represent targets or resources (Cl_R) are analyzed at the application level. Particularly, we consider two types of relationships between subject and target classes, namely *association* and *dependency* relationships.

An *association* describes a discrete connection among classes objects or instances. A *dependency* is a weaker form of relationship indicating that one class depends on another because one of its method requires the use of the other class. More concretely, when a class R_1 is associated with a class T_1, R_1 would have an attribute which is an instance of T_1. On the other hand, when a class R_1 depends on T_1, then T_1 appears as parameter variable or a local variable in one of the methods of R_1.

We formalize our analysis of class diagrams using First-Order Logic as a representation language as follows. To represent the different elements of a UML class diagram, we consider the use of predicates *class(ClassName)* to represent classes of the application and *sclass(ClassName)* and *rclass(ClassName)* to denote classes which represent system subjects and resources respectively. For example, the classes *Professor* and *Book* are represented as follows.

$$class(professor) \qquad class(book)$$
$$sclass(professor) \qquad rclass(book)$$

We represent the *public* operations supported by classes and their parameters using respectively *class_operation* and *operation_parameter*. For instance, the following facts specify that the class *Book* supports the public operation *borrow* which has a parameter of type *Professor*.

$$class\_operation(book, borrow)$$
$$operation\_parameter(book, borrow, profesor)$$

The associations between classes[1] are represented using *association_end(Classe1, Association_Label, Classe2)*. For instance, we represent the association between *Personnel* and *PersonalAccount* as follows.

$$association\_end(personnel, account, personal\_account)$$
$$association\_end(personal\_account, owner, personnel)$$

To take into account dependency relationships between classes, we consider that a class S depends on another class R through an operation Op if one of the operations of S has a parameter of type R. We specify this as follows[2].

$$dependency(S, Op, R) \leftarrow$$
$$class(S), class(R), class\_operation(S, Op),$$
$$operation\_parameter(S, Op, R)$$

We consider that instances of a subject class S can directly execute public operations of a resource class R if an association exists between the subject and resource classes.

$$can\_execute(S, Op, R) \leftarrow sclass(S), rclass(R),$$
$$association\_end(S, \_, R), class\_operation(R, Op)$$

The instances of S can also indirectly execute public operations of R if there exists an association *path* from the S to R. To identify this relation type, we specify the following rules which give the transitive closure of the relation *association_end*.

$$ind\_association(S, R) \leftarrow association\_end(S, \_, I)$$
$$ind\_association(S, R) \leftarrow$$
$$association\_end(S, \_, I), ind\_association(I, \_, R)$$

[1]Note that aggregations and compositions are particular types of associations.

[2]In the following, we use a notation similar to that of Prolog for the specification of derivation rules.

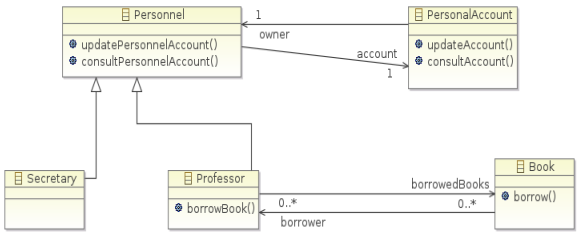

Fig. 3. Application Class Diagram

Thus the $can\_execute$ operation can be specified as follows.

$$can\_execute(S, Op, R) \leftarrow sclass(S), rclass(R), ind\_association(S, R), class\_operation(R, Op)$$

We also consider that S can execute the operation Op on R if there exists a dependency between S and R through an operation Op as follows.

$$can\_execute(S, Op, R) \leftarrow dependency(S, Op, R)$$
$$can\_execute(S, Op, R) \leftarrow dependency(R, Op, S)$$

The first rule above considers operations which access resources on the subject's side while the second rule considers operations on the resource's side. We also consider the generalization relation. Therefore, we specify that a class *P* is a generalization of another class *C* using *generalization(P,C)*. A class can execute operations of its general class. This is specified as follows.

$$can\_execute(C, Op, R) \leftarrow generalization(P, C), can\_execute(P, Op, R)$$

There are other forms of dependency such as instantiation and creation. The formalization of these forms of dependencies has not been presented in this paper for the sake of space limitation. The support of such dependencies is straightforward since they appear clearly in the abstract syntax tree of the application's model. Using the rules described above, we derive the set of possible access operations shown in Figure III-B below that are derived from the class diagram in Figure 3. Figure III-B shows possible accesses which can be made from within the application-code when the subject classes are *Secretary* and *Professor* and the target classes are *PersonalAccount* and *Book*. Note that we consider operations on both the client side and the resource side. For instance, the operations *borrowBook* of *Professor* and *borrow* of *Book* correspond to the same operation performed on both: the client side and on the resource side respectively. We consider the identification of both operations since access controls may be enforced either on the client side or on the resource side or on both.

B. Policy Specification Constraints

The identification of the set of relevant accesses at the level of the application code enables us to identify policy specification constraints. For example, it may allow to filter some rules from the security policy specified by security officers or/and detect some ill-specified rules. To provide an illustrative example, we assume that users accesses are regulated, for instance, by an *OrBAC* access control policy [8]. A permission rule in *OrBAC* is specified using a predicate *Perm(Name,Role,Activity,View, Context)*. This predicate stipulates that a permission whose identifier is *Name* authorizes *Role* (denotes a user in the application) to perform *Activity* (correspond to a method execution) on *View* (represent a protected resource) when the context is *Context*. Prohibitions are similarly defined. A security policy is a set of permission and prohibition rules. For instance, consider the following security policy.

$$Perm(rule_1, secretary, updateAccount, personal\_account, working\_days)$$
$$Perm(rule_2, professor, updateAccount, personal\_account, working\_days)$$
$$Perm(rule_3, professor, borrow, book, working\_days)$$
$$Proh(rule_4, secretary, borrow, book, working\_days)$$
$$Perm(rule_5, secretary, read, book, working\_days)$$

The policy above is composed of five access control rules. These rules are identified by the rule names *rule_1,rule_2,rule_3,rule_4,rule_5*. The policy specifies that both *secretary* and *professor* are allowed to *update account* when the context *working_days* is true. A *professor* is allowed to *borrow book* when the context *working_days* is true but a *secretary* can not. A secretary may *delete* a *book* if it is *working_days*. Using the possible accesses derived from the application's model, we are thus able to analyze the policy. For instance, we can detect that *rule_4* is redundant since the operation forbidden by this rule can not be executed. This condition can be formalized as follows.

$$redundant\_rule(Rule\_ID) \leftarrow Proh(Rule\_ID, S, Op, R, Ctx), \neg can\_execute(S, Op, R)$$

We may also detect several misspecified policy rules. For instance, the permission *rule_5* grants an access which is not relevant with regard to the application design. This is specified as follows.

$$impossible\_rule(Rule\_ID) \leftarrow Perm(Rule\_ID, S, Op, R, Ctx), \neg can\_execute(S, Op, R)$$

The detection of this latter category of rules is particularly important since it reveals a mismatch between the application model and the application structure expected by security officers. In this case, the identification of such problems and their resolution is necessary to ensure the proper functioning of the system. These problems may be resolved either by removing the security rule from the policy or by modifying the application to, for example, enable a *secretary* to read books.

C. Dynamic Analysis

The static part of our approach enables us to identify accesses that are not handled by the security policy and to remove the rules in the policies that are not taken into consideration at the application level. This step allows to narrow the scope of the enforcement mechanisms that have to be tested at the application level and also permits to generate

Subject	Operation	Target
Secretary	updateAccount	PersonalAccount
Secretary	updatePersonalAccount	PersonalAccount
Secretary	consultAccount	PersonalAccount
Secretary	consultPersonalAccount	PersonalAccount
Professor	updateAccount	PersonalAccount
Professor	updatePersonalAccount	PersonalAccount
Professor	consultAccount	PersonalAccount
Professor	consultPersonalAccount	PersonalAccount
Professor	borrow	Book
Professor	borrowbook	Book

Access on the resource side | Access on the client side

Fig. 4. Possible Accesses

security tests. In this section, we detail our dynamic approach and explain how we use mutation applied to access control policies [13][15] to facilitate detecting policy enforcement errors.

We establish a traceability link between access control rules in the policy and the PEPs in the application code through a mapping between every access rule and the PEP that evaluates it. This mapping is quite important for the verification of access control implementation when the policy evolves. In [15], we have presented different mutation operators that have been applied to access control policies. In this paper, we only use two mutation operators, namely PRP (Prohibition to Permission) and PPR (Permission to Prohibition). A concrete example of a mutation applied to an access control rule is shown below:

$$Perm(R_1, administrator, manageAccess, personnelAccount, default)$$
$$Proh(mutated\_R_1, administrator, manageAccess, personnelAccount, default)$$

This example presents the mutation of the rule R_1 into the rule $mutated\_R_1$ by applying the operator Permission to Prohibition. The context *default* represents a context that is always true, i.e. the permission and prohibition are unconditional. Note that our mutation operators do not generate equivalent mutants since injected changes produce a mutant policy that is always different from the initial one. Figure 5 presents the generated access control test case for the rule *Proh(rule, borrower, return_book, book, maintenanceDay)*. This security test case verifies the correct enforcement of the policy.

D. PEPs Localization for Access Control Enforcement Testing

The relevance of the rules in the policy and the access control mechanisms is identified using the techniques presented in section III-A. For instance, if we consider the application in Figure 3, the policy will include the unconditional permission *Perm(professor:borrow:book,professor, borrow,book,default)* since this permission corresponds to one of the application's possible accesses specified as follows:

$$Perm(S{:}Op{:}R, S, Op, R, default) \leftarrow can\_execute(S, Op, R)$$

```
// test data initialization
// log in a student
std1 = userService.logUser("login1", "pwd1");
// create a book
book1 = new Book("book title");
// book needs to be borrowed before returned
borrowBookForStudent(std1,book1);
// context
contextManager.setTemporalContext(maintenanceDay);
// run test
try { returnBookForStudent(std1,book1);
// security oracle
// SecurityPolicyViolationException is expected because an
access control rule was not applied - test failure
 fail(" SecurityPolicyViolationException expected,
returnBookForStudent with student = " + std1 + " and book = " +
book1); }
catch( SecurityPolicyViolationException e) {
 // ok security test succeeded log info
 log.info("test success for rule :
prohibition(borrower,return_book,maintenanceDay)");
```

Fig. 5. Security Test Case Example

This last rule specifies that a permission with identifier $S : Op : R$ allowing S to execute the operation Op on R should be derived if S can execute Op on R. In the dynamic step, the application is tested by a test suite that tests the permissions and the prohibitions. These tests are executed and execution traces are generated for an application that interacts, respectively with the original policy and with the mutated one. For the mutated policy, there is a single rule that is inverted from permission to prohibition. The execution traces produced are then compared: when the PEP correctly enforces the policy, the execution trace of the original policy and the mutated one differ at exactly the location of the PEP in the code. One of the execution traces also shows the procedure executed after access denial. On the other hand, when the two traces are identical, this may mean either that there is no implemented policy enforcement mechanism or that security mechanisms fail to correctly query the PDP. Also, a trace where access denial procedure is executed when access is authorized or the non-execution of the denial procedure when access should be denied indicates an error in the implementation of the enforcement mechanisms. To simplify the correction of the security mechanism implementation in these two latter cases, the execution trace is captured in the form of a structured tree of method calls and a trace comparison is made to locate the PEP in the target application. Thus, the application debugger can find the location in the application where the security mechanisms are implemented.

We describe our approach more formally as follows: we consider that a Java application that is regulated by an access control policy is a system S. This system consists of a set of methods M and is governed by a policy $P = \{R_i\}_{i=1..n}$ where every R_i is a security rule which corresponds to a possible access and *n* is a natural number. A mutated policy rule R_i^m is a non-equivalent mutated version of the access

control rule R_i in P after the application of a mutation operator on R_i. A test suite is composed of a set of test cases which cover the original rules R_i in the policy P and their mutated version R_i^m. For every test case, we generate an execution trace. An execution trace is a sequence of method calls $< m_1, ..., m_l >$ where every m_i is a method of M. We denote by $Tr(R)$ the execution trace of the test case of the rule R. To document the PEP, we consider $Map \subseteq Cl_S \times M \times Cl_R \times M$. A $map(sclass, op, rclass, pep)$ means that the PEP which enforces the execution of the operation op by instances of the class $sclass$ on instances of the class $rclass$ is the method pep. Our map is initially empty at the beginning of the dynamic analysis, i.e. $Map = \emptyset$. The execution of a test case typically produces a trace which takes one of the following two forms.

- When R is a policy rule R_i, we get an execution trace of the form $< a_1, ..., a_m, M_p, b_1, ...b_n >$ where M_p is the PEP enforcing the policy. In this trace, access is allowed and no security exception is triggered since every R in P is an unconditional permission.
- When R is a mutated rule R_i^m, we get an execution trace $< a_1, ..., a_m, M_p, M_{ex}, c_1, ...c_j >$ where M_{ex} is the method denoting the triggering of a security exception. Note that the traces of a policy rule $Tr(R_i)$ and of its mutated version $Tr(R_i^m)$ generally differ at the exact location of the PEP. Therefore, we document the PEP by adding the access $(sclass, op, rclass)$ which corresponds to the rule R to our PEP-access map, i.e. $MAP = MAP \cup (sclass, op, rclass, M_p)$.

It may occur that the two traces produced by the test cases of a policy rule R_i and its mutated version R_i^m are identical. This generally means that either the PEP is wrongly implemented or that there is no PEP to control this access. In this case, we add the entry $(sclass, op, rclass, \_)$ to the map to indicate that there is no PEP associated with the access $(sclass, op, rclass)$. Note that these accesses can not be regulated by the access control policy. It may also happen that the trace $Tr(R_i)$ of the original policy shows the access denial procedure or that $Tr(R_i^m)$ does not show the denial procedure, in this case the entry $(sclass, op, rclass, M_p)$ shows the location M_p where the security mechanisms querying the PDP are implemented.

E. PEPs Localisation: Trace Analysis

We implement our dynamic approach through the tracking of the execution of test cases using a technique based on Aspect-Oriented Programming (AOP) [3]. AOP is a programming paradigm that aims at separating cross-cutting concerns to improve modularity. We use AspectJ, a widely used defacto standard for AOP [9], to implement a logging concern to track and log the execution of test cases. We define a logging advice that is run when the program executes the test cases. This advice logs all successive routine calls once a given trace is executed. The code in Figure 6 shows our TracingTests aspect. The TraceTestsMethods() pointcut logs the execution of all Test Cases using the regular expressions *Test. After the execution of each Test Case method, the code in the after advice is executed and logs all method calls after the execution of Test Cases.

```
public aspect TracingTests {
TracingTests(){}
Logger logger1=Logger.getLogger("trace");

pointcut TraceTestsMethods(): cflow (execution(* test*(..))) & within(test.security.ChatServiceSecTest)
    || within(test.security.MeetingServiceSecTest)
    || within(test.security.PersonnelAccountServiceSecTest)
    || within(test.security.UserAccountServiceSecTest);
.....
after(): TraceTestsMethods()     {
    System.out.println("----------" + " Mutant is number: " +
TestOracle.mutantNumber);
    Signature sig = thisJoinPoint.getSignature();
    String infoformat="entering [" + sig.getDeclaringType().getName()+ "." +sig.getName()+"]";
    logger1.info(infoformat);
    printtofile(infoformat+"\n");
```

Fig. 6. Tracing Test Execution Aspect

IV. Evaluation Results

We applied the dynamic part of our approach to Virtual Meeting Management System (VMMS, a Java system that interacts with an access control policy [16], [10]. It contains 6077 lines of code, 134 classes and 581 methods. VMMS offers web conference services. It allows the organization of work meetings on a distributed platform. Once connected to the virtual meeting service, the user can join a meeting, intervene in a speech, or plan new meetings. Every meeting has a manager who is responsible for planning the meetings and setting the meeting parameters (name, agenda, duration,...). Every meeting may also have a moderator, designated by the meeting manager. The moderator gives the floor to participants wishing to participate in the meeting. The security policy P inferred from the application code specifies 87 permission rules. These rules correspond to possible accesses in the application. We execute test cases of every rule in the policy as well as its mutated version. Using the aspect presented in the previous section, we collect the resulting execution traces in output files.

To locate the PEPs based on execution traces, we compare the execution traces of the original rule and of its mutated version as previously explained in Section III. Figure 7 shows two execution traces extracted from 2 trace output files. The first trace represents a trace execution corresponding to a rule policy R_1*: Permission(rule, Personnel, SetMeetingAgenda, Meeting, Default)* and the second denotes a trace execution corresponding to the mutated version of the rule R_2*: Prohibition(mutated_rule, Personnel, SetMeetingAgenda, Meeting, Default)*.

An analysis of the two traces shows that once the policy is mutated, a call to the function service.UserService.disconnectUserFromMeeting produces a security exception. By comparing the two traces, we detect access control enforcement errors and we automate the localization and the mapping of all the PEPs which enforce a possible access in the system. The table 1 shows some of the PEPs distribution that we have identified per each class and the number of rules that are relevant for those PEPs.

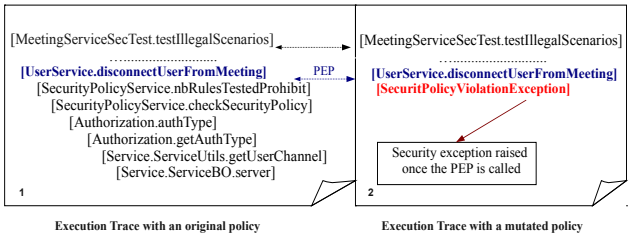

Fig. 7. PEPs localization through Trace analysis

TABLE I
PEPs DISTRIBUTION

PEPs by Class	Rules Number
ChatService	24
speakInMeeting	10
handOver	2
askTo Speak	10
PersonnelAccountService	7
deletePersonnelAccount	2
updatePersonnelAccount	1
MeetingService	32
putMeetingTitle	8
closeMeeting	4
putMeetingModerator	8

V. RELATED WORK

Some research contributions have tackled the research direction of security policy testing in the last few decades. In [5], Devanbu et al. have highlighted that the update of a system policy often raises the question of whether the security mechanisms are able to correctly enforce an evolving security policy. In [17], Ray et al. have proposed an approach to locate access control concerns as separate aspects that can be reusable as patterns. The separation of access control concerns at the system design level enables to ease the process of testing access control mechanisms. The difference between their work and our work is that, in our work, we focus on applications that do not take the problem of cross cutting concerns into consideration since the access control mechanisms that we consider are scattered across our policy-based application. In [12], the authors have used Nomad language to formalize the policy independently of the underlying application and then they have analyzed the application execution traces to analyze the conformance of the system to its security policy.

The work that we propose in this paper extends our previous work on access control testing. In [14], PEPs are implemented using AOP and security tests are assessed by mutation analysis applied to access control policies. In [10], we have proposed a test-based approach to detect hidden access control mechanisms that consists in analyzing system response to incoming access control requests. In [6], we have defined mutation operators to test the enforcement of obligation policies. In comparison with our previous work, the current contribution does not make any assumptions regarding the visibility of PEPs. Moreover, in the current approach, we propose techniques that allow to map every possible access in the application to the PEP in the application code which controls it (if this PEP exists). Our technique also provides valuable information for the analysis of the security policy specified by the security officers since it permits to detect inconsistencies between the policy explicitly specified by security officers and the application code.

VI. CONCLUSION

In this work, we have proposed an approach to help fixing policy enforcement errors. The approach relies on two steps: The static step generates the relevant accesses for an application through its class diagram using inference rules. It also removes non relevant rules in the policy and detects inconsistencies between what is specified at the policy level and the enforcement mechanisms at the application level. The dynamic part of our approach checks the errors of policy enforcement through a trace analysis applied to an original policy and a mutated one. As future work, we plan to consider other possible dependency relationships between classes and to study their impact on our approach.

REFERENCES

[1] http://www.jamopp.org/index.php/jamopp.
[2] Mutax: https://sites.google.com/site/servalteam/tools/mutax.
[3] M. Aksit. Principles of aspect-oriented programming languages, design dimensions and the composition filters approach. page 15, 2004.
[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical report, 1975.
[5] P. T. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 227–239, 2000.
[6] Y. Elrakaiby, T. Mouelhi, and Y. Le Traon. Testing obligation policy enforcement using mutation analysis. In *ICST*, pages 673–680, 2012.
[7] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. 2001.
[8] A. A. E. Kalam, S. Benferhat, A. Miège, R. E. Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin. Organization based access control. In *POLICY*, 2003.
[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pages 327–353, 2001.
[10] Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry. Test-driven assessment of access control in legacy applications. In *Proc. the 2008 International Conference on Software Testing, Verification, and Validation*, pages 238–247, 2008.
[11] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *In Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.
[12] W. Mallouli, F. Bessayah, A. R. Cavalli, and A. Benameur. Security rules specification and analysis based on passive testing. In *GLOBECOM*, pages 2078–2083, 2008.
[13] T. Mouelhi, F. Fleurey, and B. Baudry. A generic metamodel for security policies mutation. In *ICST Workshops*, pages 278–286, 2008.
[14] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon. A model-based framework for security policy specification, deployment and testing. In *MoDELS*, pages 537–552, 2008.
[15] T. Mouelhi, Y. Le Traon, and B. Baudry. Mutation analysis for security tests qualification. In *Mutation'07 : third workshop on mutation analysis in conjuction with TAIC-Part, September 10-11*, pages 171–180, 2007.
[16] T. Mouelhi, Y. Le Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. In *ICST*, pages 171–180, 2009.
[17] I. Ray, R. B. France, N. Li, and G. Georg. An aspect-based approach to modeling access control concerns. *Information & Software Technology*, pages 575–587, 2004.