A PEP-PDP Architecture to Monitor and Enforce Security Policies in Java Applications

Yehia Elrakaiby, Yves Le Traon
Security, Reliability and Trust Interdisciplinary Research Center, SnT, University of Luxembourg, Luxembourg
Email: {yehia.elrakaiby,yves.letraon}@gmail.com

***Abstract*—Security of Java-based applications is crucial to many businesses today. In this paper, we propose an approach to completely automate the generation of a security architecture inside of a target Java application where advanced security policies can be enforced. Our approach combines the use of Aspect-Oriented Programming with the Policy Enforcement Point (PEP) - Policy Decision Point (PDP) paradigm and allows the runtime update of policies.**

***Keywords*—Java Security, Security Policies, Obligations, Access Control, Usage Control.**

I. Introduction

Java security has attracted the attention of many researchers due to the widespread use of Java to develop applications and businesses today.

Several techniques have been considered to provide security for Java applications. Many researchers [1]–[8] studied the use of In-lined Reference Monitors (IRM) [9] and Aspect Oriented Programming (AOP) [10] to weave mechanisms needed to enforce security requirements into Java applications. The main limitation of this approach is that policies may not be updated afterwards. For example, a policy stating that "a user can post comments on the website if he or she is a subscribed member" can not be changed into a requirement that "any registered user can post comments on the website". Other approaches such as XACML [11] propose to separate policy enforcement from policy evaluation using a Policy Enforcement Point (PEP) / Policy Decision Point (PDP) architecture, enabling policy update at runtime. In this paradigm, a user access request is intercepted by a PEP which forwards the request to the PDP for decision-making. The decision taken by the PDP is returned to the PEP which enforces it. The difficulty in using this latter approach to secure Java applications consists in the need to develop several application-specific modules to intercept sensitive method calls in the application and manage the communication between the PEPs and the PDP.

In this paper, we propose a generic approach that combines the advantages of the two aforementioned approaches: in comparison with IRM-based approaches, the policy can be updated at runtime; with respect to other approaches using a PEP-PDP architecture, there is no need to develop application specific modules. To overcome this latter issue, we have included in the policy language the constructs necessary to define a declarative mapping between the Java entities of the target application (e.g. methods) to their corresponding policy entities (e.g. actions), thereby clarifying policy semantics and avoiding the need to implement application-specific modules. From a conceptual point of view, our approach consists in making a complete separation between the policy and application entities and providing means necessary to map the application and policy entities together. Another main advantage of this separation between the policy and application realms is that it simplifies the extension of the policy language without affecting the application. This is demonstrated by the support the specification of sophisticated security requirements such as obligations, history-based and reaction policies in our approach.

The remainder of this paper is organized as follows. Section II presents a motivating example. Section III gives an overview of our approach and introduces the MJP language. Section IV details the PEP-PDP architecture automatically produced given an MJP policy specification. Section V makes an empirical assessment of our approach. Finally, Section VI discusses related work and Section VII concludes the paper.

II. Motivating Example

Consider an Auction Sale Management System (ASMS) [12]. In ASMS, users have accounts and are allowed to buy and sell products online. Users can join auction sessions, place bids and post or read messages from the session's bulletin board. Each session has a designated moderator. Along with this functional description of the system, the following requirements are needed to comply either with regulations such as privacy laws [13], [14] or with the service's internal policy.

- R_1: Only users can update the account information,
- R_2: To ensure service availability, each user may not post more than 50 comments during a sale session,
- R_3: When a user posts a comment including a blacklisted word, then the moderator has to update the post such that the word no longer appears in it, i.e. the moderator may either completely delete the post or modify the post by removing the blacklisted word.
- R_4: Users should not have a negative account balance for more than one week. If within one week, the user's account is still negative, then the user's access is suspended until his balance is positive again.

The examples above show different requirements: R_1, R_2 are contextual access control rules. R_2 further specifies conditions on the user's past actions. R_3 and R_4 specify obligation requirements. R_4 further specifies that a user should be denied to access the service if the user fails to comply with the obligation to not have a negative balance for more than one week. We consider that R_4 represents a policy requirement (and not a functional requirement) since it may change later if there is a change in the service's policy.

TABLE I: Rule Predicates

Predicate	Informal Meaning
$perm(i, r, a, r_t, c_a)$	Users assigned the role of r are allowed to perform the action a on targets in the role of r_t if the context c_a holds.
$proh(i, r, a, r_t, c_a)$	Users assigned the role of r are prohibited to perform the action a on targets assigned the role of r_t if the context c_a holds.
$oblig_a(i, r, a, r_t, c_a, c_v)$	Users assigned the role of r are required to perform the action a on targets in the role of r_t when the context c_a holds before the context c_v is true.
$oblig_s(i, r, c_f, c_a, c_v)$	Users assigned the role of R are obliged to make the context c_f true when the context c_a holds before the context c_v is true.
$i \in \mathcal{I}, \quad r, r_t \in \mathcal{R}, \quad a \in \mathcal{A}, \quad c_a, c_v, c_f \in \mathcal{C}$	

To correctly specify and enforce the requirements above in a Java application, we identify the following challenges: (1) specification of requirements using a high-level language to simply their expression and interpretation, (2) support of fine-grained security rules and the verification of contextual conditions such as, for example, that the user is requesting to update his *own* account, (3) provision of means to map the specified policy to the Java application entities, (4) monitoring of compliance with requirements and activation of counter-measures in case of non-compliance, e.g. R_4, (5) enabling the update of requirements to enable enforcement of new internal or regulatory mandates, e.g. that also moderators may update user accounts (R_1) or that users can have a negative for a maximum of two days instead of one week (R_4).

III. Securing Java Applications using Policies

To address the challenges just identified, we introduce a methodological approach to enforce high-level requirements into target Java applications. In the following, we first introduce our policy language enabling an application-independent specification of high-level requirements, then we present how policy entities can be mapped onto the target application. Finally, we will present a security architecture that is able to enforce and monitor compliance with these requirements inside of the target Java application.

A. Policy Specification

Our policy language is a typed first-order language that includes the entities shown in Table I: A *role* [15], an *action* and a *context* [16] represent an abstraction of a set of system users, a user action and a set of rule conditions respectively. Each entity is represented using a different sort: *roles* ($\mathcal{R}$), *actions* ($\mathcal{A}$), *context expressions* ($\mathcal{C}$) and *rule identifiers* ($\mathcal{I}$). Context expressions are built from basic contexts, a basic context is an identifier of a set of state conditions. For example, a basic context *"emergency"* could denote an emergency situation. Context expressions are defined as follows:

$$exp ::= c \mid \top \mid \bot \mid [exp, exp] \mid delay(nU) \mid exp \,\&\&\, exp.$$

where c is a basic context, n is a natural number and U is from the set {s (seconds), m (minutes), h (hours), d (days)}. The special context $\top$ ($\bot$ respectively) is a context that is always true (false respectively). An interval context $[c, c']$ holds since the context c holds until the context c' holds. The context *delay(nU)* holds after the elapse of a time period of a number *n* of the entity *U* since the activation of the security rule.

For example, the ASMS policy presented in Section II can be specified as follows:

$$\begin{gathered} perm(1, user, update\_account\_info, account, \\ personal\_account), \\ proh(2, user, post\_comment, sale, \\ exceeded\_50\_comments), \\ oblig_a(3, moderator, update\_comment, comment, \\ comment\_contains\_blacklisted\_word, delay(1h)), \\ oblig_s(4, user, \neg has\_negative\_balance, \\ has\_negative\_balance, delay(7d)). \end{gathered}$$

Note that requirements are specified using a high level application-independent notation simplifying their specification and interpretation. This also simplifies policy reuse in different systems.

To complete the specification of requirements in Section II, we specify that a user should be prohibited to access the service if he or she violates the requirement 4 as follows:

$$proh(5, user, log\_in, service, violated\_r_4).$$

B. Mapping Applications to Policies

Java, as every object oriented programming language, is based on objects (alternatively called instances), classes, attributes (data fields) and methods. Policies on the other hand are specified on an abstract level using roles, actions and contexts. To specify the meaning of policy entities in a given environment (a Java application in our case), we consider making a first-order logic representation of the environment and the use of *mapping rules* to map the policy entities onto the entities of the target application, i.e. that for each policy entity, a corresponding entity in the target environment is specified. For a Java application, we consider mapping policy entities as follows: (1) a class is mapped onto a role, (2) an action is mapped onto a (chain of) method call_of(s) and (3) a context specifies a set of conditions of the application state, i.e. the values of object fields and method parameters.

1) Roles and Actions: In our approach, a Java class is denoted by a role in the policy, i.e. every instance of this class is considered a subject of the corresponding role. Similarly, a (chain of) method call_of(s) is mapped onto a policy action. These mappings are defined using the following propositions:

a) Role Declaration:

Role <*role_name*>
Class_Name <*className*>
[**Relevant_Fields** <*attType*> : <*attName*>
{, <attType> : <attName> }]. (1)

where *className* is the fully qualified name of a class and *role_name* is a role name from $\mathcal{R}$. This proposition states that instances of *className* are assigned the role of *role_name*. The *Relevant_Fields* part is an optional list of some of the attribute names of *className* and their types. It is needed only for performance reasons: only change in relevant attributes is monitored, i.e. the other attributes of a class that are not declared relevant are not monitored and their update does not cause any policy update processing.

For example, we can specify a correspondence between the class *Person* and the role *person* as follows:

Role *person*
Class_Name *person.Person*
Relevant_Fields *int : age, String : password.*

Similarly, we specify a role declaration for *Sale* as follows:

Role *sale*
Class_Name *sale.Sale*
Relevant_Fields *boolean : forMajors*, *int : id*,
ArrayList<String> : blackListOfWords,
ArrayList<Comment> : commentsList.

b) Action Declaration:

Action <*action*>
Method_ID <*method_id*> {–> <method_id> } (2)
Method_Sig <*methodSignature*> {–> <methodSignature> } .

where *methodSignature* is a fully qualified method name and *action* is a policy action from $\mathcal{A}$. The *Method_ID* part gives every method in the *Method_Sig* part an identifier to be used to refer to the method in the policy, therefore the two parts should be of the same length.

For example, the following proposition specifies that the chain composed of *CommentService.postComment* followed by *Sale.postComment* corresponds to the policy action *post_comment*:

Action *post_comment*
Method_ID *post_comment_1 –> post_comment_2*
Method_Sig *CommentService.postComment(int,Message) –> Sale.postComment(Message).*

2) Contextual Mappings: So far, we have described the specification of mappings between policy roles and actions and the target application classes and methods. These mappings are sufficient for the specification of coarse-grained security rules, e.g. that some class may execute some method of another class (or subjects of a role R_T). This is however generally insufficient. For example, it does not allow the specification of contextual conditions such that the user should be the *owner* of the account he is requesting to update.

To add support for such contextual policies, we define a fist-order representation of the *application state*. In our approach, the application state is a set of facts representing the instances of relevant classes and their attributes. The application state may also include a representation of the (chain of) method call_of(s) being currently executed.

Instances of relevant classes and their attributes are represented in the application using two predicates, namely *instance_of(s,r)* and *attribute(s,n,v)* shown in Table II. For example, the facts *instance_of(i,person)* and *attribute(i,age,18)* hold in an application state if the application has an instance

TABLE II: Application State Predicates

Predicate	**Informal Meaning**
instance_of(s,r)	s is an instance of the class r (s is assigned the role of r)
attribute(s,n,v)	v is the value of the attribute n of the instance s
attribute(s,n,op,v)	v is related to the value of the attribute n of the instance s according to the operator op
call_of(m,m_id)	m is a call of the method m_id
$s, m \in \mathcal{O}$: instance identifiers, $r \in \mathcal{R}$: roles, $v \in \mathcal{V}$: attribute values	
$n \in N \cup \mathbb{N}$: attribute names and natural numbers, $m\_id \in \mathcal{M}$: method identifiers	
$op \in \begin{cases} \{<=, >=, <, >\}, & \text{if } n \text{ is an int, double, float, etc} \\ \{startsWith, contains\}, & \text{if } n \text{ is a String} \\ \{includes\}, & \text{if } n \text{ is a collection, ArrayList, etc} \end{cases}$	

i of the class *person* and the value of the field *age* of *i* is 18. Note that *instance_of* facts are also used to represent the super classes of an instance. For example, if *moderator* is a subclass of *person*, then an instance *i* of *moderator* would have two *instance_of* facts: *instance_of(i,person)* and *instance_of(i,moderator)*.

Note that the application state is dynamic, i.e. facts are added and removed from it to reflect the current state of the target application. Method calls are represented in the application state using the predicates *call* and *attribute*. A fact *call_of(m,post_comment_1)* means that *m* is a (current) call of the method *post_comment_1*. The parameters of a method are represented using the predicate *attribute*, similarly to class fields. However, the argument *n* corresponds in this case to the position of the method argument as opposed to the field name. For example, a fact *attribute(m,1,y)* means that the first parameter of the call *m* is *y*. Each method call has two special attributes, namely the *this* and *target* attributes. The *this* attribute identifies the calling instance and the *target* denotes the instance on which the method is invoked.

In the following, we explain how security rule contexts, such as *personal account* can be defined. Then, we show how method calls are mapped into a policy subject, action and target to enable the evaluation of access control policies.

a) Hold Rules: Basic contexts are defined using hold rules shown in Table III. They specify conditions on the application state that have to hold for security rules to be applicable. A hold rule is a proposition of the following form:

$$hold(S, A, T, C) \leftarrow F. \tag{3}$$

A hold rule specifies that *C* holds between the security rule subject *S*, target *T* and action *A* if the formula *F* holds in the application state. For example, we can define the context *personal_account* of the rule 1 in Section III-A as follows[1]:

$$hold(S, \_, T, personal\_account) \leftarrow instance\_of(S, person) \wedge \\ instance\_of(T, account) \wedge attribute(S, userAccount, T).$$

The previous rule specifies that if the value of the attribute *userAccount* of *S* is *T*, *S* is a *person* and *T* is an *account*, then *personal_account* holds between *S* and *T*.

To enable the specification of more sophisticated conditions over the application state, we consider a quaternary predicate

[1]The *"any term"* Prolog symbol _ is used when the S, A or T of the context is irrelevant.

TABLE III: Policy Mapping Predicates

Predicate	Informal Meaning
hold(s,a,t,c)	the context c holds between s, a and t
operation(s,a,t)	the instance s is taking the action a on t
$s, t \in \mathcal{O},\quad a \in \mathcal{A},\quad c \in \mathcal{C}$	

$attribute$ where the third argument is an operator. For example, *attribute(I,age,<,19)* holds in an application state if the age of *I* is less than *19*. It can be used in the specification of context rules. For example, we may specify that the context *"major"* holds for a subject if its age attribute is more than or equals 18 as follows:

$$hold(S, \_, \_, major) \leftarrow attribute(S, age, >=, 18).$$

We have defined operators for Java primitive types and collections such as *ArrayList* as shown in Table II. These operators enable the specification of sophisticated rule conditions. For example, consider the context *"comment_contains_blacklisted_word"* of rule 3 in Section II identifying the comment that should be deleted when the comment contains a word that is blacklisted in the sale where the comment is posted. It may be specified as follows:

$$\begin{gathered} hold(\_, \_, Comment, comment\_contains\_blacklisted\_word) \leftarrow \\ instance\_of(Sale, sale)\ \& \\ attribute(Sale, commentsList, includes, Comment)\ \& \\ attribute(Sale, blackListOfWords, includes, Word)\ \& \\ attribute(Comment, content, contains, Word). \end{gathered}$$

This context thus holds if there is a *Sale* that includes a *Comment* (in the list of comments) and this *Comment* includes a *Word* that appears in the sale's *blackListOfWords*.

b) Operation Rules: Security policies define controls over operations, i.e. actions taken by users on target objects. An operation rule defines an operation by specifying conditions on the application state. They are propositions of the following form:

$$operation(S, A, T) \leftarrow F. \qquad (4)$$

where *F* is a formula and *A* is an action.

For example, consider the method *CommentService.updateComment(Person person, Comment comment)* where *person* is the subject updating the *comment*. An operation rule could map a call to this method onto a subject (the *person*), action (*update_comment)* and target (*the comment*) as follows:

$$\begin{gathered} operation(S, update\_comment, T) \leftarrow \\ call\_of(m, updateComment)\ \& \\ attribute(m, 1, S)\ \&\ attribute(m, 2, T). \end{gathered}$$

To show the use of operation rules with a chain of method calls, consider the sequence diagram shown in Figure 1:

- An object *Person* calls the method *SaleService.postMessage(int saleID, Message message)* where *saleID* is the identifier of the sale where the message is to be posted.

TABLE IV: Rule State Predicates

Predicate	Informal Meaning
act_perm(i,s,a,t)	i allows s to take a on t
act_proh(i,s,a,t)	i forbids s to take a on t
act_obl(i,s,a,t)	s is obliged by i to take a on t
viol_obl(i,s,a,t)	s has violated i but i is still required
fulf_obl(i,s,a,t)	s has fulfilled i and i is not required
fviol_obl(i,s,a,t)	s has fulfilled i after i was violated
$i \in \mathcal{I},\quad s, t \in \mathcal{O},\quad a \in \mathcal{A},\quad r \in \mathcal{R}$	

- The object *SaleService* is the server, it uses the *saleID* to retrieve the *Sale* object from a list of sales where the message should be posted and calls the method *Sale.postComment* to post the comment.

This chain of method call may be mapped to a policy operation, i.e. a policy subject taking a policy action on a target, by specifying that *Person* is the subject and *Sale* is the target and that the action is *post_comment* as follows:

$$\begin{gathered} operation(S, post\_comment, T) \leftarrow \\ call\_of(M1, post\_comment\_1)\ \&\ attribute(M1, this, S)\ \& \\ call\_of(M2, post\_comment\_2)\ \&\ attribute(M2, target, T). \end{gathered}$$

The previous rule specifies that calling the method *post_comment_2* after the execution of *post_comment_1* corresponds to the execution of the policy action *post_comment*. The rule also identifies the calling instance of *post_comment_1* as the subject of the operation and the called instance of *post_comment_2* as the operation target. Note that it is possible not to identify a subject or a target for the operation by using the "any term" symbol _. For example, *operation(S,post_comment,_)* is interpreted as denoting the taking of the action *post_comment* by *S* on any target.

c) Compliance Monitoring and Reaction to Violation: In our approach, compliance with obligation requirements is monitored and it is possible to react to a violation of a requirement by activating new security rules. The support of this kind of policies relies on the special set of predicates shown in Table IV. This set represent the state of security rules. For example, we can specify the context *violated_*r_4 of the rule 5 in Section III-A as follows:

$$hold(S, \_, \_, violated\_r_4) \leftarrow viol\_obl(4, S, A, O).$$

The previous context rule specifies that $violated\_r_4$ holds for a subject S if he is in violation of obligation requirement 4.

C. State Variables

Policies often depend on the history of past events (*history-based policies*) [2]. To support this kind of policies, a *policy variables* can be created and updated. There are two types of policy variables: (1) instance variables: extend the instances of relevant classes with additional attributes, and (2) global variables which are attributes of the global policy state.

Instance variables are declared in *role declarations*. For example, we may specify that instances of the relevant class person should be extended with an attribute *nb_of_comments* of type *int* as follows:

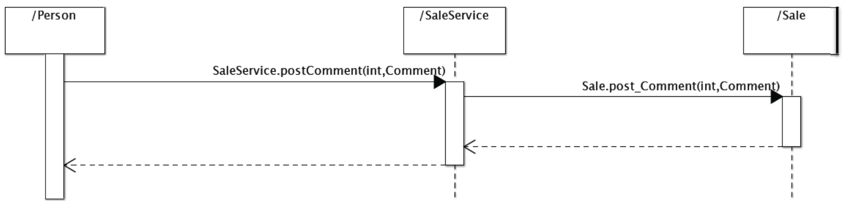
/Person
/SaleService
/Sale
SaleService.postComment(int,Comment)
Sale.post_Comment(int,Comment)

Fig. 1: Posting a Comment Sequence Diagram

Role <*role_name*>
Class_Name <*className*>
[**Relevant_Fields** <*attType*>:<*attName*>
{ <attType>:<attName> }]
[**Additional_Fields** <*attType*>: <*attName*>
{, <attType>: <attName> }].

Role *person*
Class_Name *com.sales.bo.person.Person*
Relevant_Fields *int : age, String : password*
Additional_Fields *int : nb_of_comments*.

Currently, we only support the declaration of instance attributes of type *int* and *String*. The value of an instance attribute is initialized according to the attribute type.

Instance attributes may be updated after action occurrences or when some conditions are true using *state-update rules*. State-update rules are of the following form:

$$F \rightarrow update\_actions. \tag{5}$$

where $update\_actions$ is a set of assertions and retractions. For example, we may specify that the attribute fact *nb_of_-comments* should be updated each time a user posts a comment as follows:

$$operation(S, post\_comment, T) \rightarrow \\ retract(attribute(S, nb\_of\_comments, X)), \\ assert(attribute(S, nb\_of\_comments, X + 1)).$$

D. Policy Evaluation Logic

Due to space limitations, we only briefly present the access control decision logic. Obligation management and their formal operational semantics are presented in [17].

$$allow\_operation(S, A, T) \leftarrow \\ operation(S, A, T), permitted(S, A, T), \neg prohibited(S, A, T), \\ permitted(I, S, A, T) \leftarrow \\ permission(I, Rs, A, Rt, C), instanceof(S, Rs), action(A), \\ instance\_of(T, Rt), hold(S, A, T, C), \\ prohibited(I, S, A, T) \leftarrow \\ prohibition(I, Rs, A, Rt, C), instanceof(S, Rs), action(A), \\ instanceof(T, Rt), hold(S, A, T, C).$$

The rules above specify that an operation is authorized if it is permitted and not denied by the specified policy. An operation is permitted (prohibited) if there is a permission (prohibition) matching the subject, action and target of the permission (prohibition) and the context of this permission (prohibition) holds.

TABLE V: MJP Specification

<MJP Specification>::=	<Role Declarations> <Action Declarations> <Operation Rules> <Hold Rules> <Security Rules> [<State Update Rules>]

IV. Security Architecture to Enforce the Policy

A policy specification includes sets of operation rules, hold rules, security rules, role and action declarations as shown in Table V. A specification may optionally include a set of state update rules for policy state variables. Given a policy specification and a target application, the (logical) architecture in Figure 2 is used to enforce the policy. The architecture comprises the following components: (1) an extended PEP (xPEP) primarily monitors change in the target application and notifies the PDP when a policy-relevant event is detected, e.g. when the attribute of an instance of a relevant class is updated, (2) a stateful PDP updates the policy after reception of notifications from the xPEP, e.g. activates obligations. This section describes these two components and their implementation.

A. Extended Policy Enforcement Point

1) Implementation: The xPEP is implemented using Aspect-Oriented Programming (AOP) [10]. AOP is a programming paradigm providing an efficient way to encapsulate cross-cutting functionalities in one place (called an aspect) instead of having it spread across the target application. An aspect defines one or more *pointcut expressions* and *code advices*. A pointcut expression selects points in the program at which a cross-cutting concern needs to be applied. These points are called *join points* and they correspond to points exposed by the AOP system, e.g. method executionand object creation. The *advice* is the additional code that should be executed when a pointcut expression is matched. It is possible to *run* an advice before, after, or around a join point.

In our implementation, we use a set of generic aspects to monitor the target application. Note that although the weaving of aspects into the application is performed at compilation time, policies may be changed at runtime. The extended PEP monitors the instantiation of classes, change in the values of attributes and method calls. The PDP is notified of change in the elements specified in role and action declarations.

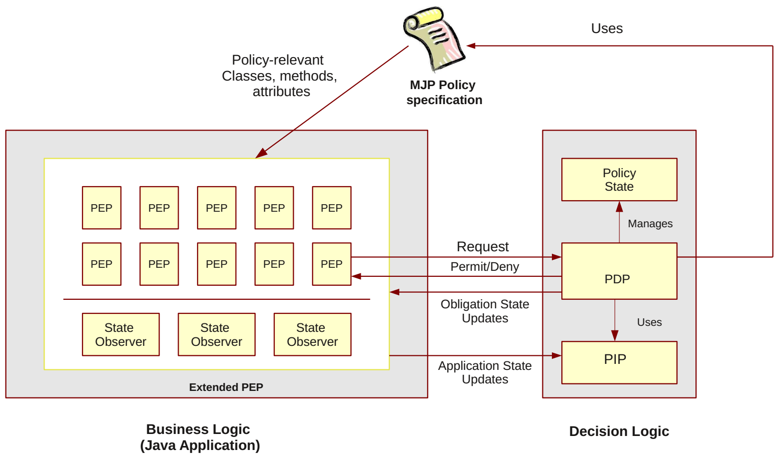

Fig. 2: Policy Enforcement Architecture

a) Access Control Enforcement: To enforce access control rules, we use a generic advice of type *around()*, i.e. method calls are interrupted and their execution is either skipped or interrupted if their execution is not authorized by the access control policy. Whether unauthorized method calls should be interrupted or skipped is specified in the policy[2].

B. Policy Decision Point

The stateful PDP is an application-independent module that enforces the access control policy and monitors and updates the states of obligations according to change notifications. The PDP logically consists of a Policy Information Point (PIP) where an internal first-order predicate logic representation of the target application state is maintained, the access-decision and obligation state-update logic and a policy state. The policy state includes applicable security rules, e.g. activated obligations. In our current implementation, the PDP is implemented using the production rule system *drools* [18]. Algorithm 1 sketches the operation of the PDP after the reception of a notification from the xPEP: First, the application state is updated and the operation rules are evaluated. Instance variables are then updated as specified in state update rules. The policy is then evaluated and updated accordingly.

Note our policies follow the access control and obligation models in [17], [19]. Interested readers are referred to these papers for the formal semantics of policies. Currently, only access control and obligations are supported. However, it is straightforward to add support for more advanced features such as delegation [20] and more advanced conflict detection and resolution strategies [21].

V. Empirical Assessment

We evaluated our implementation by studying policy enforcement in accordance with a policy specification for the ASMS application (122 classes, 797 methods and 10703 line of code). We also implemented the xPEP in Java/AspectJ [22],

Algorithm 1: PDP Operation

```
Input  : A change notification of type $T$ where $T \in$
         $\{$instance-creation, attribute-update,
         method-execution$\}$
Output: Access Decision and the set of activated,
         fulfilled, violated and canceled obligations
begin
    (1) Update_application_state;
            updates the application state facts;
    (2) Apply_operation_rules;
            identifies and updates the current operation;
    (3) Update_variables_update_rules;
            update the values of policy state variables;
    (4) Evaluate_Security_Policy;
            derives the new policy state;
    return the set of newly activated, violated, fulfilled
    and canceled obligations and the access control
    decision if T=method_execution;
end
```

which consists of a set of generic aspects, and the communication between the PEP and the PDP and modules to make a first-order representation of Java objects. Our implementation has 42 classes, 442 methods and 4485 lines of code. In the following, we refer to a policy specification as an MJP specification for a Mapping Java to security Policies specification.

To test our implementation, we have considered the scenario in Table VI: a user having the role of *personnel* creates a number of regular users, in addition to a *seller*, an *administrator* and a *moderator*. The *seller* creates an auction and opens it. Then regular users connect to the auction and they are selected in turn to post a comment. One every five users posts a comment containing a word included in the auction's blacklist. Consequently, the *moderator* becomes obliged to update the post. From every three obligations, the moderator fulfills two: once by entirely deleting the post and another time by deleting the blacklisted word from the post. The third obligation is violated. In this case, another obligation is activated for the

[2]When method execution is interrupted, a *runtime SecurityException* is thrown. Additional code may be needed to deal with these exceptions.

	FPL	**PU**	**CP**	**OS**	**RS**	**PS**	**CR**	**ASM**
SASI	✓	×	○	×	×	○	×	✓
JavaMOP	✓	×	✓	×	×	✓	○	✓
Polymer	×	×	○	×	×	✓	○	✓
SPoX	✓	×	○	×	×	○	×	✓
PoET	×	×	○	×	×	✓	×	✓
XACML	✓	✓	✓	○	×	×	✓	×
MJP	✓	✓	✓	✓	✓	✓	✓	✓

×: not supported, ✓: supported, ○: partially supported

FPL: Formal Policy Language, **PU**: Policy Update at Runtime, **CP**: Contextual Policies, **PS**: Policy State,
OS: User obligations, **RS**: Reaction policies, **CR**: Policy Conflict Management, **ASM**: Application-Specific Modules

TABLE VII: Comparison with Existing Frameworks

TABLE VI: MJP Specification

operation	**illustrates**
User posts a comment	Access allowance
Moderator required to delete the post	Oblig. activation
Moderator does not delete the post	Oblig. violation
Moderator updates the post	Oblig. cancellation
Moderator deletes the post	Oblig. fulfillment
Admin is obliged to delete the post	Reaction policy
When the number of comments posted by a user exceeds 50 comments	State Update
The user is forbidden to post comments	Access denial

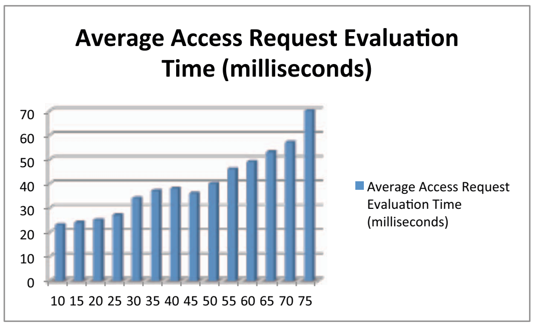

Fig. 3: Average Access Request Evaluation Time

administrator to delete the post. The post is then deleted by the administrator. When the number of comments posted by a user exceeds 50 comments, the user is forbidden to post comments. The scenario ends when all users exceed their comments limit.

Figure 3 shows the evolution of the average time necessary to evaluate access with the increase of number of users. There are several ways to improve the efficiency of policy management in our framework. For example, the current implementation maintains a large number of (unnecessary) facts to evaluate the policy. We are currently working on several optimizations which should largely improve policy evaluation and update times.

VI. Related Work

Several frameworks have been proposed to enforce security policies in Java applications. Table VII lists many of these frameworks and compares them with our approach MJP.

Security policy monitoring and enforcement using JavaMOP [23] is studied in [8]. JavaMOP enables the declaration of events in the form of AspectJ pointcuts. A set of properties can be specified using these events using various logical formalisms. In [8], handling the validation (violation) of a property and possible conflicts between handlers and their resolution are discussed. In our work, mapping policies are specified in a first-order providing them declarative semantics and simplifying their interpretation, specification and update.

SPoX [6], [7] is an XML-based formal security policy specification language. A security state in SPoX is a set of security-state variables with integer values. A SPoX specification defines a security automaton [9] where states of the automaton are the sets of security-state variables and their values and its edges are pointcut expressions. With respect to our work, policy state in SPoX only includes security-state variables whereas our policy state additionally includes application-state variables (policy-relevant instance attributes). Therefore, we are able to specify more contextual policies.

Security Automata Software-Fault Isolation (SFI) Implementation (SASI) [2] generalizes SFI to security policies specified as security automata. Two SASI prototypes have been implemented: one for the output of a GNU gcc C compiler and another for Java Virtual Machine Language. Two limitations to policy specification in SASI are highlighted in [2]: (1) the inability to store typed variables in the enforcement state and (2) the need to search for code sequences that correspond to the application-level abstractions in the case of x86 SASI. In comparison, our approach enables the keeping of both application and security policy variables. Moreover, we support obligation policies and can specify new security rules that only apply when obligations are violated.

XACML [11] defines a declarative attribute-based access control policy language. XACML supports the definition of obligations. However, XACML requires the development of application-specific modules to enforce and monitor obligations and to evaluate contextual policies. Our approach is more generic and does not require development of any application-specific module.

PoET/PSLang [1] is a successor to SASI. A PSLang specification includes a set of security events, a security state and security updates. PSLang has a Java-inspired syntax and is expressive enough to specify the EM policies of [9]. It does not however support obligation policies necessary to express usage control requirements [24].

Several other works studied the security of Java applications. For example, [5] uses AOP to weave into applications the code necessary to enforce rewrite-based access control policies [25]. In [26], RBAC [27] is realized in Java programs through the annotation of classes, methods and interfaces. [26] do not support the runtime modification of the policy nor contextual policies.

VII. Conclusion

In this paper, we have introduced a framework (MJP) to enable the integration of advanced security policies in Java applications. The framework enables the generation of a security architecture in Java applications supporting the specification of fine-grained contextual security policies. Thus, application developers have the choice between the specification of coarse-grained policies (on the level of classes and simple methods as considered in most frameworks today) or more fine-grained contextual policies (on the level of class instances and chains of method calls). Another important advantage of the framework is that policies can be specified and updated at runtime without requiring any modification to the target's application. In previous work, AOP has been used to directly integrate security mechanisms inside the target application making their change at runtime impossible. In our work, AOP is used for the monitoring of change in the application state. Therefore, the application developers may focus on the development of the business logic and then security controls can be later integrated and can be updated at runtime. The framework also supports the expression and enforcement of advanced security policies such as user obligations, history-based policies and reaction policies providing security officers the expressiveness needed to specify practical sophisticated policies.

This work can be extended in several ways. For example, the policy language can be extended to add support for more advanced usage controls [24]. The study of formal application analysis is another research direction. We also intend to incorporate different strategies to handle policy violation [8], [23] and study optimizations to improve the performance of MJP. The integration of more advanced conflict detection and resolution techniques is also envisaged.

References

[1] U. Erlingsson and F. Schneider, “IRM Enforcement of Java Stack Inspection,” *IEEE Symposium on Security and Privacy*, pp. 246–255, 2000.

[2] U. Erlingsson and F. B. Schneider, “SASI enforcement of security policies,” *NSPW*, pp. 87–95, 2000.

[3] J. Ligatti, L. Bauer, and D. Walker, “Enforcing Non-safety Security Policies with Program Monitors,” *ESORICS'05 Proceedings of the 10th European conference on Research in Computer Security*, pp. 355–373, 2005.

[4] L. Bauer, J. Ligatti, and D. Walker, “Composing security policies with polymer,” *ACM SIGPLAN Notices*, vol. 40, no. 6, p. 305, Jun. 2005.

[5] A. S. de Oliveira, E. K. Wang, C. Kirchner, and H. Kirchner, “Weaving rewrite-based access control policies,” *FMSE*, pp. 71–80, 2007.

[6] K. W. Hamlen and M. Jones, “Aspect-oriented in-lined reference monitors,” *PLAS*, p. 11, 2008.

[7] M. Jones and K. W. Hamlen, “Enforcing IRM security policies: Two case studies,” *2009 IEEE International Conference on Intelligence and Security Informatics*, pp. 214–216, 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5137306

[8] S. Hussein, P. Meredith, and G. Rolu, “Security-policy monitoring and enforcement with JavaMOP,” *PLAS*, pp. 1–11, 2012.

[9] F. B. Schneider, “Enforceable security policies,” *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, Feb. 2000.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-m. Loingtier, J. Irwin, and C. Lopes, “Aspect-Oriented Programming,” *ECOOP European Conference on Object-Oriented Programming*, no. June, 1997.

[11] “extensible access control markup language (xacml) version 3.0,” http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf, 2012, retrieved: 07/09/2012.

[12] A. Pretschner, T. Mouelhi, and Y. L. Traon, “Model-based tests for access control policies,” in *ICST*, 2008, pp. 338–347.

[13] US Congress, “Health Insurance Portability and Accountability Act of 1996,” pp. 1–169, 1996.

[14] The European Parliment and the Council, “Directive 1995/46/EC of the european parliment and the council of 24 october 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data,” *Official Journal of the European Communities*, 1995.

[15] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, “Role-based access control models,” *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[16] A. Abou El Kalam, S. Benferhat, A. Miège, R. El Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin, “Organization based access control,” *POLICY*, vol. 0, p. 120, 2003.

[17] Y. Elrakaiby, F. Cuppens, and N. Cuppens-Boulahia, “Formal enforcement and management of obligation policies,” *Data & Knowledge Engineering*, pp. 1–21, Sep. 2011.

[18] “Drools - the business logic integration platform,” http://www.jboss.org/drools/drools-expert.html, retrieved: 17/09/2012.

[19] F. Cuppens and N. Cuppens-Boulahia, “Modeling contextual security policies,” *International Journal of Information Security*, vol. 7, no. 4, pp. 285–305, Nov. 2007.

[20] M. Ben-Ghorbel-Talbi, F. Cuppens, N. Cuppens-Boulahia, and A. Bouhoula, “A delegation model for extended RBAC,” *IJIS*, vol. 9, no. 3, pp. 209–236, May 2010.

[21] F. Cuppens, N. Cuppens-Boulahia, and M. B. Ghorbel, “High Level Conflict Management Strategies in Advanced Access Control Models,” *Electronic Notes in Theoretical Computer Science*, vol. 186, pp. 3–26, Jul. 2007.

[22] R. Laddad, *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co, 2009.

[23] F. Chen and G. Ro, “Java-MOP: A monitoring oriented programming environment for Java,” *Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[24] R. Sandhu and J. Park, “The UCON ABC usage control model,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 1, pp. 128–174, 2004.

[25] D. J. Dougherty, C. Kirchner, H. Kirchner, and A. S. de Oliveira, “Modular access control via strategic rewriting,” *ESORICS*, pp. 578–593, 2007.

[26] J. Zarnett, M. Tripunitara, and P. Lam, “Role-based access control (RBAC) in Java via proxy objects using annotations,” *SACMAT*, p. 79, 2010.

[27] R. Sandhu, E. Coyne, and H. Feinstein, “Role-Based Access Control Models,” *Computer*, vol. 29, no. 2, pp. 38–47, 1996.