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 manage authorization, access control architectures are designed to separate the business logic from an access control policy. An access control policy consists of rules which specifies who can access to resources. A request is formulated from a component, called Policy Enforcement Points (PEPs). Given a request, a Policy Decision Point (PDP) evaluates 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 to refactor access control policies for performance improvement by splitting an access control policy (handled by single PDP) into its corresponding multiple access control policies with smaller number of rules (handled by multiple PDPs). We define seven attribute-set based splitting criteria to facilitate split an access control policy. We have conducted an evaluation on three subjects of real-life JAVA program, each of which interact with access control policies. Our evaluation results show that our proposed approach preserves the initial architectural model of the subjects in terms of interaction between the business logic and its corresponding rules in the policy. Our evaluation results show that 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 system resources based on access control policies. Access control policies (i.e., policies in this paper) are based on various access control models such as Role-Based Access Control (RBAC) [7], mandatory access control (MAC) [5], discretionary access control (DAC) [10], and Organisation-based access control (OrBAC) [9]. In this paper, we consider access control policies specified in the eXtensible Access Control Markup Language (XACML) [2]. XACML is a popularly used XML-based language to specify rules in a policy. A rule specifies actions (e.g., access) that subjects (e.g., student) can take on resources (e.g., grades) if required conditions are met.

In the context of policy-based systems, access control architectures are often built with respect to a popular architectural concept, which separates Policy Enforcement Points (PEP) from Policy Decision Point (PDP) [19]. More specifically, the PEP is located inside an application code (i.e., business logic of the system). Given requests (e.g., can student A access 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. An important benefit of this 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 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 pitfall for degrading performance as our previous work showed that the large number of rules is critical for efficient request evaluation [12]. This performance bottleneck issue may impact service availability as well, especially in face of explosive number of requests.

In order to address this performance bottleneck issue, we propose an approach to refactor 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 improves performance. Our approach includes two facets: (1) refactor an access control policy (handled by single PDP) into its corresponding multiple access control policies with smaller number of rules (handled by multiple PDPs), and (2) refactor PEPs with regards to refactored PDPs while preserving architectural property that a single PDP is triggered by a given PEP at a time.

In the first facet, 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 smaller number of rules. This refactoring involves grouping rules in the global policy into several subsets based on splitting criteria. More specifically, we propose a set of splitting criteria to refactor the global policy into smaller policies. A splitting criterion selects and groups the access 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 facet, our approach aims at preserving architectural property where a 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 behaviors of initial architectures in policy-based systems. More specifically, each PEP should be mapped to a PDP with a set of relevant rules for given requests issued by the PEP. Therefore, our refactoring maintains behaviors of initial architectures in policy-based systems.

We collect three subjects of real-life JAVA program, each of which interact with access control policies. 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; First one is Sun PDP implementation [1], which is a popular open source PDP and Second one is XEngine [12], which transforms an original XACML policy into its corresponding policy in a tree format by mapping attribute values with numerical values. Our evaluation results show that our approach preserves the initial architectural model of the subjects in terms of interaction between the business logic and its corresponding rules in the policy. 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 with 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 does not alter behaviors of initial access control 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 initial access control architectures in terms of request evaluation time.

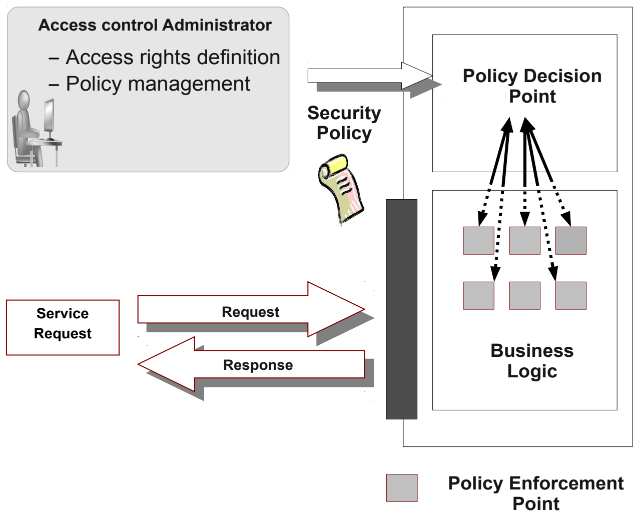


Figure 1: Access Control Request Processing

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 presents (1) requirements that we aim to preserve while refactoring access control policies, and (2) motivation for our proposed approach presented in this paper.

2.1 Synergy Requirement for Access Control Architecture

Managing access control policies is an one of challenging tasks faced by organizations. For example, in policy-based systems, policy authors handle specific requirements such as role swap for given temporary assignments, and changes in procedures, system resources for protection, users' roles, roles' access rights in an organization. Moreover, these changes in access control policies should comply with existing business logic in the systems.

To address these issues, there is a great needs for a simple access control architecture that help handle changes in access control policies easily. Recently, a popular access control architecture is to handle an access control policy as a separate component encapsulated in a PDP. Figure 1 illustrates interactions between the PEPs and the PDP: the PEP calls the PDP to retrieve an authorization decision based on the encapsulated policy. The PDP evaluates a request against rules in the policy and return an authorization decision to the PEP. The separation between the PEP and the PDP in access control systems simplifies policy management across many heterogeneous systems and enables to avoid potential risks arising from incorrect policy implementation, especially when the policy is hardcoded inside the business logic.

Along with the reasoning about performance, we maintain the simplicity of the access control architecture based on model presented in Figure 2. In this model, a set of business processes, which comply to users' needs, are encapsulated in a given business logic. The business logic is enforced by multiple PEPs. Conceptually, evaluating request to return an authorization is a decision making process where each PEP interacts with one single PDP. Given an access control request (issued by a given PEP), the PDP evaluates the request against *only* a single XACML policy and returns its suitable response.

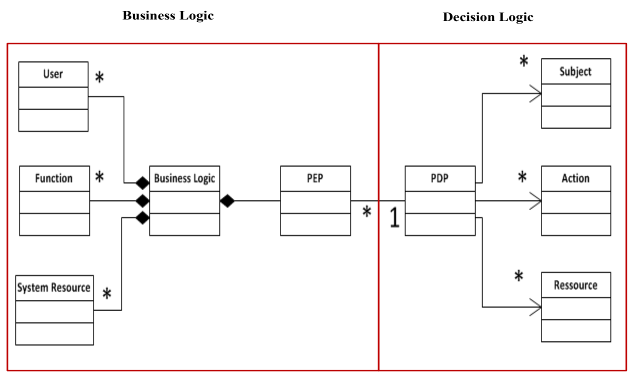


Figure 2: Access Control Model

Given a single policy, administrators are easy to manage a policy and maintain a simple architecture where a given PEP is mapped to a fixed single PDP at each decision making process. Consider that we have multiple policies, each of which is loaded by different PDPs. In this paper, given multiple PDPs, we define synergy requirement in the access control architecture by mapping a PEPs with a PDP loaded with a suitable access control policy dynamically before deployment. The goal behind maintaining synergy requirement helps keep a strong traceability between a policy and the internal security mechanisms, which enforces a policy at the business logic level. In such a setting, when administrators update or remove an access control policy, its corresponding PEPs can be easily updated synchronously with the policy changes based on traceability.

2.2 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 standard policy specification language that defines a syntax of access control policies and request/response. XACML enables administrators 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.

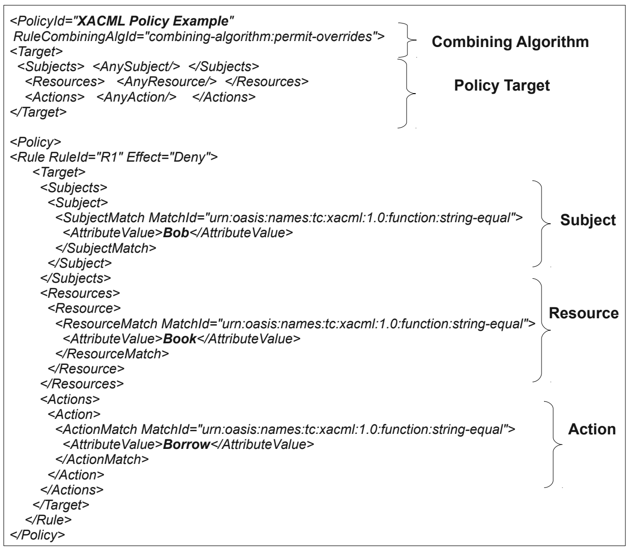
An XACML is constructed as follows. A `policy set` element consists of a sequence of `policy elements`, a combining algorithm, and a `target` element. A `policy element` is expressed through a target, a set of `rules`, and a rule combining algorithm. A `target` element consists of the set of resources,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

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

When more than one rule is applicable to a request, the combining algorithm helps determine which rule and decision can be finally applied to the request. For example, given two rules, which 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 engenders 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 in Lines 6-14.


```
<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 become 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, administrators 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 raise 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 factors, which may lead to degrade XACML requests evaluation performance:

- An XACML policy may contain various attribute elements including `target` elements. Retrieval of attributes 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 final authorization decision (i.e., effect) of the whole `policy set` after combining all of applicable rules' decisions according to the request. Computation of combining applicable rules' decisions may contribute for a potential evaluation time latency.
- `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. 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 requirement in the access control architecture.

3.1 Definition of Policy based Splitting Criteria

Given a request, a PDP adopts brute force searching to find its decision by evaluating the request against policy rules one by one until the PDP finds an evaluation decision. For request evaluation processing, not of all the rules are applicable to the request. In order words, only part of the rules (i.e., relevant rules) are applicable to the request and can contribute to determine a 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 refactors a policy into a set of sub-policies where the rules have the same values of attributes of subject, resource, or action. Formally, our approach refactors an policy P into multiple policies P_{SC_w} based on Splitting Criteria SC_w. These multiple policies contain less number of rules (compared to the number of rules in P) and conforms to a Splitting Criteria SC_w. An SC_w defines the set of attributes, which classify all of policy rules into subsets with the same attribute values for selected attributes. w denotes the number of attributes that have to be considered conjointly for aggregating rules based on selected attributes.

We first consider two attributes (e.g., $\langle Subject, Action\rangle$ or $\langle Action, Resource\rangle$) for Splitting Criteria. In such a setting, our approach refactors a policy into a set of policies each of which rules include the same couple of attribute elements. For example, $\langle Subject, Action\rangle$ criterion classifies rules with same subjects and actions into sub-policies. We also extend our Splitting Criteria by considering three attributes such as resource, action, and subject. 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

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 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. Once the applicable policy. The PDP then retrieve the applicable rules that are applicable to the request.

Figure 4 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.

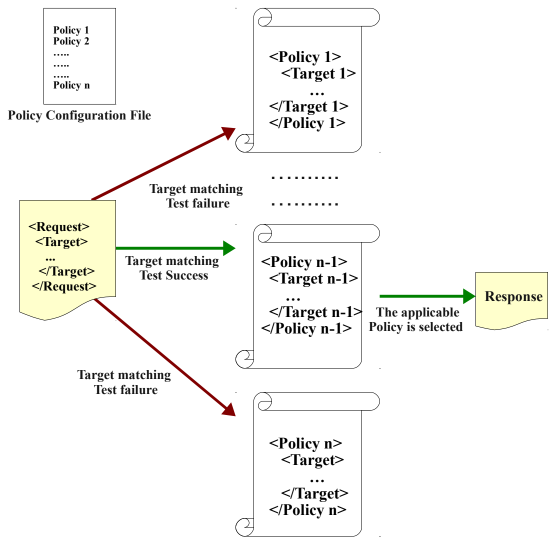


Figure 4: Applicable policy Selection

Figure 5 shows an overview of our approach. In our approach, the policy splitter component plays a role to refactor access control policies. Given a single PDP loaded with initial global policy, the policy splitter component 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 changed, the policy splitter component is required to conduct refactoring of 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.

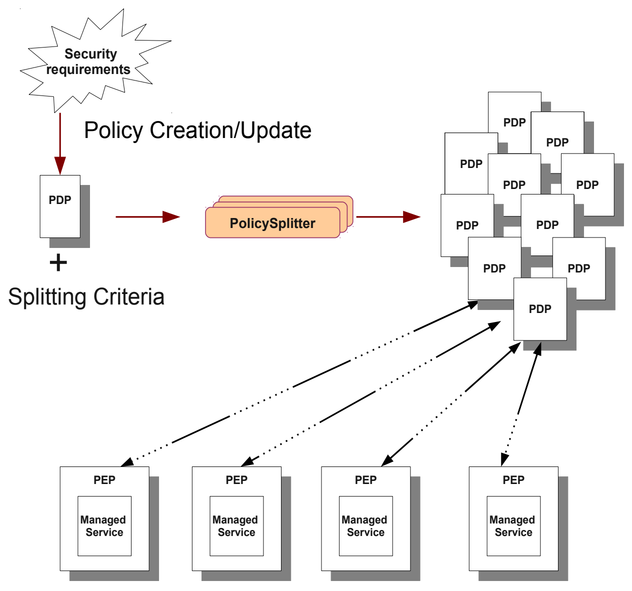


Figure 5: Overview of the Refactoring Process

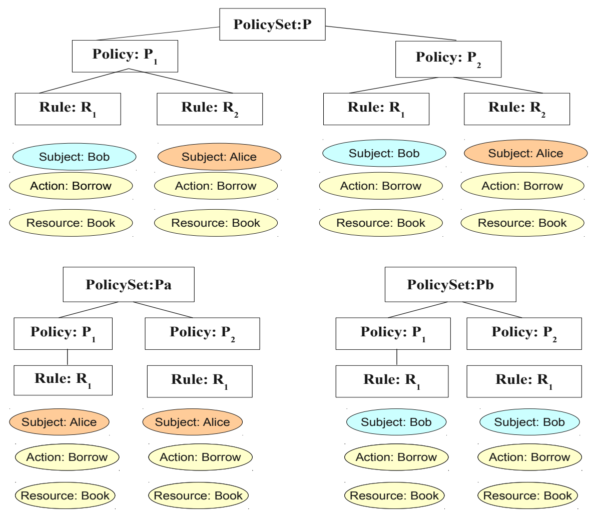


Figure 6: Refactoring a policy according to $SC_1 = \langle Subject \rangle$

```
<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 7: Multi-attributes values in `target` element

To illustrate our approach, we present examples that take into consideration the XACML language features:

- In Figure 6, our approach refactors an XACML policy P according to the splitting criterion $SC_1 = \langle Subject \rangle$. Our refactoring results in two sub-policies Pa and Pb. Each sub-policy consists of relevant rules with regards to the same subject (Alice or Bob in this case).
- 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 7, 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.

3.2 Architecture Model Preservation: PEP-PDP Synergy

We propose to preserve the synergy requirement in the access control architecture by mapping a PEP and a PDP loaded with a relevant policy for a request dynamically at run-time. As shown in Section 3.1, our proposed splitting criteria helps refactoring in access control policies. Given multiple PDPs after the policy splitting refactoring, we consider (1) how PEPs are organized at the application level, and (2) how PEPs are linked to their corresponding PDPs.

In this paper, our goal is to improve performance by reducing request evaluation time. While our approach refactors an initial single PDP into multiple PDPs, this refactoring may not lead to improve performance: a PEP may send its requests to several PDPs. A PDP, which receives a request is only known at runtime. Such a architecture may not reduce evaluation time without PEP-PDP synergy and preserve 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 between PEPs to PDPs. Given a request, our approach maps a PEP to a PDP with relevant rules for the request. Threfore, a request issued from a PEP should be handled by the same PDP. Operationally, the request evaluation process involves one XACML policy. In this case, our refactoring does not impact the conceptual architecture of the system.

At the application level, the PEP is represented by a method call that triggers a decision making process. Figure 8 presents sample PEP code from [18]. This code snippets shows an example of a PEP represented by the method `checkSecurity`, which calls the class

`SecurityPolicyService`, which formulates a request initiates the PDP component.

```
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 8: PEP deployment Example

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 using $SC_2 = \langle Resource, Action\rangle$. Give a request issued from the PEP, our approach fetches a PDP loaded with a policy containing rules according those action and resource attributes. 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 enforcements points in an application.

4. EMPIRICAL RESULTS

To measure the efficiency of our approach, we conducted two empirical studies. The first takes into consideration the whole system (PEPs and PDPs) to evaluate the performance improvement regarding the decision making process. The request processing time, for each splitting criterion is compared to the processing time of the initial architecture implementing the global policy (the evaluation that considers the global policy, is denoted (IA), the "Initial Architecture"). The second empirical study focuses only on the PDPs in isolation to measure the gain in performance independently from the system. To make such study of PDPs in isolation, we use XEngine [12]. The objective of the second study is to see how our approach can be combined with XEngine and how this impacts the performance. The first subsection introduces our empirical studies and presents the tool that supports our approach. The remaining two sections present and discuss the results of the two empirical studies.

4.1 Empirical Studies and PolicySplitter Tool

The empirical studies were conducted using the following systems [17]:

- LMS: The library management system offers services to manage books in a public library.
- VMS: The virtual meeting system offers simplified web conference services. The virtual meeting server allows the organization of work meetings on a distributed platform.
- ASMS (Auction Sale Management System): allows users to buy or sell items online. A seller can start an auction by submitting a description of the item he wants to sell and a minimum price (with a start date and an ending date for the auction). Then usual bidding process can apply and people can bid on this auction. One of the specificities of this system is that a buyer must have enough money in his account before bidding.

We started by a processing step, in which we have augmented the rules number in the three original policies for these studies, as it would be difficult to observe a performance improvement with systems including few rules. After adding extra rules to the three systems, LMS policy contains 720 rules, VMS has 945 rules while ASMS implements 1760 rules. The rules that we have added do not modify the system behavior as they are conform to the specifications. Our evaluations were carried out on a desktop PC, running Ubuntu 10.04 with Core i5, 2530 Mhz processor, and 4 GB of RAM. We have implemented the PolicySplitter tool in Java. PolicySplitter enables splitting the policies according to the chosen splitting criteria. The tool is available for download from [3]. The execution time of the tool is not considered as a performance factor as it takes up to few seconds (for very large policies) to perform the splitting according to all SCs. Moreover the refactoring process is executed only once to create a configuration that supports a selected splitting criterion.

4.2 Performance Improvement Results

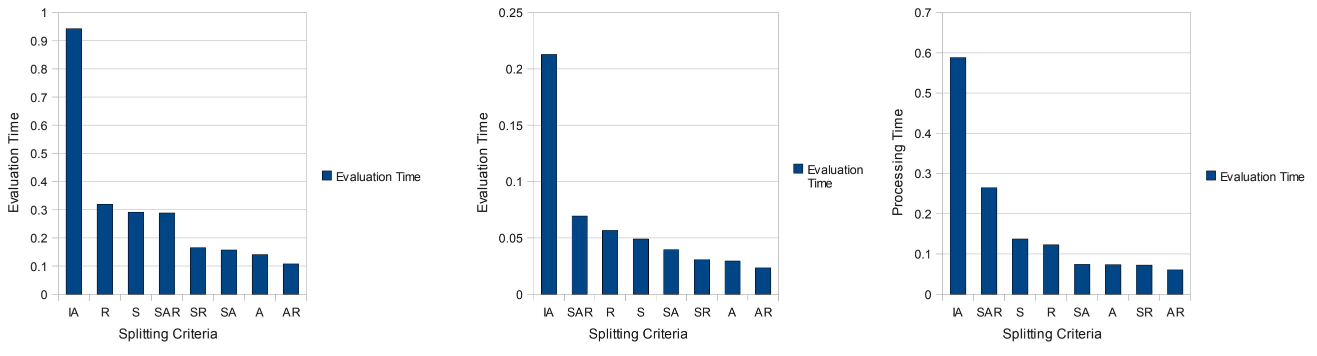
For the 3 evaluation studies, we generated the resulting sub-policies for all the splitting criteria that we have defined in Section III. The decision Engine in our three case studies is based on Sun XACML PDP implementation [2]. We choose to use Sun XACML PDP instead of XEngine in order to prove the effectiveness of our approach when compared to the traditional architecture. For each splitting criteria, we have conducted systems tests to generate requests that trigger all the PEPs in the three evaluation studies. The test generation step leads to the execution of all combination of possible requests, all the details related to this step are presented in details in our previous [17].

The process of tests generation is repeated ten times in order to avoid the impact of randomness. We applied this process to each splitting criterion and calculated the average execution time.

The results are shown in Figure 9. They show the execution time considering the sub-policies resulting from each splitting criterion and the global policy that corresponds to the initial architecture (IA). Note that the results are ranked from the largest processing duration to the smallest one. From the Figure 9, we can make two observations:

- Compared to the initial architecture (IA), the evaluation time is considerably reduced for all the splitting criteria. This is consistent with our initial intuition. In fact, splitting the policy into small policies improves requests processing duration.
- The splitting criteria $SC = \langle Action, Resource\rangle$ enables to have the best evaluation time. In fact, the PEPs in the 3 empirical studies are scattered across the applications by a categorization that is based on $\langle Action, Resource\rangle$. This pleads in favor of adopting a splitting criteria that takes into account the PEP-PDP synergy requirement.

In a second step, we have evaluated PDPs number generated by each splitting criterion, to study the impact of the refactoring process on the initial policy. For the 3 XACML studies, we executed



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

Figure 9: Processing Time for our 3 systems LMS, VMS and ASMS

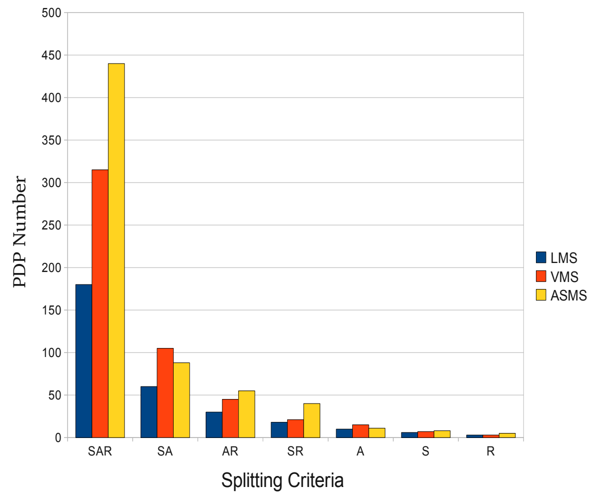


Figure 10: PDP Number According to Splitting Criteria

the PoliySplitter tool on the 3 initial policies and we generated the number of resulting policies, in each study. As highlighted by Figure 10, we notice that there are three categories of results: SC_1 category leads to a small number of PDPs, SC_2 category produces a reasonable number of PDPs whereas SC_3 leads to a huge number of PDPs. SC_2 category, thus appears as a good tradeoff in terms of performance and number of PDPs generated: In our evaluation studies, $SC = \langle Action, Resource \rangle$ is the best criterion, both in terms of performances and low number of PDPs.

4.2.1 Performance improvement with XEngine

The goal of this empirical study is to show the impact of combining XEngine as a decision engine rather than Sun XACML PDP implementation with our approach. We have chosen to use [12], mainly for 3 reasons:

- It uses a refactoring process that transforms the hierarchical structure of the XACML policy to a flat structure.
- It converts multiple combining algorithms to single one.
- It lies on a tree structure that minimizes the request processing time.

We propose to use XEngine conjointly with the refactoring process presented in this work: We have evaluated our approach in 2 settings:

- Considering evaluations with a decision engine, based on SUN PDP, with split policies and with the initial policy.
- Considering evaluations with a decision engine, based on XEngine rather than Sun PDP, with split policies and with the initial policy as well.

We have measured the processing time (in ms) of a randomly generated set of 10,000 requests. For request generation, we have used the technique presented in [14]. The request time processing is evaluated for LMS, VMS, ASMS. The results are presented in Table I, II and III.

	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

From the different tables, we can observe that, when used with a decision engine based on XEngine rather than Sun PDP, our proposed approach provides more performance improvement. This empirical observation pleads in favour of applying our proposed refactoring process with XEngine.

In a second step, we have considered the workload of the software system. We evaluated the request procesing time according to the number of requests incoming to the system. For each policy in the three systems (ASMS, LMS, and VMS), we generate successively 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.

We have also calculated the pourcentage of the fetching time by considering the global time of request evaluation in LMS system. The results are shown in Table 5. This time varies between 0 and 28.6%.

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 [8,11,13]. 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 [13] 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 [11]. 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 [15].

The current contribution brings new dimensions over our previous work on access control [12] [17] [16]. We have focused particularly in [12] on performance issues addressed with XACML policies 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. We have proposed an automated refactoring process that enables to reduce request evaluation time. Our approach has been applied on three JAVA projects interacting with XACML policies. To support and automate the refactoring process, we have designed and implemented a tool, called PolicySplitter, which transforms a given policy into multiple policies according to a chosen splitting criterion.

Our evaluation results have shown that our approach gain a significant performance improvement in terms of request evaluation time. Our evaluation results have shown that our proposed approach preserves the initial architectural model of the subjects in terms of interaction between the business logic and its corresponding rules in the policy. Our evaluation results have shown that our approach is efficient in terms of reducing request evaluation time by up to nine times.

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 *In Proceedings of ECOOP*. Springer, 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 IRSW2008 International Workshop on Tenerife*, 2008.

[7] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and

Requests Number (ASMS)	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
IA	972	1639	2390	2699	3254	3781	4377	4727	5293	5779
S	446	566	666	727	870	1107	1143	1368	1436	1641
A	189	262	255	337	507	487	531	616	705	762
R	292	310	467	572	845	874	986	1127	1249	1449
SA	42	104	133	169	188	232	262	283	315	370
AR	28	60	84	117	142	172	195	236	256	300
SR	111	196	252	339	391	461	552	720	710	798
SAR	38	49	63	89	95	130	145	152	175	199

Table 5: Processing Time for ASMS using XEngine

Requests Number (LMS)	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
IA	250	613	775	964	1373	1286	1507	1752	1980	2005
S	80	164	246	309	458	485	541	607	810	773
A	61	120	185	236	336	377	563	484	521	616
R	116	190	322	388	538	582	673	776	853	1045
SA	45	67	93	131	158	175	191	221	256	298
AR	37	47	75	147	154	145	166	199	212	246
SR	66	95	138	208	312	276	307	371	399	451
SAR	14	26	44	63	78	88	97	110	121	143

Table 6: Processing Time for LMS using XEngine

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 *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 97–108, 2011.

[9] A. A. E. Kalam, S. Benferhat, A. Miège, R. E. Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin. Organization based access contro. In *POLICY*, 2003.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

Requests Number (VMS)	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
IA	508	858	1235	1688	2117	2579	2998	3348	3815	4214
S	137	274	388	504	619	865	880	1104	1103	1298
A	105	149	262	299	361	452	517	575	640	745
R	199	384	631	725	933	1077	1226	1409	1585	1738
SA	47	96	127	160	228	201	233	284	350	344
AR	42	67	107	120	256	186	249	265	270	365
SR	74	145	192	267	344	406	449	528	569	646
SAR	21	34	59	83	100	97	131	140	158	196

Table 7: Processing Time for VMS using XEngine

Requests Number (LMS)	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
S	1.31	0.64	0.91	1.01	1.09	1.14	1.16	1.03	0.07	1.08
A	1.66	1.70	1.71	1.70	1.36	1.42	1.47	1.50	1.55	1.56
R	1.04	1.06	1.06	0.80	0.86	0.91	0.93	0.82	0.96	0.87
SA	7.40	7.69	6.32	6.60	6.10	6.32	6.48	6.16	6.38	6.31
AR	8.33	6.12	6.94	7.29	7.56	7.09	6.74	6.80	6.94	6.66
SR	2.27	3.40	3.81	3.40	3.68	3.83	3.61	3.73	3.79	3.68
SAR	14.28	19.23	21.05	19.23	17.64	18.98	18.08	18.69	18.33	18.97

Table 8: Percentage of Fetching Time