Refactoring Access Control Policies for Performance Improvement

Donia El Kateb
University of Luxembourg
6 rue Coudenhove-Kalergi
L-1359 Luxembourg
donia.elkateb@uni.lu

Tejeddine Mouelhi
University of Luxembourg
6 rue Coudenhove-Kalergi
L-1359 Luxembourg
tejeddine.mouelhi@uni.lu

Yves Le Traon
University of Luxembourg
6 rue Coudenhove-Kalergi
L-1359 Luxembourg
yves.letraon@uni.lu

JeeHyun Hwang
Dept. of Computer Science,
North Carolina State
University
Raleigh, NC 27695, U.S.A
jhwang4@ncsu.edu

Tao Xie
Dept. of Computer Science,
North Carolina State
University
Raleigh, NC 27695, U.S.A
xie@csc.ncsu.edu

ABSTRACT

In order to facilitate managing authorization, access control architectures are designed to separate the business logic from an access control policy. An access control policy consists of rules that specify who have access what resources. A request is formulated from components, called Policy Enforcement Points (PEPs). Given a request, a Policy Decision Point (PDP) evaluates the request against an access control policy and returns its access decision (i.e., Permit or Deny) to the PEPs. With the growth of sensitive information for protection in an application, an access control policy consists of larger number of rules, which often cause a performance bottleneck. In order to address this issue, we propose an approach to refactoring access control policies for performance improvement by splitting an access control policy (traditionally handled by a single PDP) into its corresponding multiple access control policies each with a smaller number of rules (handled by multiple PDPs). We define seven attribute-set-based splitting criteria to facilitate splitting an access control policy. We have conducted an evaluation on three subjects of real-life Java programs, each of which interacts with access control policies. Our evaluation results show that (1) our proposed approach preserves policy behaviors (i.e., which responses are evaluated in terms of requests) of the subjects after refactoring, and (2) our approach is efficient in terms of reducing request evaluation time by up to nine times.

Keywords

Access Control, Performance, Refactoring, Policy Enforcement Point, Policy Decision Point, eXtensible Access Control Markup Language

1. INTRODUCTION

Access control mechanisms regulate which users could perform which actions on what system resources based on access control policies. Access control policies (i.e., policies in short) are based on various access control models such as Role-Based Access Control (RBAC) [7], Mandatory Access Control (MAC) [5], Discretionary Access Control (DAC) [9], and Organization-Based Access Control (OrBAC) [**?**]. Access control policies are specified in various policy specification languages such as the eXtensible Access Control Markup Language (XACML) [2] and Enterprise Privacy Authorization Language (EPAL) [**?**]. In a policy, a rule specifies actions (e.g., read) that subjects (e.g., students) can take on resources (e.g., grades) if required conditions are met.

A policy-based system is a system that interacts with user-specified policies to determine users' access on resources. A policy-based systems allows policy authors to define rules in a policy. In the context of policy-based systems, access control architectures are often designed with respect to a popular architectural concept that separates Policy Enforcement Points (PEPs) from a Policy Decision Point (PDP) [18]. More specifically, a PEP is located inside an application's code (i.e., business logic of the system). Business logic describes functional algorithms to govern information exchange between access control decision logic and a user interface (i.e., presentation).

Given requests (e.g., student A requests to read her grade resource B) formulated by the PEP, the PDP evaluates the requests and returns their responses (e.g., permit or deny) by evaluating these requests against rules in a policy. We define policy behaviors as request/response behaviors (i.e., which responses are evaluated in terms of requests) observed from external behaviors of access control decision logic. An important benefit of the architecture is to facilitate managing access rights in a fine-grained way by decoupling the business logic from the access control decision logic, which can be standardized.

However, this architecture may cause performance degrade especially when policy authors maintain a single policy with a large number of rules to regulate a whole system's resources. Various factors such as complex and dynamic behaviors of organizations and the growth of organizations's assets may increase the number of rules in the policy [6]. Consider that the policy is centralized into *only* one single PDP. The PDP evaluates requests (issued by PEPs) against the large number of rules in the policy in real-time. Such centralization can be a major factor for degrading performance as our previous work showed that the large number of rules is a challenge for efficient request evaluation [11]. This performance bottleneck issue may impact service availability as well, especially when dealing with a huge number of requests within a short time.

In order to address this performance bottleneck issue, we propose an approach to refactoring policies automatically to significantly reduce request evaluation time. As manual refactoring is tedious and error-prone, an important benefit of our automated approach is to reduce significant human efforts as well as improving performance. Our approach includes two techniques: (1) refactoring an access control policy (handled by single PDP) into its corresponding multiple access control policies each with a smaller number of rules (handled by multiple PDPs), and (2) refactoring PEPs with regards to the refactored PDPs while preserving architectural property that a single PDP is triggered by a given PEP at a time.

In the first technique, our approach takes a splitting criterion and an original global policy (i.e., a policy governing all of access rights in the system) as an input, and returns a set of corresponding sub-policies, each of which consists of a smaller number of rules. This refactoring involves grouping rules in the global policy into several subsets based on the splitting criterion. More specifically, we propose a set of splitting criteria to refactor the global policy into smaller policies. A splitting criterion selects and groups the rules of the overall PDP into specific PDPs. Each criterion-specific PDP encapsulates a sub-policy that represents a set of rules that share a combination of attribute elements (Subject, Action, and/or Resource).

In the second technique, our approach aims at preserving the architectural property, which represents that only single PDP is triggered by a given PEP at a time. Our approach refacors PEPs according to multiple PDPs loaded with sub-policies while complying policy behaviors of the centralized architectures and preserving the architectural property. More specifically, given a request, each PEP should be mapped to a PDP loaded with a policy, which includes a set of rules to be applicable for the request. Therefore, our refactoring maintains architectural property of the centralized architectures in policy-based systems.

We collect three subjects of real-life Java programs, each of which interacts with access control policies. The policies consist of a large number of rules. The largest one has 1760 rules, which lead performance degrade during evaluation. The policies are specified in XACML. XACML become an XML-based policy specification language popularly used for web-based applications and services. While our subjects are based on XACML policies, our approach could be applicable to any program, which interact policies specified in other policy specification languages for the following reason: As a large number of rules is performance bottleneck for request evaluation against given policies (specified in other policy specification languages), our approach could reduce the number of rules in a systemic way for request evaluation against the policies to improve performance as well.

We conduct an evaluation to show performance of our approach in terms of request evaluation time. We leverage two PDPs to measure request evaluation time. The first one is the Sun PDP implementation [1], which is a popular open source PDP and the second one is XEngine [11], which transforms an original policy into its corresponding policy in a tree format by mapping attribute values with numerical values. Our evaluation results also show that our approach preserves the policy behaviors of the centralized architecture and the architectural property. Our evaluation results show that our approach is efficient in terms of reducing request evaluation time by up to nine times.

This paper makes the following three main contributions:

- We propose an automated approach that refactors a single global policy into policies each with a smaller number of rules. This refactoring helps improve performance of request evaluation time.
- We propose a set of splitting criteria to help refactor a policy in a systematic way. Our proposed splitting criteria do not alter policy behaviors of the centralized architectures.
- We conduct an evaluation on three Java programs interacting with XACML policies. We measure performance in terms of request evaluation time. Our evaluation results show that our approach achieves more than nine times faster than that of the centralized access control architectures in terms of request evaluation time.

The remainder of this paper is organized as follows. Section 2 introduces concepts related to our research problem addressed in this paper. Section 3 presents the overall approach. Section 4 presents evaluation results and discusses the effectiveness of our approach. Section 5 discusses related work. Section 6 concludes this paper and discuses future research directions.

2. CONTEXT/PROBLEM STATEMENT

This section further details the centralized access control architecture as well as its desirable feature (synergy, reconfigurability) and its weakest ones (performance bottlenecks). Managing access control policies is one of the most challenging issues faced by organizations. Frequent changes in policy-based systems may be required to meet business needs. A policy-based system has to handle some specific requirements like role swapping when employees are given temporary assignments, changes in the policies and procedures, new assets, users and job positions in the organization.

2.1 Centralization of Access Control Architectures

To enable high reconfigurability, the access control policy is traditionally modeled, analyzed and implemented as a separate component encapsulated in a PDP. This leads to the centralized architecture presented in Figure 1, in which one PDP is responsible for granting/denying the accesses that are requested. This centralized architecture is a simple solution to easily handle changes in policy-based systems by having an immediate translation from the policy author to the PDP. Reconfiguration consists of modifying the PDP accordingly to the changes in the access control policy. The separation between the PEP and the PDP simplifies policy management across many heterogeneous systems and limits potential risks arising from incorrect policy implementation, when the policy is hardcoded inside the business logic.

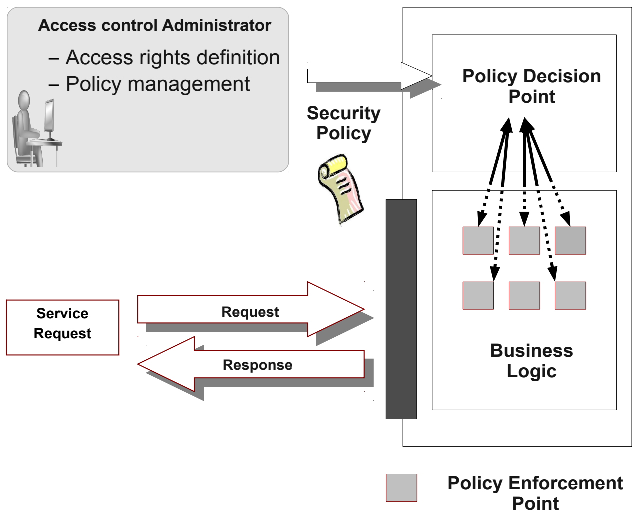


Figure 1: Access Control Request Processing

2.2 Centralization: A Threat for performances

In such a system (Figure 1), when the service execution requires an access control, the PEP calls the PDP to retrieve an authorization decision based on the PDP encapsulated policy. This authorization decision is made through the evaluation of rules in the policy. Subsequently, an authorization decision (permit/deny) is returned to the PEP. When a huge number of access requests are sent to the PDP, two bottlenecks cause a degradation of performances:

- 1) all the access requests have to be managed through the same input channel of the PDP.
- 2) the centralized PDP computes an access request by searching which rule is applicable among all the rules it contains.

The execution time for the treatment of a request is thus strongly related to:

- the number of access rules the PDP contains as well as to the ordering of the rules into a PDP [12].
- the workload (number of requests).

The execution time to treat one request depends on the size (number of rules) the PDP contains. For a given PDP size, the execution time to treat requests increases linearly with the workload (number of requests). Our hypothesis is: the more rules a PDP contains, the higher the slope gradient of the execution time submitted to an increasing workload. *Hypothesis 1* validity is studied in section 4. As a consequence, one possibility to increase performances consists in splitting the centralized PDP into PDPs of smaller sizes.

2.3 Centralization allows a good synergy between PEPs and PDP

Centralization offers a desirable feature by simplifying the routing of requests to the right PDP. Figure 2 illustrates the model of the access control architecture. In this model, a set of business processes, which comply to users' needs, are encapsulated in a given business logic which is enforced by multiple PEPs. Conceptually, the decision is decoupled from the enforcement and involves a decision making process in which each PEP interacts with one single PDP. The keypoint concerns the cardinality linking PEPs to the PDP. While a PDP is potentially linked to many PEPs, any PEP is strictly linked to exactly one PDP (which is unique in the centralized model). Since there is only one PDP, the requests are all routed to this unique PDP. This means that no particular treatment is required to map a given PEP in the business logic to the corresponding PDP, embedding the requested access rule. Another advantage of this many-to-one association is the clear traceability between what has been specified by the policy at the decision level and the internal security mechanisms enforcing this policy at the business logic level. In such setting, when access control policies are updated or removed, the related PEPs can be easily located and removed. Thus the application is updated synchronously with the policy changes. We call this desirable property *synergy* of the access control architecture: an access control architecture is said to be *synergic* if any PEP always sends its requests to the same PDP. As a consequence, splitting the centralized PDP into PDPs of smaller sizes may break this "synergy". In the following, we will consider various ways to split a centralized PDP into smaller PDPs. The related *hypothesis 2* is: with comparable PDP sizes, the overall system will be more performant when the architecture is synergic. This hypothesis is questioned in section 4.

2.4 Tradeoff for refactoring a centralized architecture into a decentralized one

As a synthesis for this section, the following facts are taken into account:

- Access control architectures are centralized with a unique PDP
- Centralization eases reconfiguration of access control policy
- Centralization threatens performances
- Direct mapping from any PEP to only one PDP makes the access control architecture "synergic"
- A synergic system facilitates PEPs request routing and eases access control policy updates

The goal of this paper is to propose systematic ways to improve performance by refactoring the centralized model into a decentralized version, with multiple PDPs. The resulting architecture must be functionally equivalent and should not impact the desirable properties of the centralized model, namely reconfigurability and synergy. Automating the transformation from a centralized to a decentralized architecture is required to preserve reconfigurability. With automation, we can still reconfigure the centralized policy, and then automatically refactor the architecture. Automated refactoring is thus a viable solution for having high reconfigurability. However, refactoring the architecture by splitting the centralized PDP into smaller one may break the initial synergy. This phenomenon is studied in the empirical study of section 4 together with *hypothesis 2*. We describe the XACML Language since it is the standard language used to implement a PDP.

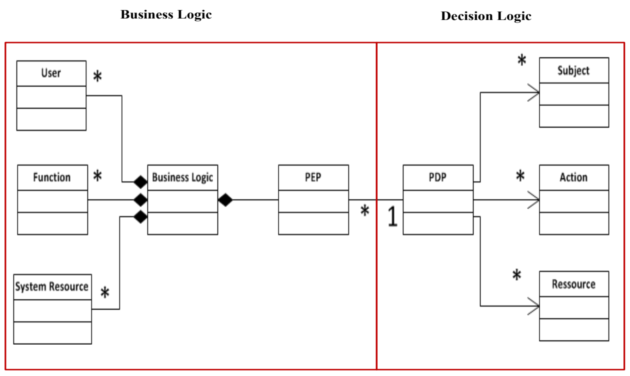


Figure 2: The Access Control Model

2.5 XACML Access Control Policies and Performance Issues

In this paper, we focus on access control policies specified in the eXtensible Access Control Modeling Language (XACML) [2]. XACML is an XML-based standard policy specification language that defines a syntax of access control policies and request/response. XACML enables policy authors to externalize access control policies for the sake of interoperability since access control policies can be designed independently from the underlying programming language or platform. Such flexibility enables to easily update access control policies to comply with new requirements.

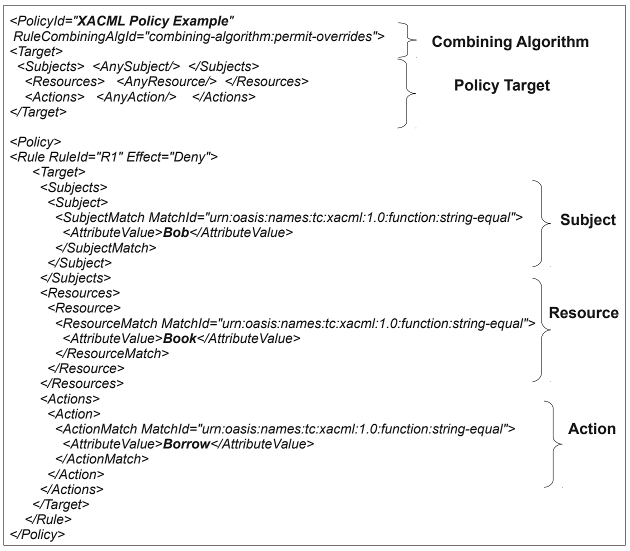
An XACML is constructed as follows. A `policy set` element consists of a sequence of `policy elements`, a combining algorithm, and a `policy target` element. A `policy element` is expressed through a target, a set of `rules`, and a rule combining algorithm. A `target` element consists of the set of resources, subjects, and actions to which a rule is applicable. A `rule` consists of a `target` element, a `condition` element, and an `effect`. A `condition` element is a boolean expression that specifies the environmental context (e.g., time and location restrictions) in which the rule applies. Finally, an `effect` is the rule's authorization decision, which is either permit or deny.

Given a request, a PDP evaluates the request against the `rules` in the policy by matching resources, subjects, actions, and condition in the request. More specifically, an XACML request encapsulates attributes, which define which subject requests to take action on which resource in which condition (e.g., subject Bob requests to borrow a book). Given a request, that satisfies `target` and `condition` elements in a rule, the rule's effect is taken as the decision. If the request does not satisfy `target` and `condition` elements in any rule, its response yields the "NotApplicable" decision.

When more than one rule is applicable to a request, the combining algorithm helps determine which rule's effect can be finally given as the decision for the request. For example, given two rules, that are applicable to the same request and provide different decisions, the permit-overrides algorithm prioritizes a permit decision over the other decisions. More precisely, when using the permit-overrides algorithm, the policy evaluation produces one of the following three decisions:

- Permit if at least one permit rule is applicable for a request.
- Deny if no permit rule is applicable and at least one deny rule is applicable for a request.
- NotApplicable if no rule is applicable for a request.

Figure 3 shows a simplified XACML policy that denies subject Bob to borrow a book.



```
<PolicyId="XACML Policy Example"
 RuleCombiningAlgId="combining-algorithm:permit-overrides">
<Target>
 <Subjects>  <AnySubject/>  </Subjects>
   <Resources>   <AnyResource/>  </Resources>
   <Actions>   <AnyAction/>    </Actions>
</Target>

<Policy>
<Rule RuleId="R1" Effect="Deny">
      <Target>
        <Subjects>
          <Subject>
            <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue>Bob</AttributeValue>
            </SubjectMatch>
          </Subject>
        </Subjects>
        <Resources>
          <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue>Book</AttributeValue>
            </ResourceMatch>
          </Resource>
        </Resources>
        <Actions>
          <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
              <AttributeValue>Borrow</AttributeValue>
            </ActionMatch>
          </Action>
        </Actions>
      </Target>
    </Rule>
</Policy>
```

Combining Algorithm
Policy Target
Subject
Resource
Action

Figure 3: XACML Policy Example

Recently, an XACML policy becomes more complex to handle increasing complexity of organizations in terms of structure, relationships, activities, and access control requirements. In such a situation, the policy often consists of a large number of rules to specify policy behaviors for various resources, users and actions in the organizations. In policy-based systems, policy authors manage centralized a single PDP loaded with a single policy to govern all of system resources. However, due to a large number of rules for evaluation, this centralization raises performance concerns related to request evaluation time for XACML access control policies and may degrade the system efficiency and slow down the overall business processes.

We present following three main factors, that may cause to degrade XACML request evaluation performance:

- An XACML policy may contain various attribute elements including `target` elements. Retrieval of attribute values in the `target` elements for request evaluation may increase the evaluation time.
- A `policy set` consists of a set of policies. Given a request, a PDP determines the final authorization decision (i.e., effect) of the whole `policy set` after combining all the applicable rules' decisions according to the request. Computing and combining applicable rules' decisions contributes to increase the evaluation time.

- `Condition` elements in rules can be complex because these elements are built from an arbitrary nesting of non-boolean functions and attributes. In such a situation, evaluating `condition` elements may slow down request evaluation time.

3. XACML POLICY REFACTORING PROCESS

This section describes our approach of refactoring access control policies to improve performance by reducing the number of policy rules potentially applicable to the request. For refactoring policies in a systematic way, we propose seven policy splitting criteria based on attribute set. Moreover, we explain how to select the splitting criterion, which preserves the synergy in the access control architecture.

3.1 Definition of Policy based Splitting Criteria

Given a request, a PDP uses brute force searching to retrieve the decision, by evaluating the request against all the policy rules. For request evaluation processing, not of all the rules are applicable to the request. In other words, only part of the rules (i.e., relevant rules) are applicable to the request and contribute to build the final authorization decision. We propose an approach to evaluate a request against only the relevant rules for a given request by refactoring access control policies. Our approach aims at splitting a single global policy into multiple smaller policies based on attribute values combination. For a given policy-based system, we transform the policy P into policies P_{SC_w} containing less number of rules and conforming to a Splitting Criteria SC_w. An SC_w defines the set of attributes that are considered to classify all the rules into subsets having similar one or more attribute values, w denotes the number of attributes that have to be considered conjointly for aggregating rules based on specific attribute elements selection. Table 1 shows our proposed splitting criteria categories according to the attribute elements combination.

Categories	Splitting Criteria
SC_1	$\langle Subject\rangle, \langle Resource\rangle, \langle Action\rangle$
SC_2	$\langle Subject, Action\rangle, \langle Subject, Resource\rangle$ $\langle Resource, Action\rangle$
SC_3	$\langle Subject, Resource, Action\rangle$

Table 1: Splitting Criteria

Figure 4 presents a splitting process example in which an XACML policy P is refcatored according to the splitting criterion $SC_1 = \langle Subject\rangle$. The refactoring results in two sub-policies Pa and Pb where each consists in rules having respectively the same subject (Alice or Bob in this case).

The Algorithm below illustrates the splitting process for $SC_1 = \langle Subject\rangle$.

It is worth mentioning the following issues related to the refactoring process:

- XACML supports multi-valued attributes in policies and requests. In XACML, `Target` elements define a set of attribute values, which match with the context element in an access control request. In Figure 5, subject attribute includes two attributes (one is "role" and the other is "isEq-subjUserId-resUserId"). In order to match the subject with multi-valued attributes, a request should include at least `pc-member` and `ture` for "role" and "isEq-subjUserId-resUserId", respectively. Our approach considers such a whole subject element as a single entity, which is not splitted the policy splitter component.

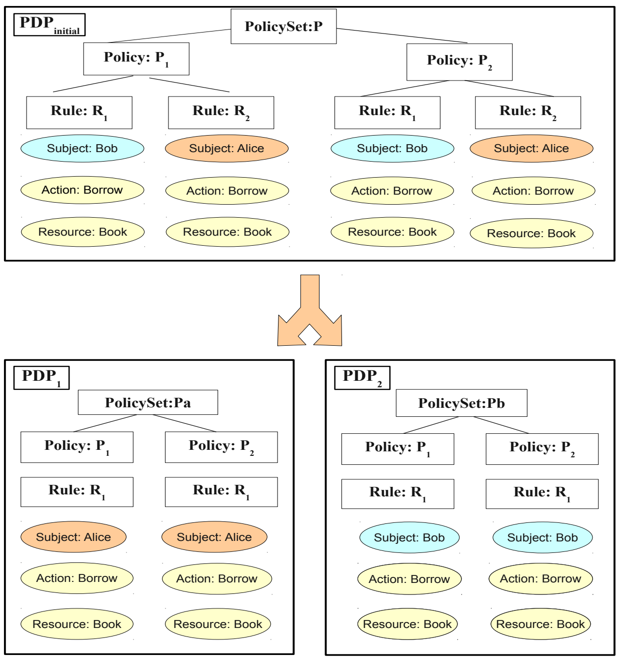


Figure 4: XACML policy refactoring according to $SC_1 = \langle Subject\rangle$

- XACML supports multi-valued attributes in policies and requests. For example, a subject in a given request could be both a manager role and a employee role. We don't consider multi-valued attributes in this paper.

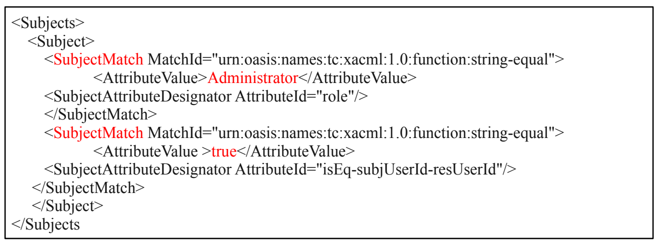

```
<Subjects>
   <Subject>
      <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue>Administrator</AttributeValue>
      <SubjectAttributeDesignator AttributeId="role"/>
      </SubjectMatch>
      <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue >true</AttributeValue>
      <SubjectAttributeDesignator AttributeId="isEq-subjUserId-resUserId"/>
   </SubjectMatch>
   </Subject>
</Subjects
```

Figure 5: Multi-attributes values in `target` element

After the splitting process is performed, our approach creates one or more (PDPs) that comply with a certain splitting criterion. We use SUN PDP [2] that evaluates requests against the policies specified in XACML. During request evaluation, SUN PDP checks the

Algorithm 1 Policy Splitting Algorithm

```
Input: XACML Policy $P$, Splitting Criteria $SC_1=\langle Subject\rangle$
Output: Sub-policies Set: S
SplitPolicy()
/* Collect all subjects in all the rules /*
for all $R_i \in P$ do
  /* Fetch all the targets to extract attributes collections depending on SC */
  for all Target t in $R_i$ do
    if t.contains(SubjectElement) then
      SubjectCollection.add(SubjectElement.attribute)
    end if
  end for
end for
/* Build sub-policies based on subjects collected in SubjectCollection */
for int i = 0; i < SubjectCollection.size(); ++i do
  /* Remove all the rules that do not contain SubjectCollection.at(i) in their Targets*/
  for all $R_i \in P$ do
    BOOL remove=TRUE
    for all Target t in $R_i$ do
      if t.SubjectElement.attribute in $R_i$ equals SubjectCollection.at(i) then
        remove=FALSE
      end if
    end for
    if remove.equals(True) then
      Remove $R_i$
    end if
  end for
  /* $P_{(SubjectCollection.at(i))}$ is a sub-policy having only rules where the subjectAttribute is equals to SubjectCollection.at(i) */
  $S = S + P_{(SubjectCollection.at(i))}$
end for
```

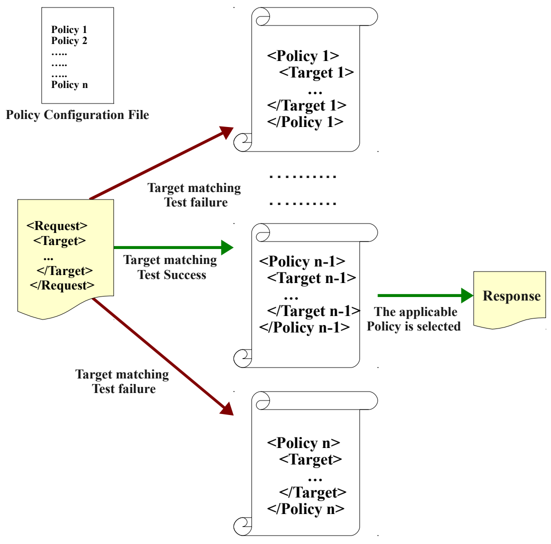

Policy 1
Policy 2
.....
.....
.....
Policy n
Policy Configuration File
<Policy 1> <Target 1> ... </Target 1> </Policy 1>
....
....
Target matching Test failure
<Request> <Target> ... </Target> </Request>
Target matching Test Success
<Policy n-1> <Target n-1> ... </Target n-1> </Policy n-1>
The applicable Policy is selected
Response
Target matching Test failure
<Policy n> <Target> ... </Target> </Policy n>

Figure 6: Applicable policy Selection

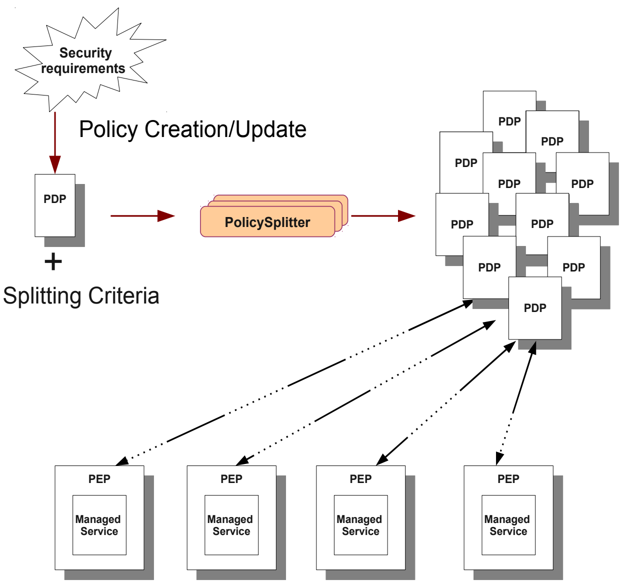


Figure 7: Overview of the Refactoring Process

request against the policy and determines whether its authorization decision is permit or deny. Given a request, our approach fetches a SUN PDP loaded with the relevant policy, which is used during the decision making process. The PDP then retrieve the applicable rules that are applicable to the request. Figure 6 presents our approach to handle request evaluation with multiple policies. During the evaluation process, given a request, our approach verifies the matching between the request's attributes and the policy set attributes. Our approach then selects only relevant policy among all of policies for a given request. After the selection of the relevant policy, all of its relevant rules for the decision making are evaluated.

Figure 7 shows a general overview of the process. Given a single PDP loaded with initial global policy, the policy splitter tool conducts automated refactoring process by creating multiple PDPs loaded with XACML policies, which are split from the initial global policy based on user specified SC. If the initial global policy is updated, the policy splitter is required to refactor the policy again to create PDPs with the most recent relevant policies. Our refactoring approach is safe in the sense that the approach does not impact existing functional aspects in a given system.

3.2 Architecture Model Preservation: PEP-PDP Synergy

We consider the different splitting criteria that we have identified in the previous section and we propose to select the splitting criterion that enables to preserve the synergy in the access control architecture, this splitting criterion enables to have a valid refactoring and respects how PEPs are organized at the application level and how they are linked to their corresponding PDPs. In the worst case, splitting the initial PDP into multi-PDPs may lead to a non-synergic system: a PEP may send its requests to several PDPs. The PDP, which receives a request is only known at runtime. Such a resulting architecture breaks the PEP-PDP synergy and the conceptual simplicity of the initial architecture model. In the best case, the refactoring preserves the simplicity of the initial architecture, by keeping a many-to-one association from PEPs to PDPs. A given request evaluation triggered by one PEP will always be handled by the same PDP. Operationally, the request evaluation process will involve one XACML policy file. In this case, the refactoring is valid, since its does not impact the conceptual architecture of the system. Figure 8 presents a PDP encapsulating a global policy that has been refactored. The system that is presented in the left is a valid refactoring whereas the one in the right shows a non valid refactoring. A deep analysis of the PEPs at the application enables

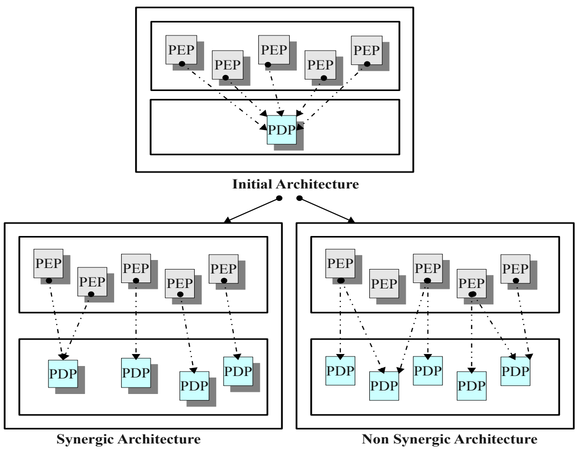


Figure 8: Synergic vs Non synergic System

to observe the mapping between the PEPs and the PDP. At the application level, the PEP is represented by a method call that triggers a decision making process by activating some specific rules in the PDP. The code in Figure 9 is taken from [17], this code excerpt shows an example of a PEP represented by the method checkSecurity which calls the class SecurityPolicyService that initiates the PDP component.

The PEP presented by the method `ServiceUtils.checkSecurity` may issue requests that have subject user role along fixed action and resource ("LibrarySecurityModel.BORROWBOOK_METHOD"), ("LibrarySecurityModel.BOOK_VIEW"). Consider that we refactor a policy according to $SC_2 = \langle Resource, Action \rangle$. Give a request issued from the PEP, our approach fetches a PDP loaded with a policy containing rules having those action and resource attributes.

```
public void borrowBook(User user, Book book) throws
SecuritPolicyViolationException {

// call to the security service
            ServiceUtils.checkSecurity(user,
LibrarySecurityModel.BORROWBOOK\_METHOD,
LibrarySecurityModel.BOOK\_VIEW);
ContextManager.getTemporalContext());}

  // call to business objects
  // borrow the book for the user
book.execute(Book.BORROW, user);
  // call the dao class to update the database
bookDAO.insertBorrow(userDTO, bookDTO);}
```

Figure 9: PEP deployment Example

Thus the splitting process that will preserve the mapping between the PEPs and the PDP is $SC_2 = \langle Resource, Action \rangle$ since the rules in the policy are triggered by Action, Resource. Depending on the organization of the PEPs in a given application, connecting the rules with their PEPs at the application level may require to identify all the enforcements points in the application, and to track the different method calls triggered from these specific enforcement calls to map them to the relevant access control rules.

Our empirical results, presented in section 4, have shown that adopting a policy refactoring based on system functions, as a refactoring strategy, enables to have the best splitting criterion in term of performance.

4. EVALUATION

We carried out our evaluation on a desktop PC, running Ubuntu 10.04 with Core i5, 2530 Mhz processor, and 4 GB of RAM. We have implemented a tool, called `PolicySplitter` to split the policies according to given splitting criteria automatically. The tool is implemented in Java and is available for download from [3]. To measure the efficiency of our approach, we conducted evaluation as follows. We compared request processing time with a single global policy (handled by a single PDP) and policies splitted by each splitting criterion. We first conduct an evaluation on subjects (including PEPs and SUN PDPs) to evaluate the performance improvement in terms of decision making processing. `IA` denotes an "Initial Architecture", which uses the single global policy for request processing time. We next conduct an evaluation on modified subjects, which replace SUN PDPs with a novel request evaluation engine, XEngine [11]. The objective of this evaluation is to investigate how our approach impacts performance for subjects combined with XEngine.

4.1 Subjects

The subjects include three real-life Java programs each which interact with access control policies [16]. We next describe our three subjects.

- Library Management System (LMS) provides web services to manage books in a public library.
- Virtual Meeting System (VMS) provides web conference services. VMS allows users to organize online meetings in a distributed platform.
- Auction Sale Management System (ASMS) allows users to buy or sell items on line. A seller initiates an auction by submitting a description of an item she wants to sell with its expected minimum price. Users then participate in bidding process by bidding the item. For the bidding on the item, users have enough money in her account before bidding.

Our subjects are initially equipped with Sun PDP [2], which is a popularly used PDP to evaluate requests. Policies in LMS,VMS, and ASMS contain a total of 720, 945, and 1760 rules, respectively. Moreover, to compare performance improvement over existing PDPs, we adopt XEngine (instead of Sun PDP) in our subjects to evaluate requests. XEngine is a novel policy evaluation engine, which transforms the hierarchical tree structure of the XACML policy to a flat structure to improve request processing time. XEngine also handles various combining algorithms supported by XACML.

4.2 Objectives and Measures

In the evaluation, we intend to answer the following research questions:

1. RQ1: Can our approach preserve policy behaviors of original subjects after refactoring? This question helps to show that our approach does not alter policy behaviors after refactoring based on splitting criteria.

2. RQ2: How faster request processing time of multiple Sun PDPs with policies splitted by our approach compared to an existing single Sun PDP? This question helps to show that our approach can improve performance in terms of request processing time. Moreover, we compare request processing time for different splitting criteria.

3. RQ3: How faster request processing time of XEngine compared to that of Sun PDP for both a single policy and policies splitted by our approach. This question helps to show that our approach can improve performance in terms of request processing time for other policy evaluation engines such as XEngine.

In our evaluation, we measure request processing time by evaluating randomly generated requests developed by our previous work [11]. In particular, for multiple PDPs, our approach fetches a PEP with a corresponding PDP for a given request at run time. Therefore, request processing time includes both fetching time and request evaluation time.

4.3 Performance Improvement Results

We generated the resulting sub-policies for all the splitting criteria defined in Section 3.1. For each splitting criteria, we have conducted systems tests to generate requests that trigger all the PEPs in the evaluation. The test generation step leads to the execution of all combination of possible requests described in our previous work [16]. The process of tests generation is repeated for ten times in order to avoid the impact of randomness. We applied this process to each splitting criterion and calculated evaluation time on average of a system under tests.

Figure 12 shows our evaluation results. Our results show that evaluation time for policies splitted based on each splitting criterion and the global policy of subjects. In order to answer RQ1, we compared decisions against the PDP in `IA` and multiple PDPs with splitted policies. consistent with our expected results.

In order to answer RQ2, from the results in Figure 12, we observed the followings:

- Compared to evaluation time of `IA`, our approach improves performance for all of splitting criteria in terms of evaluation time. This observation is consistent with our expected results; evaluation time against policies with smaller number of rules (compared with the number of rules in `IA`) is faster than that against policies in `IA`.

- The splitting criteria $SC = \langle Action, Resource\rangle$ enables to show the lowest evaluation time. Recall that the PEPs in the evaluation are scattered across different methods in a subject by a categorization that is based on $SC_2 = \langle Resource, Action\rangle$.

	SAR	AR	SA	SR	R	S	A	IA
SUN	485	922	1453	1875	2578	2703	2703	2625
XEn	26	47	67	95	190	164	120	613

Table 2: Evaluation time in LMS

	SAR	AR	SA	SR	R	S	A	IA
SUN	1281	2640	3422	3734	6078	5921	6781	5766
XEn	34	67	96	145	384	274	149	265

Table 3: Evaluation time in VMS

	SAR	AR	SA	SR	R	S	A	IA
SUN	2280	2734	3625	8297	7750	8188	6859	7156
XEn	49	60	104	196	310	566	262	1639

Table 4: Evaluation time in ASMS

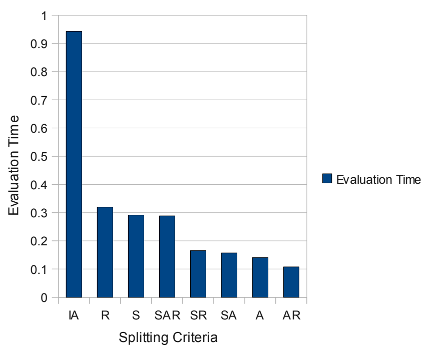
This observation pleads in favor of applying a splitting criteria that takes into account the PEP-PDP synergy and show the best performance.

Figure 11 shows the number of policies splitted by our approach. We observed the number of policies based on our proposed three categories: (1) SC_1 category leads to the smallest number N_1 of PDPs, (2) SC_2 category produces a reasonable number N_2 (N_1<N_2<N_3) of PDPs, and (3) SC_3 leads to the largest number N_3 of PDPs. While SC_1 category leads to the smallest number of PDPs, each PDP encapsulates a relatively high number of rules in a policy (compared with that of SC_2 and SC_3, which leads to performance degrade. We observed that SC_3 category leads to the largest number of PDPs, which takes higher fetching time (compared with that of SC_1 and SC_2). We observed that, in our evaluation, $SC = \langle Action, Resource\rangle$ is the best criterion, both in terms of performances and relatively low number of PDPs.

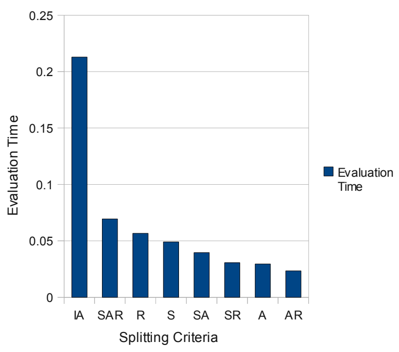
In order to answer RQ3, we measurer request processing time of XEngine compared with that of Sun PDP for policies splitted by our approach. We generated 10,000 random requests proposed in our previous work [13] and measured request processing time (in ms). Tables 2, 3, and 4 show our evaluation results. We observed that, in cases where our subjects equipped with XEngine instead of Sun PDP, our proposed approach improves performance (compared that of Sun PDP). This observation pleads in favour of applying our proposed refactoring process with XEngine as a decision instead of Sun PDP.

We next evaluated request processing time according to the number of requests incoming to the system. For each policy in the three systems (ASMS, LMS, and VMS), we generated 5000, 10000, .., 50000 random requests to measure the evaluation time (ms). The results are shown in (reference to tables). For the three systems, we notice that the evaluation time increases when the number of requests increases in the system. Moreover, the request evaluation time is considerably improved when using the splitting process compared to the initial architecture.

Figure 13 shows percentage of fetching time over overall evaluation time for request evaluation in LMS. Our results show that feting time is relatively small in comparison with a total evaluation time.



(a) LMS



(b) VMS

ASMS.pdf

(c) ASMS

Figure 10: Request Processing Time for our subjects LMS, VMS and ASMS

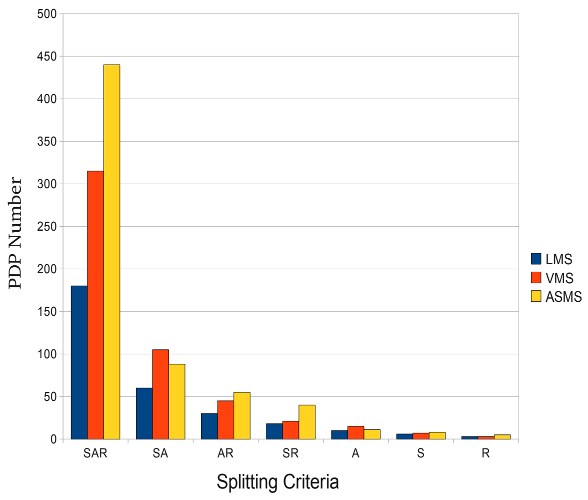


Figure 11: PDP Number According to Splitting Criteria

5. RELATED WORK

There are several previous work about performance issues in security mechanisms. Ammons et al. have presented techniques [4] to reduce the overhead engendered from implementing a security model in IBM's WebSphere Application Server (WAS). Their approach identifies bottlencks through code instrumentation and focuses on two aspects: the temporal redundancy (when security checks are made frequently) and the spatial redundancy (using same security techniques on same code execution paths). For the first aspect, they use caching mechanisms to store checks results, so that the decision is retrieved from the cache. For the second aspect they used a technique based on specialization, which consists in replacing an expensive check with a cheaper one for frequent codes paths. While this previous approach focus on bottlencks in program code, in this paper, we propose a new approach to refactor access control policies by reducing the number of rules in each split policy.

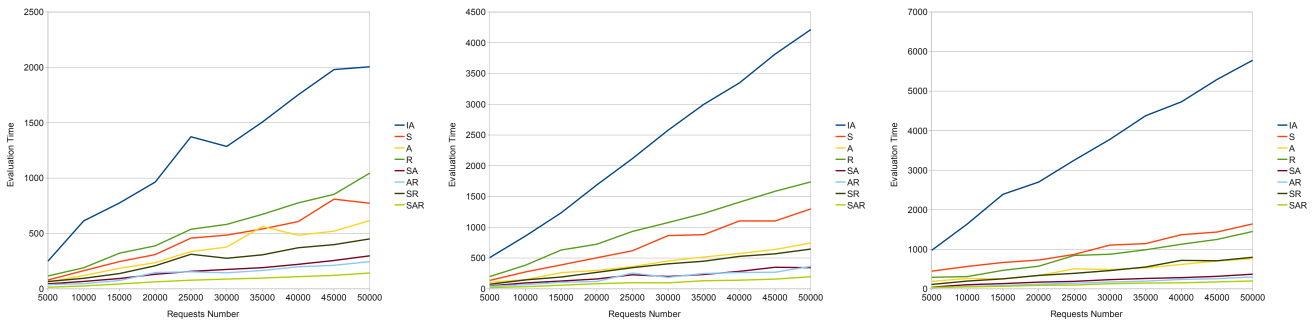
Various techniques have been proposed to addressed performance issues in systems interacting with access control policies [8, 10, 12]. Jahid et al. focus on XACML policy verification for database access control [8]. They presented a model, which converts attribute-based policies into access control lists. They implemented their approach, called MyABDAC. While they measure performance on MyABDAC in terms of request evaluation, however, they does not show how much MyABDAC gain improvement over an existing PDP.

Marouf et al. have proposed an approach [12] for policy evaluation based on a clustering algorithm that reorders rules and policies within the policy set so that the access to applicable policies is faster, their categorization is based on the subject target element. Their technique requires identifying the rules that are frequently used. Our approach follows a different strategy and does not require knowing which rules are used the most. In addition, the rule reordering is tightly related to specific systems. If the PDP is shared between several systems, their approach could not be applicable since the most "used" rules may vary between systems.

Lin et al. decomposed the global XACML policy into local policies related to collaborating parties, the local policies are sent to corresponding PDPs [10]. The request evaluation is based on local policies by considering the relationships among local policies. In their approach, the optimization is based on storing the effect of each rule and each local policy for a given request. Caching decisions results are then used to optimize evaluation time for an incoming request. However the authors have not provided experimental results to measure the efficiency of their approach when compared to the traditional architecture. While the previous approaches have focused on the PDP component to optimize the request evaluation,

Miseldine et al. addressed this problem by analyzing rule location on XACML policy and requests at design level so that the relevant rules for the request are accessed faster on evaluation time [14].

The current contribution brings new dimensions over our previous work on access control [11] [16] [15]. We have focused particularly in [11] on performance issues addressed with XACML policies



(a) LMS (b) VMS (c) ASMS

Figure 12: Processing Time for our subjects, LMS, VMS and ASMS depending on the requests Number

Figure 13: Percentage of Fetching Time

evaluation and we have proposed an alternative solution to brute force searching based on an XACML policy conversion to a tree structure to minimize the request evaluation time. Our previous approach involves a refactoring process that transforms the global policy into a decision diagram converted into forwarding tables. In the current contribution, we introduce a new refactoring process that involves splitting the policy into smaller sub-policies. Our two refactoring processes can be combined to dramatically decrease the evaluation time.

6. CONCLUSION AND FUTURE WORK

In this paper, we have tackled the performance issue in access control decision making mechanism and we have proposed an automated refactoring process that enables to reduce access control policies evaluation time up to 9 times. Our approach has been applied to XACML policies and it can be generalized to policies in other policy specification languages (such as EPAL). To support and automate the refactoring process, we have designed and implemented the "PolicySplitter" tool, which transforms a given policy into small ones, according to a chosen splitting criterion. The obtained results have shown a significant gain in evaluation time when using any of the splitting criteria. The best gain in performance is reached by the criterion that respects the synergy property. This plead in favor of a refactoring process that takes into account, the way PEPs are scattered inside the system business logic. In this work, we have easily identified the different PEPs since we know exactly how our system functions are implemented and thus how PEPs are organized inside the system.

As a future work, we propose to automatically identify the different PEPs of a given application. This technique is an important step that is complementary to this paper approach, since it enables knowing how PEPs are organized in the system and thus allows to select the most suitable splitting criterion for a given application.

7. REFERENCES

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

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

[3] PolicySplitter Tool. http://www.mouelhi.com/policysplitter.html, 2011.

[4] G. Ammons, J. deok Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *Proc. 18th European Conference on Object-Oriented Programming*, pages 170–194, 2004.

[5] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation. Mitre Corporation, 1976.

[6] J. Chaudhry, T. Palpanas, P. Andritsos, and A. Mana. Entity lifecycle management for OKKAM. In *Proc. 1st International Workshop on Tenerife*, 2008.

[7] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[8] S. Jahid, C. A. Gunter, I. Hoque, and H. Okhravi. MyABDAC: compiling xacml policies for attribute-based database access control. In *Proc. 1st Conference on Data and Application Security and Privacy*, pages 97–108, 2011.

[9] B. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, 1971.

[10] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo. Policy

decomposition for collaborative access control. In *Proc. 13th ACM Symposium on Access Control Models and Technologies*, pages 103–112, 2008.

[11] A. X. Liu, F. Chen, J. Hwang, and T. Xie. Xengine: A fast and scalable XACML policy evaluation engine. In *Proc. International Conference on Measurement and Modeling of Computer Systems*, pages 265–276, 2008.

[12] S. Marouf, M. Shehab, A. Squicciarini, and S. Sundareswaran. Statistics & clustering based framework for efficient xacml policy evaluation. In *Proc. 10th IEEE International Conference on Policies for Distributed Systems and Networks*, pages 118–125, 2009.

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

[14] P. L. Miseldine. Automated xacml policy reconfiguration for evaluation optimisation. In *Proc. 4th International Workshop on Software Engineering for Secure Systems*, pages 1–8, 2008.

[15] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Traon. A model-based framework for security policy specification, deployment and testing. In *Proc. 11th International Conference on Model Driven Engineering Languages and Systems*, pages 537–552, 2008.

[16] T. Mouelhi, Y. L. Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. In *Proc. 2009 International Conference on Software Testing Verification and Validation*, pages 171–180, 2009.

[17] Y. L. 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.

[18] R. Yavatkar, D. Pendarakis, and R. Guerin. A framework for policy-based admission control. RFC Editor, 2000.