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

Modern access control architectures tend to separate the business logic from access control policy specification for the sake of easing authorization manageability. Thus, request evaluation is processed by a Policy Decision Point (PDP) that encapsulates the access control policy and interacts with the business logic through Policy Enforcement Points (PEPs). Such architectures may engender performance bottleneck due to the number of rules that have to be evaluated by a single PDP at decision making time. In this paper, we propose to optimize the decision-making process by splitting the PDP into smaller decision points. We conducted studies on XACML (eXtensible Access Control Markup Language) to identify the best PDP splitting configuration. Our evaluation results show that the best splitting criterion is the one that preserves the initial architectural model in terms of interaction between the business logic and the decision engine and enables to reduce the time of request evaluation time by up to 9 times.

Keywords

Performance, Optimization, Access Control Policies, PEP, PDP, XACML

1. INTRODUCTION

Access control policies are derived from an access control model like RBAC, MAC, DAC and OrBAC [10, 5, 7, 9] to define authorization strategies which regulate the actions that users could perform on system resources. In the context of policy based software 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 the application (i.e., business logic of the system) and formulates requests. The PDP encapsulates the policy. The PDP then receives requests from the PEP and returns evaluation responses (e.g., permit or deny) by evaluating these requests against rules in the policy. This architecture enables to facilitate managing access rights in a fine-grained way since it decouples the business logic from the access control decision logic, which can be standardized.

In this architecture, the policy is centralized in one point, the PDP. Centralization of the PDP offers advantages such as facilitating managing and maintaining the policy, which is encapsulated into a single PDP. However, such centralization can be a pitfall for degrading performance since the PDP manages all of access rights to be evaluated by requests issued by PEPs. Moreover, the dynamic behaviour of organizations, the complexity of its workflow and the continuous growth of its assets may increase the number of rules in the policy [6]. This fact may dramatically degrade the decision-making time, lead to performance bottlenecks and impact service availability.

In this paper, we tackle perfomance requirements in policy based software systems and propose a mechanism for refactoring access control policies to reduce the request evaluation time. Reasoning about performance requirements in access control systems is intended to enhance the reliability and the efficency of the overall system and to prevent business losses that can be engendered from slow decision making. In our approach, we consider policy-based software systems, which enforce 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 the actions that subjects can take on resources if required conditions are met.

We propose an automated approach for optimizing the process of decision making in XACML request evaluation. Our approach includes two facets: (1) performance optimization criteria to split the PDP into smaller decision points, (2)

architectural property preservation to keep the initial architectural model in which only a single PDP is triggered by a given PEP.
The performance optimization facet of the approach relies in transforming the single PDP into smaller PDPs, through an automated refactoring of a global access control policy (i.e., a policy governing all of access rights in the system). 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). Our approach takes as input a splitting criterion and an original global policy, and returns a set of corresponding sub-policies. Given a set of requests, we then compare evaluation time of the requests against the original policy and a set of sub-polices based on the proposed splitting criteria.

The second facet aims at keeping the cardinality based feature between the PEP and the PDP by selecting the splitting criteria that do not alter the initial access control architecture. The best splitting criteria must thus both improve performance and comply to our architecture: the relationship must be maintained, linking each PEP to a given PDP. When refactoring the system, each PEP in our system can be mapped to a set of rules that will be relevant when evaluating requests.

We automate this refactoring process and conduct an evaluation to show that our approach enables the decision-making process to be 9 times faster than using only a single global access control policy. In our previous work [12], we addressed performance issues in policy evaluation by using a decision engine called XEngine that outperforms Sun PDP implementation [1]. XEngine transforms an original XACML policy into its corresponding policy in a tree format by mapping attribute values with numerical values. Our contribution in this paper goes in the same research direction as it aims to reduce the request evaluation time by refactoring the policy. Our evaluation results show that decision making time is reduced up to nine times with split policies if compared to its original global policy and that we have a considerable performance improvement, if the policies resulting from our refactoring process are evaluated with XEngine rather than Sun PDP.

This paper makes the following three main contributions:

- We develop an approach that refactors a single global policy into smaller policies to improve performance in terms of policy evaluation.
- We propose a set of splitting criteria to help refactor a policy in a systematic way. The best splitting criterion is the one that does not alter the initial access control architecture.
- We conduct evaluations on three real world Java programs interacting with XACML policies to measure performance improvement using our approach. We compare our approach with an approach based on the global policy in terms of efficiency. Our approach achieves more than 9 times faster than that of the approach based on the global policy in terms of evaluation time.

The remainder of this paper is organized as follows: Section 2 introduces some concepts related to the topic and the research problem. Section 3 presents the overall approach. Section 4 presents evaluation results and discusses the effectiveness of our approach. Section 6 discusses related work. Section 7 concludes this paper and discuses future research directions.

2. CONTEXT/PROBLEM STATEMENT

The present section introduces the synergy requirement that we aim to preserve while reasoning about performance requirements for access control policies and clarifies the motivation for the work presented in this paper.

2.1 Synergy requirement related to the considered Access Control Architecture

Managing access control policies is one of the most challenging issues faced by organizations. Frequent changes in access control systems may be required to meet business needs. An access control system has to handle some specific requirements like role swap when employees are given temporary assignments, changes in the policies and procedures, new assets, users and job positions in the organization. All these facts make access control architectures very difficult to manage, and plead in favor of a simple access control architecture that can easily handle changes in access control systems.

Today's access control scenarios involve an access control policy which is modeled, analyzed and implemented as a separate component encapsulated in a PDP. Figure 1 illustrates the interactions between the PEPs and the PDP: 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. 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, when the policy is hardcoded inside the business logic.

Along with the reasoning about performance, we propose to maintain the simplicity of the access control architecture whose model is presented in Figure 2. 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, thus a single XACML policy is evaluated to provide the suitable response for an access control request provided by a given PEP.

Considering a single XACML policy file for each initiating PEP enables to ease policies management and to maintain a simple architecture in which a given PEP is mapped to a

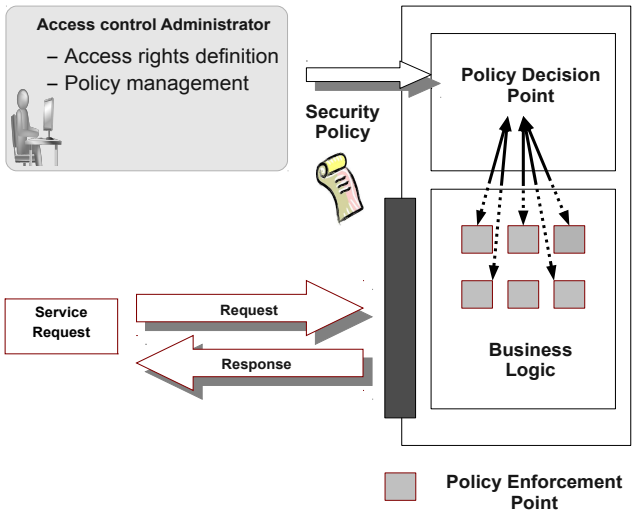


Figure 1: Access Control Request Processing

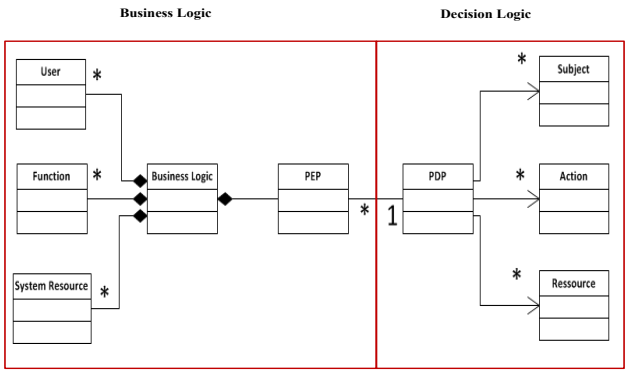


Figure 2: The Access Control Model

fixed PDP at each decision making process. In this work, We define the synergy requirement in the access control architecture by the specialization of an access control policy to be used for a particular PEP before deployment. The goal behind maintaining a single policy related to each PEP in the system is to keep a strong traceability between what has been specified by the policy 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 and thus the application is updated synchronously with the policy changes.

2.2 XACML Access Control Policies and Performance Issues

In this paper, we focus on access control decision making for policies expressed in XACML. XACML [2] is a standard policy specification language that defines a syntax of access control policies in XML. In the XACML paradigm, a policy set is a sequence of policies, a combining algorithm and a target. A policy element is expressed through a target, a set of rules and a rule combining algorithm. The target defines the set of resources, subjects and actions to which a rule is applicable. The rule element consists of a target, a condition, and an effect. The condition is a boolean expression that specifies restrictions (e.g., time and location restrictions) on the elements in the target. It represents the environmental context in which the rule applies. Finally, the effect element is either permit or deny. When more than one rule is applicable to a given request, the combining algorithm enables selecting which rule to choose. 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 decision. 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 simple XACML policy that denies subject Bob to borrow a book.

```
<PolicyId="XACML Policy Example"                                  } Combining Algorithm
 RuleCombiningAlgId="combining-algorithm:permit-overrides">       }
<Target>                                                          } Policy 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">   } Subject
            <AttributeValue>Bob</AttributeValue>                                        }
          </SubjectMatch>                                                               }
        </Subject>                                                                      }
      </Subjects>                                                                       }
      <Resources>                                                                       } Resource
        <Resource>                                                                      }
          <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">  }
            <AttributeValue>Book</AttributeValue>                                       }
          </ResourceMatch>                                                              }
        </Resource>                                                                     }
      </Resources>                                                                      }
      <Actions>                                                                         }
        <Action>                                                                        } Action
          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">    }
            <AttributeValue>Borrow</AttributeValue>                                     }
          </ActionMatch>                                                                }
        </Action>                                                                       }
      </Actions>                                                                        }
    </Target>
  </Rule>
</Policy>
```

Figure 3: XACML Policy Example

The XACML request encapsulates attributes that define which subject is requested to take an action on which protected resource in a system. This can be under/not a condition. A request, which satisfies both target and condition in a rule, is evaluated to decision specified in the rule's effect element. If the request does not satisfy target and condition

elements in any rule, its response yields "NotApplicable" decision.
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 the access control policy to support new requirements.

However, the increasing complexity of organizations in terms of structure, relationships, activities, and access control requirements has impacted access control policy design. This has led to deal with access control policies where the number of rules depicting the associations between resources, users and actions is in continuous increase. Moreover, the policy is usually centralized in a single point PDP, which has to manage multiple access control requests. These observations 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. Moreover, considering XACML specifities, some factors may lead to degrade XACML requests evaluation performance:

- An XACML policy may contain several attributes that constitute descriptive information about the target elements, the retrieval of varying size of attributes values for each request within XML elements in a request, may increase the evaluation time.
- A PolicySet includes a set of policies, the effect of the whole PolicySet is determined by combining the effects of the policies according to a combining algorithm, the computation of resulting effect from policies is also considered as a potential evaluation time latency.
- Conditions evaluation in XACML rules may slow down the decision making process since these conditions can be complex because they can be built from an arbitrary nesting of non-boolean functions and attributes.

3. XACML POLICY REFACTORING PROCESS

This section describes our approach and gives an insight of the process supporting it. In the first step, we show how performance improvement can be achieved by reducing the number of access control rules that have to be considered at the evaluation time, through the definition of policy based splitting criteria. Secondly, we explain how to select the splitting criterion that preserves the synergy requirement in the access control architecture.

3.1 Definition of Policy based Splitting Criteria

Decision engines performing XACML policy evaluation use brute force searching techniques by comparing a given request with all the rules in a policy. What happens in a request evaluation process is that for a given access control request, some rules are evaluated in the global policy whereas some of those rules are not applicable to the request. Starting from this observation, we propose to evaluate only the relevant rules for a given request in the decision making process. We propose to split the initial policy into smaller policies based on attribute values combination. For a given 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. A policy can be split into a set of sub-policies where the rules have the same values of attributes of subject, resource, or action. We also consider two attributes like $\langle Subject, Action\rangle$ or $\langle Action, Resource\rangle$ for splitting the policies, in this setting, a policy is transformed into a set of smaller policies where rules in the resulting policies have the same couple of attribute elements. We can go further in our grouping strategy by considering a triplet of attributes elements: resource, action and subject to split P. Table I shows all 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	$\langle Subject, Action\rangle, < Subject, Resource\rangle$ $\langle Resource, Action\rangle$
SC_3	$\langle Subject, Resource, Action\rangle$

Table 1: Splitting Criteria

Once the splitting process is performed, the access control architecture will include one or more (PDPs) that comply with a certain splitting criterion. We use SUN's XACML implementation [2] as a decision engine framework that evaluates the policies against requests. During the evaluation process, SUN's XACML decision engine checks the request against the policy and determines whether the request is permitted or denied. SUN's XACML implementation permits to load the considered policy in a file which is used during the decision making process, once the applicable policy is selected from this file, the decision engine fetches the policy to retrieve the applicable rules that are applicable to the request.

Figure 4 presents an evaluation process that considers resulting policies from the splitting process. During the evaluation process the applicable policy is selected from the configuration file. Sun's XACML enables to select the applicable policy, by verifying the matching between the request's attributes and the policy set attributes. After the selection of the applicable policy, all its rules which are considered relevant for the decision making are evaluated.

As depicted in Figure 5, the refactoring process is automated and starts by specifying and creating the XACML file which will be split by our tool according to a specified SC that can be chosen by an access control stakeholder. Afterwards, the policies are included in the framework that supports our approach. For every change in the access control policy, the initial policy is updated and split again in order to be

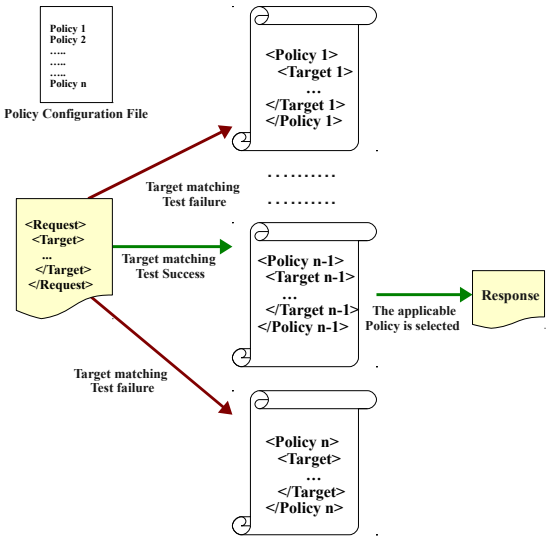


Figure 4: Applicable policy Selection

included again in the framework. From a point of view of the system administration, maintaining and updating the access control policies is completely a standalone and simple process. The input is usually a centralized XACML policy. This input remains the same than before the access control performance issue is tackled. Our process is transparent in the sense that it does not impact the existing functional aspects of the access control management system, for system administrators, who have to update the policy frequently and have to manage various dimensions of access control systems such the scalability and maintainability.

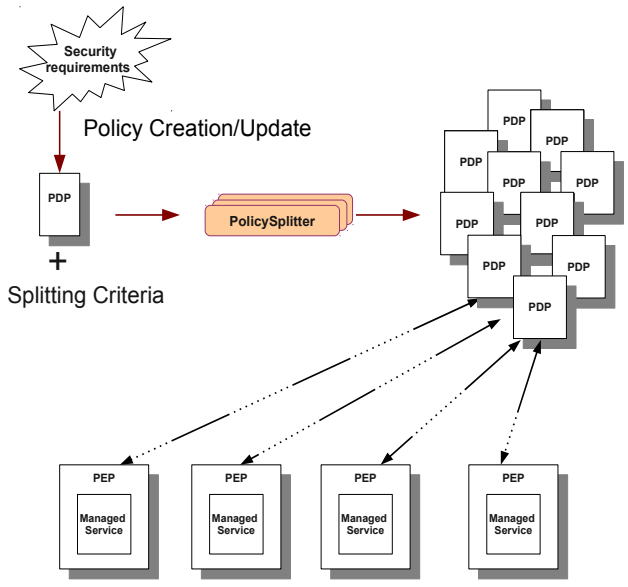


Figure 5: Overview of the Refactoring Process

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

- The refactoring of the XACML policy P presented in Figure 6 according to the splitting criterion $SC_1 = \langle Subject \rangle$ consists in splitting P into sub-policies Pa and Pb where each sub-policy groups the rules that are relevant to the same subject (Alice and Bob in this case).
- XACML target elements may have an intricate structure, if they encapsulate the following elements $\langle ResourceMatch \rangle$, $\langle SubjectMatch \rangle$, $\langle ActionMatch \rangle$. These elements define a set of target elements related entities that match at the decision level with the context element in an access control request. In the example, shown in Figure7, subject attribute includes two attributes (one is "role" and the other is "isEq-subjUserId-resUserId"). This subject can match with a request if the request has that at least role is pc-member and isEq-subjUserId-resUserId is true. In such configuration, the whole element subject is considered as a single entity that is not splitted by our supporting tool.
- In XACML, policies and requests could be multi-valued. For example, the subject in a given request could both be the manager and the employee as a principal. For the sake of simplicity, we don't consider multi-valued requests in this work.

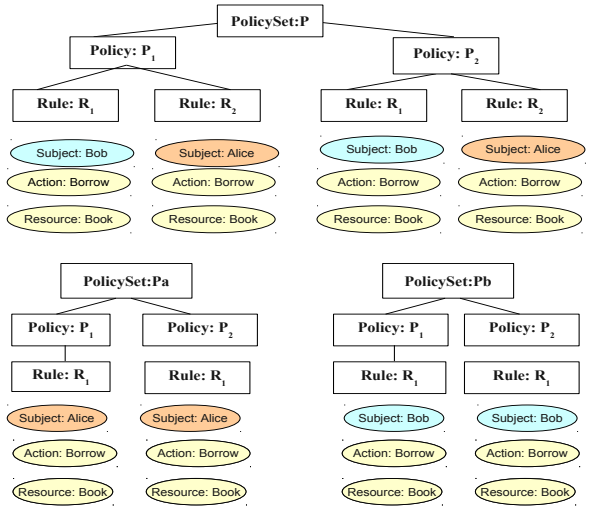


Figure 6: Splitting a Policy according to $SC_1 = \langle Subject \rangle$

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 requirement in the access control architecture, this splitting criterion enables to have a valid refactoring and respects

```
<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 Target Element

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.

A deep analysis of the PEPs at the application enables 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 8 is taken from [18], this code excerpt shows an example of a PEP represented by the method checkSecurity which calls the class SecurityPolicyService that 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

An analysis of this code reflects that the PEP presented by the method ServiceUtils.checkSecurity will trigger exclusively all the rules that have the subject user (provided as input parameter in the PEP) and fixed Action and Resource (LibrarySecurityModel.BORROWBOOK_METHOD), (LibrarySecurityModel.BOOK_VIEW). 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. In this work we consider 3 evaluation studies where the PEPs trigger fixed couples of (action, resource) for variable subjects in the policies, thus the splitting criteria $SC_2 = \langle Resource, Action \rangle$ is considered as the best splitting criteria that enables to preserve the synergy requirement in the architectural model. Inferring automatically the PEPs from the application enables to identify automatically for a given application the best splitting criteria, this can be easily applied using the testing technique presented in [18].

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 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.

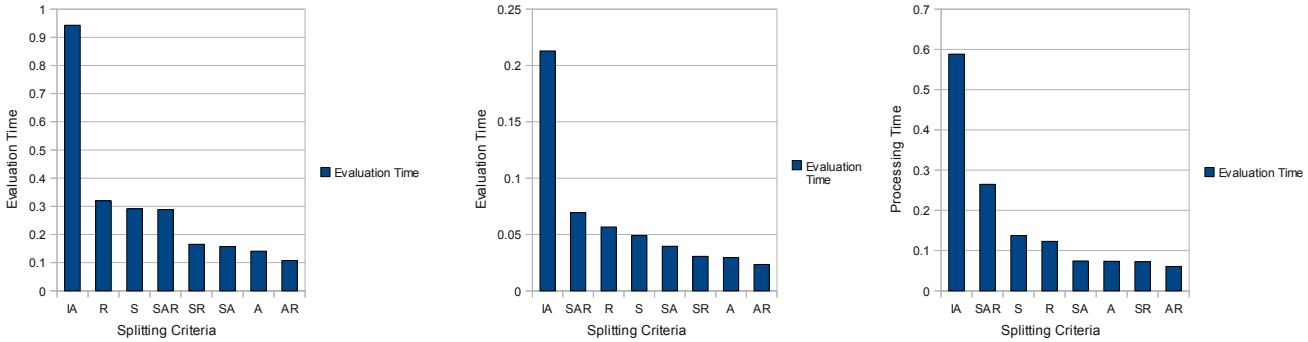
	SAR	AR	SA	SR	R	S	A	IA
SUN	485	922	1453	1875	2578	2703	2703	2625
XEn	26	47	67	95	190	164	120	613

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

From the different tables, we can observe that, when used with a decision engine based on XEngine rather than Sun



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

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

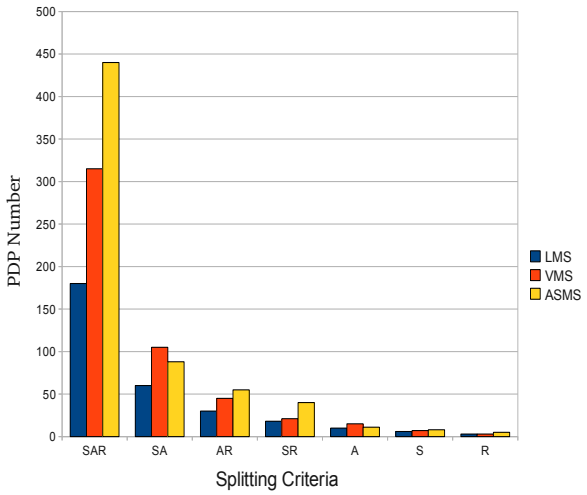


Figure 10: PDP Number According to Splitting Criteria

	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

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. THREATS TO VALIDITY

To find the best splitting criteria in a given application, we made assumptions that method calls are organized by the same distribution of target elements. In our evaluation studies, rules are triggered by functions calls by the couple (Action, Resource), however, this is really related to the organization of the application code, we can find different organizations of method calls.

6. RELATED WORK

Some previous work have focused on performance issues when considering security. In [4], the authors have presented new techniques 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 pathes). 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 whic consists in replacing an expensive check with a cheaper one for frequent codes pathes. At the level of access control security, some contributions have addressed access control policies performance issues. In [8], the authors focus on XACML policy verification for database access control. They presented a model which converts attribute-based policies into access control lists that manage databases resources and they give an implementation that supports their approach. In [13], the authors have proposed an approach 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

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

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.
In [11], the authors decomposed the global XACML policy into local policies related to collaborating parties, the local policies are sent to corresponding PDPs. 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, The authors in [15], 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.

The current contribution brings new dimensions to 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.

7. 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.

8. 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.

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

[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. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. 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. Mière, 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.