Refactoring Access Control Policies for Performance Improvement

Donia El Kateb
Laboratory of Advanced Software SYstems (LASSY)
University of Luxembourg
Luxembourg
donia.elkateb@uni.lu

Tejeddine Mouelhi
Security, Reliability and Trust Interdisciplinary Research Center, SnT
University of Luxembourg
Luxembourg
tejeddine.mouelhi@uni.lu

Yves Le Traon
Laboratory of Advanced Software SYstems (LASSY) &
Security, Reliability and Trust Interdisciplinary Research Center, SnT
University of Luxembourg
Luxembourg
yves.letraon@uni.lu

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

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

ABSTRACT

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

Categories and Subject Descriptors

C.4 [**Performance of systems**]: Performance attributes
; D.4.8 [**Performance**]: Measurements

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA

General Terms

Performance, Design

Keywords

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

1. INTRODUCTION

Access control mechanisms regulate which users could perform which actions on what system resources based on access control policies. Access control policies (i.e., policies in short) are based on various access control models such as Role-Based Access Control (RBAC) [8], Mandatory Access Control (MAC) [6], Discretionary Access Control (DAC) [11], and Organization-Based Access Control (OrBAC) [10]. Access control policies are specified in various policy specification languages such as the eXtensible Access Control Markup Language (XACML) [3] and Enterprise Privacy Authorization Language (EPAL) [1]. In a policy, a rule specifies actions (e.g., read) that subjects (e.g., students) can take on resources (e.g., grades) if required conditions are met. A policy-based system is a system that interacts with user-specified policies to control users' access on resources. A policy-based system allows policy authors to define rules in a policy. In the context of policy-based systems, an access control architecture is often designed with respect to a popular architectural concept that separates Policy Enforcement Points (PEPs) from a Policy Decision Point (PDP) [21]. More specifically, a PEP is located inside an application's code (i.e., business logic of the system). Business logic describes functional algorithms to govern information exchange between access control decision logic and a user interface (i.e., presentation). Given requests (e.g., student A requests to read her grade resource B) formulated by the PEP, the PDP evaluates the requests and returns their responses (e.g., permit or deny) by evaluating these requests against rules in a policy. An important benefit of the architecture is to facilitate managing access rights in a fine-grained way by decoupling the business logic from the

access control decision logic, which can be standardized and separately managed. However, this architecture may cause performance degradation especially when policy authors maintain a single policy with a large number of rules to regulate the whole system's resources. Various factors such as complex and dynamic behaviors of organizations and the growth of organizations's assets may increase the number of rules in the policy [7]. Consider that the policy is centralized with *only* one single PDP. The PDP evaluates requests (issued by PEPs) against the large number of rules in the policy in real-time. Such centralization can be a major factor for degrading performance as our previous work [13] showed that a large number of rules is a challenge for efficient request evaluation. This performance bottleneck issue may impact service availability as well, especially when dealing with a huge number of requests within a short time. In order to address this performance bottleneck issue, we propose an approach to refactoring policies automatically to significantly reduce request evaluation time. As manual refactoring is tedious and error-prone, an important benefit of our automated approach is to reduce significant human efforts as well as improving performance. Our approach includes two techniques: (1) refactoring a policy (handled by single PDP) to its corresponding multiple policies each with a smaller number of rules (handled by multiple PDPs), and (2) preserving architectural property that a single PDP is triggered by a given PEP at a time.

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

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

We collect three subjects of real-life Java systems, each of which interacts with access control policies. The policies consist of a large number of rules. The largest one has 1760 rules, whose corresponding request evaluation faces performance degrade. The policies are specified in XACML [3]. XACML is an XML-based policy specification language popularly used for web-based applications and services. While our subjects are based on XACML policies, our approach could be applicable to any software system that interacts with policies specified in other policy specification languages. As a large number of rules are performance bottlenecks for request evaluation against given policies, our approach could reduce the number of rules in a systematic way for request evaluation against the policies to improve performance.

We conduct an evaluation to show performance improvement achie-ved by our approach in terms of request evaluation time. We leverage two types of PDPs to measure request evaluation time. The first one is the Sun PDP implementation [2], which is a popular open source PDP, and the second one is XEngine [13], which transforms an original 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 policy behaviors of the centralized architectures and the architectural property. Our evaluation results also show that our approach enables reducing the 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 to policies each with a smaller number of rules. This refactoring helps improve performance of request evaluation time.
- We propose a set of splitting criteria to help refactor a policy in a systematic way. Our proposed splitting criteria do not alter policy behaviors of the centralized architectures.
- We conduct an evaluation on three Java systems interacting with XACML policies. We measure performance in terms of request evaluation time. Our evaluation results show that our approach achieves more than nine times faster than that of the centralized architectures in terms of request evaluation time.

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

2. CONTEXT/PROBLEM STATEMENT

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

2.1 Centralization of Architectures

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

2.2 Centralization: A Threat for Performance

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

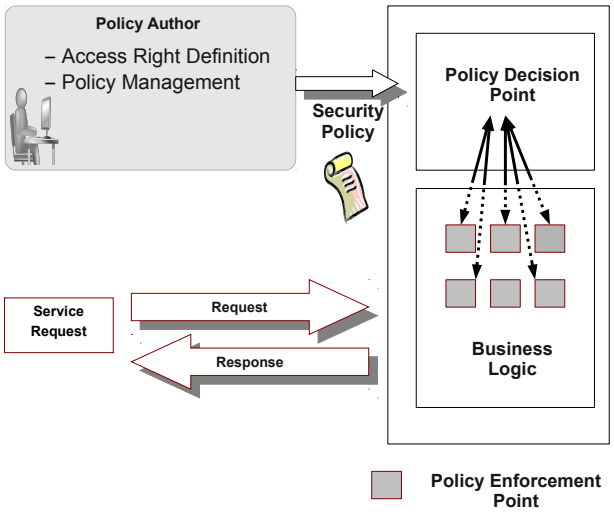


Figure 1: Access Control Request Evaluation

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

The execution time for evaluating a request is thus strongly related to

- the number of rules that the PDP contains as well as the order of the rules [14].
- the workload (the number of requests that have to be managed by the system).

The execution time to evaluate a request depends on the size (number of rules) of the policy that the PDP encapsulates. For a given policy size, the execution time to evaluate requests increases linearly with the workload the (number of requests). Our hypothesis 1 is that the more rules a policy contains, the higher the slope of the execution time with an increasing workloads. *Hypothesis 1* validity is discussed in Section 4. As a consequence, one possibility to improve performance consists in splitting the centralized PDP into PDPs with smaller policy sizes. It is worth to mention that in this paper, we keep the same input channel in the decentralized architecture. In such setting, we do not need to change the PEP code that is calling the PDP. In fact, to set a specific input channel for each PEP, we need to change the code, so that each PEP calls directly its PDP, however, in this work, we consider the system as a black box for sake of simplicity and scalability.

2.3 Centralization: PEPs and PDP Synergy

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

2.4 Tradeoff for refactoring

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

- Access control architectures are centralized with a unique PDP.
- Centralization eases reconfiguration of an access control policy.
- Centralization threatens performance.
- Direct mapping from any PEP to only one PDP makes the access control architecture 'synergic'.
- A synergic system facilitates PEP request routing and eases policy maintenance.

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

2.5 XACML Policies and Performance Issues

In this paper, we focus on access control policies specified in the eXtensible Access Control Modeling Language (XACML) [3]. XACML is an XML-based standard policy specification language

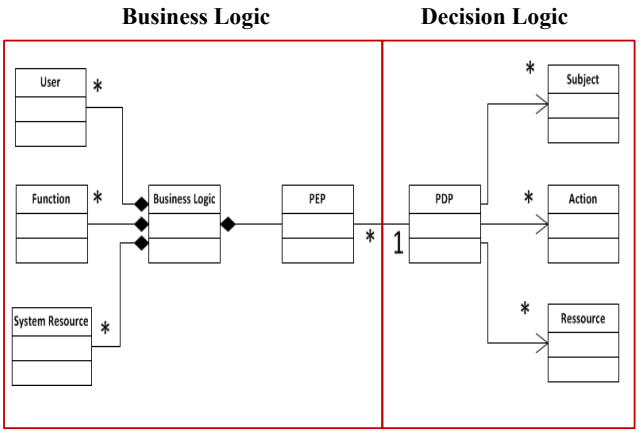


Figure 2: Access Control Model

that defines a syntax of access control policies and requests/responses.

XACML enables policy authors to externalize access control policies for the sake of interoperability since access control policies can be designed independently from the underlying programming language or platform. Such flexibility enables to easily update access control policies to comply with new requirements. An XACML policy is constructed as follows. A `policy set` element consists of a sequence of `policy` elements, a combining algorithm, and a `policy target` element. A `policy` element is expressed through a `target`, a set of `rules`, and a rule combining algorithm. A `target` element consists of the set of resources, subjects, and actions to which a rule is applicable. A `rule` consists of a `target` element, a `condition` element, and an `effect`. A `condition` element is a boolean expression that specifies the environmental context (e.g., time and location restrictions) in which the rule applies. Finally, an `effect` is the rule's authorization decision, which is either permit or deny. Given a request, a PDP evaluates the request against the `rules` in the policy by matching resources, subjects and actions in the request. More specifically, an XACML request encapsulates attributes, that define which subject requests to take action on which resource (e.g., subject Bob requests to borrow a book). Given a request that satisfies `target` and `condition` elements in a rule, the rule's effect is taken as the decision. If the request does not satisfy `target` and `condition` elements in any rule, its response yields the "NotApplicable" decision.

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

- Permit if at least one permit rule is applicable for the 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 the request.

A `policy target` element describes what the policy applies to by referring to attributes of users, resources and actions.

Figure 3 shows a simplified 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>                                                 } Subject
          <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <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

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

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

- An XACML policy may contain various attribute elements including `target` elements. Retrieval of attribute values in the `target` elements for request evaluation may increase the evaluation time.
- A `policy set` consists of a set of policies. Given a request, a PDP determines the final authorization decision (i.e., effect) of the whole `policy set` after combining all the applicable rules' decisions for to the request. Computing and combining applicable rules' decisions contribute to increasing the evaluation time.
- `Condition` elements in rules can be complex because these elements are built from an arbitrary nesting of non-boolean functions and attributes. In such a situation, evaluating `condition` elements may slow down request evaluation time.

3. POLICY REFACTORING

This section describes our approach of refactoring access control policies to improve performance by reducing the number of policy

rules potentially applicable to a request. For refactoring policies in a systematic way, we propose seven policy splitting criteria based on attribute sets. Moreover, we explain how to select a splitting criterion that preserves the synergy in the access control architecture.

3.1 Policy Splitting Criteria

Given a request, a PDP typically uses brute force searching to determine a decision by evaluating the request against all the policy rules one by one until the PDP finds a decision.

During the evaluation process, the attribute values in a given request are compared with the attribute in the target of a rule. If there is a match between the request's attribute and target's attribute values, the rule is then applicable to the request. In decision making process, applicable rules contribute to build the final authorization decision whereas non-applicable rules are not relevant in this process. For request evaluation, not all the rules are applicable to the request. In other words, only part of the rules (i.e, relevant rules) are applicable to the request and can contribute to determining the final decision.

We propose an approach to evaluate a request against only the relevant rules for the given request by refactoring the access control policies. Our approach aims at splitting a single global policy into multiple smaller policies based on attribute combination. For a given policy-based system, we transform its policy P into policies P_{SC_w} containing a less number of rules and conforming to a Splitting Criterion SC_w. An SC_w defines the set of attributes that are considered to classify all the rules into subsets each with the same attribute values and w denotes the number of attributes that have to be considered conjointly for aggregating rules based on specific attribute elements. Table 1 shows our proposed splitting criteria categorized according to attribute element combinations.

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

To illustrate our approach, we present examples that take into consideration the XACML language features. In Figure 4, 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).

Technically, to split a given policy P according to $SC_1 = \langle Subject\rangle$, we start by parsing the global policy P and by collecting the overall subject attributes in the policy. For each collected subject attribute Sa, we consider the global policy and we delete the rules that do not contain Sa as a subject attribute in the target element attributes. After all the successive deletions, the global policy is refactored to a policy that contains only the rules with Sa in their subject attribute elements. Algorithm 1 describes the splitting process for $SC_1 = \langle Subject\rangle$.

Our algorithm is safe in the sense that it does not change the authorization behavior of the PDP. There are two important issues to be considered when reasoning about the safety of the algorithm:

- Can the splitting impact authorization results when a policy set includes multiple policies with different combining algorithms?
- When 'AnySubject', 'AnyAction', 'AnyResource' is used as target element values, does the splitting change the behavior of the PDP?

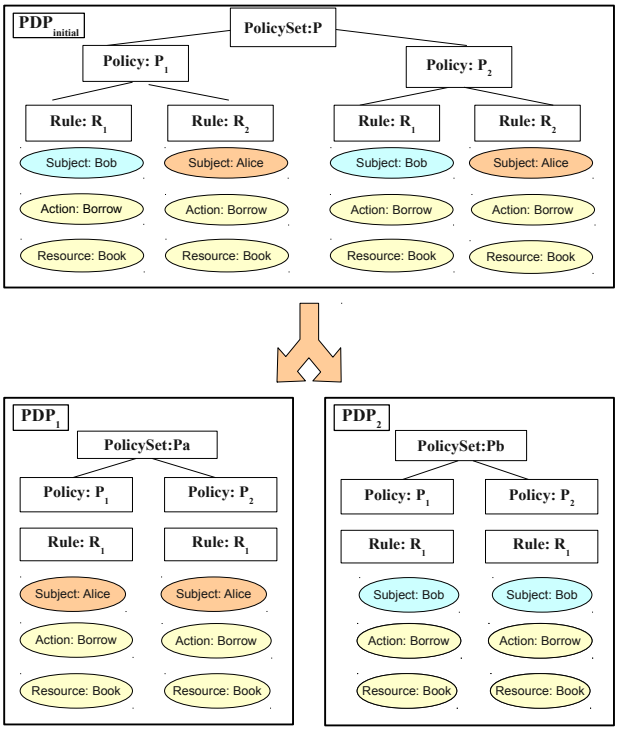


Figure 4: Refactoring a policy according to $SC_1 = \langle Subject\rangle$

The first issue is addressed by the way the algorithm operates. The first step of the algorithm goes through all the rules and extracts depending on the splitting criterion, the set of target elements (the set of subjects, the set of actions, and/or the set of resources). Then, based on the extracted result, the splitting is performed by removing the rules with different splitting criterion values (such as a subject different from the splitting criterion subject). The rules that are kept are therefore not modified and therefore their behavior is not altered. When there are several policies with different combining algorithms, the rules that are kept do not impact the evaluation behavior because they remain attached to the same combining algorithm. Moreover their order and their content are not modified.

The second issue is handled by keeping all the rules that involve 'AnySubject', 'AnyAction' or 'AnyResource' because by definition during evaluation, these values are taken into consideration for evaluating all possible values of subjects, actions, and resources. Therefore, it is important to keep them in all split policies because they are used in all evaluations, this way, the behavior is not changed because these rules are present in all split policies and their number is negligible compared to those existing in specific targets.

It is worth mentioning this following consideration related to the refactoring process: XACML supports multi-valued attributes in policies and requests. In XACML policies, `target` elements define a set of attribute values, which match with the context element in an access control request. In Figure 5, the subject at-

Algorithm 1 Policy Splitting Algorithm

Input: XACML Policy P, Splitting Criterion SC_1=$\langle Subject \rangle$
Output: Sub-policies Set: S
SplitPolicy()
S=Ø
/* Collect all subjects in all the rules /*
for all Rule R_i in Policy P **do**
  /* Fetch all the targets to extract attribute collection depending on SC */
  for all Target.Subject in R_i **do**
    SubjectCollection.add(SubjectElement.attribute)
  end for
end for
/* Build sub-policies based on subjects collected in SubjectCollection */
for int $i = 0$; $i <$ SubjectCollection.size(); $i++$ **do**
  /* Remove all the rules that do not contain SubjectCollection.at(i) in their Target */
  for all Rule R_i in Policy P **do**
    if $R_i.Target.SubjectElement$! = AnySubject **then**
      if (Target.SubjectElement.attribute in R_i) ! = SubjectCollection.at(i) **then**
        Remove R_i
      end if
    end if
  end for
  /* $P_{(SubjectCollection.at(i))}$ is a sub-policy with only rules where the subjectAttribute is equal to SubjectCollection.at(i) */
  /* Add the sub-policy to the set of sub-policies */
  $S = S \cup P_{(SubjectCollection.at(i))}$
end for

tribute 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 split by the policy splitter component.

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

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

After the splitting is performed, our approach creates one or more PDPs that comply with the splitting criterion. We use Sun PDP [3] to evaluate requests against policies specified in XACML. During request evaluation, Sun PDP checks the request against the policy and determines whether the decision is permit or deny. Given a request, our approach runs Sun PDP loaded with the request's relevant policy, which is used during the decision making process. The PDP then retrieves the rules that are applicable to the request. Figure 6 presents our approach to handling request evaluation with multiple policies. During the evaluation process, given a request, our approach verifies the matching between the request's attribute values, and the policy target elements attributes. Our approach then selects only the relevant policy among all the policies for a given request. After the selection of the relevant policy, all of its relevant rules for the decision making are evaluated. Figure

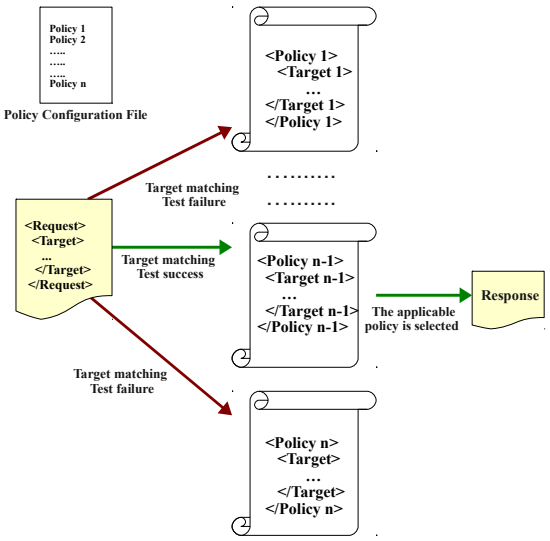


Figure 6: Applicable-Policy Selection

7 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 the initial global policy, the policy splitter component conducts automated refactoring by creating multiple PDPs loaded with XACML policies, which are split from the initial global policy based on the user-specified splitting criterion. If the initial global policy is changed, the policy splitter component is required to refactor the policy again to create PDPs with the most recent relevant policies. Our refactoring approach is safe in the sense that the approach does not impact existing security aspects in a given system.

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 the relevant policy for a request dynamically at runtime. As shown in Section 3.1, given multiple PDPs after the policy refactoring, we consider (1) how PEPs are organized at the application level, and (2) how PEPs are linked to their corresponding PDPs. In the worst case, splitting the initial PDP into multiple PDPs may lead to a non-synergic system: a PEP may send its requests to several PDPs. The PDP, that handles a given 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 between PEPs to PDPs. Given a request, our approach maps a PEP to a PDP with relevant rules for the request. Threfore, different requests issued from a PEP should be handled by the same PDP. Operationally, the request evaluation involves one policy. In this case, our refactoring does not impact the conceptual architecture of the system.

Figure 8 presents a PDP encapsulating a global policy that has

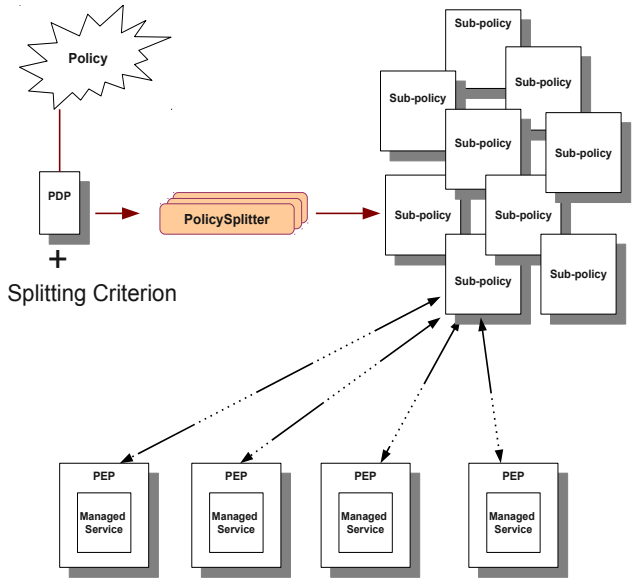


Figure 7: Overview of the Refactoring Process

been refactored. The system that is presented on the left is a desirable refactoring whereas the one on the right shows an undesirable refactoring.

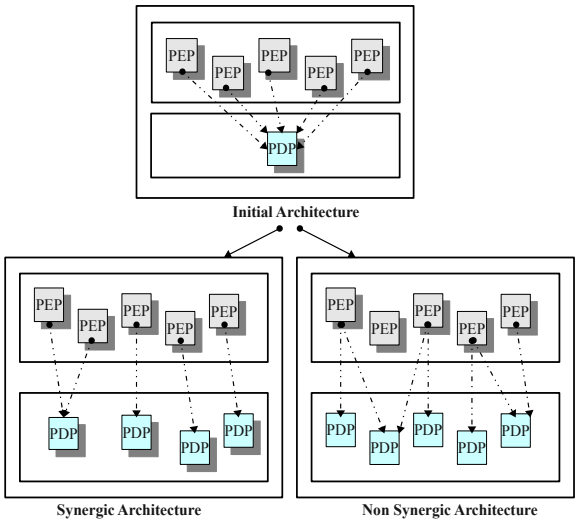


Figure 8: Synergic vs Non synergic System

At the application level, the PEP is represented by a method call that triggers a decision making process. Figure 9 presents sample PEP code from our previous work in [20]. This code snippet shows an example of a PEP represented by the method `checkSecurity`, which calls a method of the class `SecurityPolicyService`, which formulates a request to invoke the PDP component. The PEP represented by the method `ServiceUtils.checkSecurity` may issue requests that have subject "user" along fixed action and resource ("LibrarySecurityModel.BORROWBOOK_METHOD"), ("LibrarySecurityModel.BOOK_VIEW"). Consider that we refactor a policy using $SC_2 = \langle Resource, Action \rangle$, $SC_1 = \langle Action \rangle$ or $SC_1 = \langle Resource \rangle$. Given a request issued from the PEP, our approach runs a PDP loaded with a policy containing rules sharing the same action and resource attribute values. Thus the splitting process that preserves the mapping between the PEPs and the PDP is the one that considers the following splitting criteria: $SC_2 = \langle Resource, Action \rangle$, $SC_1 = \langle Action \rangle$ and $SC_1 = \langle Resource \rangle$ in this case. In the evaluation section, we investigate the impact of the synergy property on performance.

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

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

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

Figure 9: PEP Deployment Example

4. EVALUATION

We carried out our evaluation on a desktop PC running Ubuntu 10.04 with a Core i5, 2530 Mhz processor, and 4 GB of RAM. We have implemented a tool, called `PolicySplitter` to split policies according to a given splitting criterion automatically. The tool is implemented in Java and is available for download [4].

4.1 Objectives and Metrics

Our evaluation intends to answer the following research questions:

1. **RQ1.** How faster can request processing time of multiple Sun PDPs with policies split by our approach achieve compared to an existing single Sun PDP? This question helps show that our approach can improve performance in terms of request processing time. Moreover, we compare request processing time for different splitting criteria.

2. **RQ2.** This research question, questioned in Section 2, investigates *Hypothesis 2*: With comparable PDP policy sizes? Is the system more performant when the architecture is synergic?

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

4. **RQ4.** This research question investigates *Hypothesis 1* (mentioned in Section 2) on the impact of the number of rules in a given PDP on system response: the bigger the PDP policy size, do we observe higher slope of the execution time with an increasing workload?

To address these research questions, we go through the following evaluation setup based on two different empirical studies:

- First, we evaluate the performance improvement regarding the decision making process by taking into consideration the whole system (PEPs and PDPs). We compared request processing time with a single global policy (handled by a single PDP) against request processing time with split policies. All the splitting criteria have been considered in our evaluation. `IA` denotes an "Initial Architecture", which uses the single global policy for request processing. This step allows studying the behavior of splitting criteria that preserve the synergy property in the access control architecture.

- Second, we evaluate the performance of PDPs in isolation to compare the splitting criteria independently from the system. Recall that in this step, we do not reason about the synergy property since we don't consider the application level. We replace Sun PDPs with a request evaluation engine XEngine [13]. The objective of this evaluation is to investigate how our approach impacts performance for subjects combined with XEngine.

4.2 Subjects

The subjects include three real-life Java systems each of which interacts with access control policies. Full details on our subjects are available in [18]. We next describe our three subjects.

- Library Management System (LMS) provides web services to manage books in a public library.

- Virtual Meeting System (VMS) provides web conference services. VMS allows users to organize online meetings in a distributed platform.

- Auction Sale Management System (ASMS) allows users to buy or sell items online. A seller initiates an auction by submitting a description of an item that she wants to sell with its expected minimum price. Users then participate in the bidding process by bidding the item. To bid on the item, user must have enough money in her account before bidding.

Our subjects are initially built upon Sun PDP [3] as a decision engine, which is a popularly used PDP to evaluate requests. Policies in LMS, VMS, and ASMS contain a total of 720, 945 and 1760 rules, respectively. Moreover, to compare performance improvement over existing PDPs, we adopt XEngine (instead of Sun PDP) in our subjects to evaluate requests. XEngine is an advanced policy evaluation engine, which transforms the hierarchical tree structure of the XACML policy to a flat structure to reduce request processing time. XEngine also handles various combining algorithms supported by XACML.

4.3 Performance Improvement: Sun PDP

In order to answer **RQ1**, we generated the resulting sub-policies for all the splitting criteria defined in Section 3.1. For each splitting criterion, we have executed system tests to generate requests that trigger all the PEPs in the evaluation. The test generation step leads to the execution of all combinations of possible requests described in our previous work [19]. The process of test generation is repeated ten times to alleviate the impact of randomness. We applied this process to each splitting criterion and calculated evaluation time on average of a system under test. Figure 10 presents evaluation time for policies split based on each splitting criterion and the global policy of subjects. We can make two observations:

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

- The splitting criterion $SC = \langle Action, Resource \rangle$ enables to show the shortest evaluation time. Recall that the PEPs in the evaluation are scattered across different methods in a subject by a categorization based on $SC_2 = \langle Resource, Action \rangle$. This observation pleads in favor of applying a splitting criterion that takes into account the PEP-PDP synergy.

	S	A	R	SA	SR	AR	SAR	IA
Synergic		x	x			x		x
Not Synergic	x			x	x		x	

Table 2: Splitting Criterion Classification

To identify the splitting criterion that generates the smallest number of PDPs, we have studied the number of policies generated by the splitting. Figure 11 shows the results. We observed the number of policies based on our proposed three categories: (1) the SC_1 category leads to the smallest number N_1 of PDPs, (2) the SC_2 category leads to a reasonable number N_2 (N_1<N_2<N_3) of PDPs, and (3) SC_3 leads to the largest number N_3 of PDPs. While the SC_1 category leads to the smallest number of PDPs, each PDP encapsulates a relatively high number of rules in a policy (compared with that of SC_2 and SC_3, which leads to performance degradation.

We have classified splitting criteria according to their preservation of the synergy property shown in Table 2. AR, A and R are synergic splitting criteria since all the PEPs in our three systems are organized as shown in Figure 9.

To answer **RQ2**, we have evaluated PDPs in the three systems and for the different splitting criteria. The results presented in Figure 12 show the average number of rules in each PDP, for each splitting criterion in the three systems. We can observe that the AR criterion produces comparable size of PDPs with the SR criterion, however, as shown in Figure 10, AR is the best splitting criterion, in term of evaluation time performance. Moreover, the number of PDPs produced with the splitting critera S and A is comparable and the criterion A which is synergic has evaluation time less than the one produced by the splitting criterion S which is not synergic. This result supports our hypothesis *Hypothesis 2*, which states that with comparable PDP sizes, the overall system would be more performant when the architecture is synergic.

4.4 Performance Improvement: XEngine

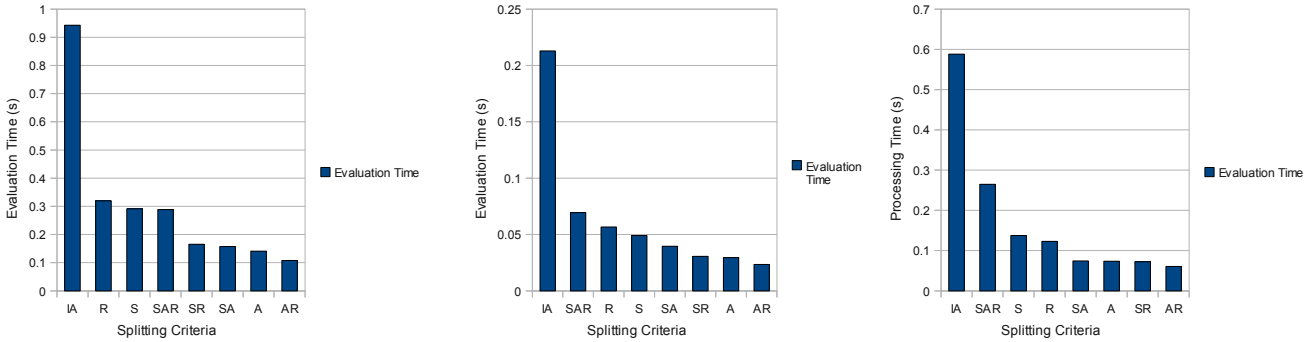
In order to answer **RQ3**, we measure request processing time of XEngine compared with that of Sun PDP for policies split by our approach. The goal of this empirical study is to show the impact of combining XEngine with our splitting process. XEngine improves dramatically the performance of the PDP, mainly for three 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 relies 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 two settings:



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

Figure 10: Request Processing Time for the Three subjects

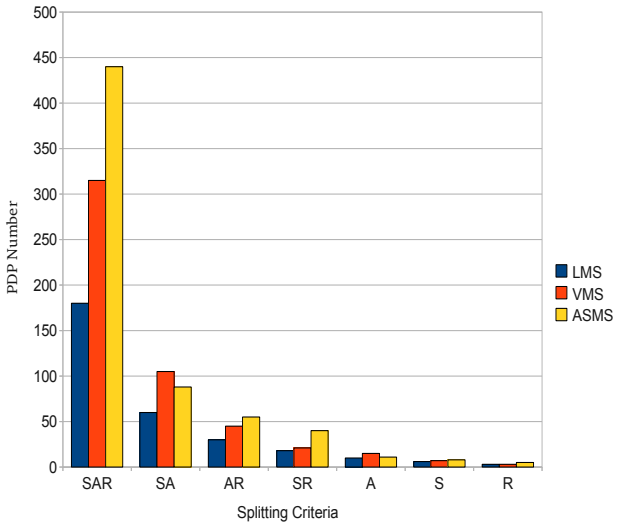


Figure 11: PDP Number produced with Splitting Criteria

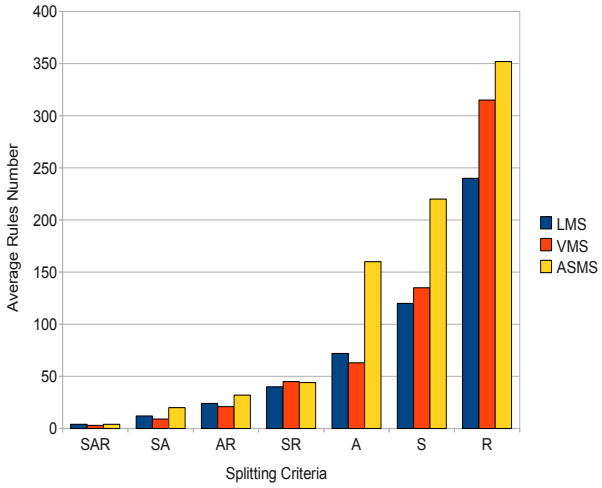


Figure 12: Average of Rules Number/PDP in the Three Systems

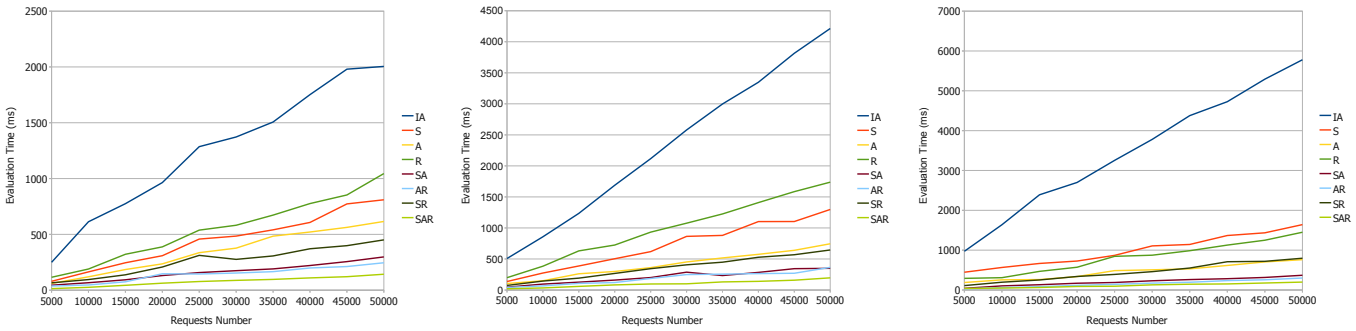
- Considering evaluation with a decision engine based on Sun PDP with split policies and with the initial policy.
- Considering evaluation with a decision engine based on XEngine with split policies and with the initial policy.

In this step, we do not reason about the synergy, since we do not consider the application level for the three systems. We measure request processing time by evaluating randomly generated set of 10,000 requests as proposed in our previous work [15]. The request time processing is evaluated for the three systems. The results bare presented in Tables 3, 4 and 5 and enable to answer **RQ3**. We observe that, when our subjects are equipped with XEngine, our proposed approach substantially improves performance (compared to the results with Sun PDP). For the splitting criteria SC=$\langle Action \rangle$, in the LMS system, the evaluation time is reduced about 9 times: from 2703 ms to 290 ms with XEngine. This empirical observation pleads in favor of applying our proposed refactoring process with XEngine.

4.5 Impact of Increasing Workload

To investigate **RQ4**, we evaluated request processing time according to the number of requests incoming to a system. For each policy in the three systems (ASMS, LMS, and VMS), we generated 5000, 10000, .., 50000 random requests to measure the evaluation time (ms). The results are shown in Figure 13. For the three systems, we observe that the evaluation time increases when the number of requests increases in a system. With an increasing system load, the request evaluation time is considerably improved when using the splitting process compared to the initial architecture. The results shown in Figure 13 are interpreted by the average of PDPs size presented in Figure 12. The result is consistent with *Hypothesis 1* (mentioned in Section 2), which states that the slope of execution time increases with PDPs size in a system with an increasing workload.

It is worth to mention that, to be able to deploy our technique, we need to fetch the relevant PDP for a given request at runtime. Therefore, request processing time includes both fetching time and request evaluation time. Figure 14 shows percentage of fetching



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

Figure 13: Processing Time for our subjects, LMS, VMS and ASMS depending on the requests Number

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

Table 3: Evaluation Time in LMS

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

Table 4: Evaluation Time in VMS

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

Table 5: Evaluation Time in ASMS

time over the global evaluation time for request evaluation in LMS. The fetching time increases according to the PDP size. The fetching time is relatively small in comparison with the total evaluation time and thus does not impact significantly the decision making time.

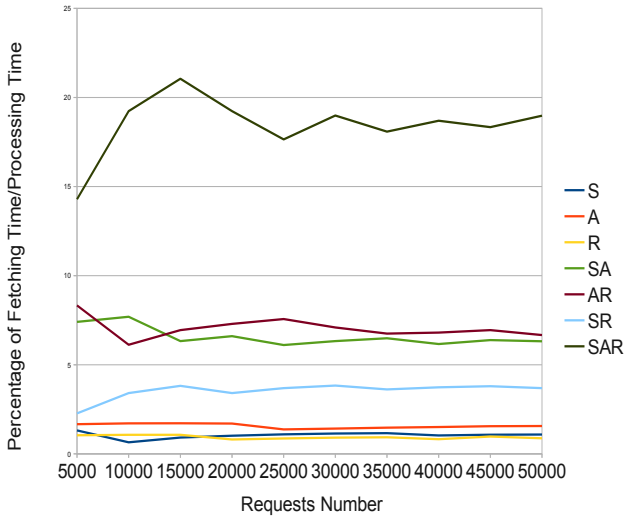


Figure 14: Percentage of Fetching Time

4.6 Summary

We summarize the results of the evaluation section:

- We have experimentally shown the effectiveness of the splitting in reducing the decision making time. Our refactoring process is more effective when it uses XEngine rather than Sun PDP to evaluate requests.
- When the sizes of PDPs are comparable, the splitting criteria that are synergic enable to have the best results in terms of decision making time.

The evaluation of the synergy property on improving performance has to be strengthened by conductinng others experiments on other evaluation studies and by considering different organizations of PEPs at the application level.

4.7 Threats to Validity

The threats to external validity primarily include the degree to which subjects, policies, splitting criteria and test requests are representative of true practice. These threats could be reduced by further evaluation on a wider type and larger number of subjects and policies and a larger number of test requests in future work. In particular, our approach is based on only seven proposed splitting criteria. We could develop additional splitting criteria to split policies and measure efficiency in terms of request processing time. In addition, our approach generates random test requests, which may bias our results. To prevent such a bias, we conduct our evaluation for 10 times and measure an average value of evaluation results. The threats to internal validity are instrumentation effects that can bias our results such as faults in Sun PDP, XEngine, `PolicySplitter tool`, measurement tool in terms of request processing, and random request generators.

5. RELATED WORK

There are several previous work about performance issues in security mechanisms. Ammons et al. [5] have presented 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 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 [9,12,14] have been proposed to address performance issues in systems interacting with access control policies. Jahid et al. [9] focus on XACML policy verification for database access control. 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 do not show how much MyABDAC gains improvement over an existing PDP.

Marouf et al. [14] 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 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. [12] 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, Miseldine et al. [16] 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 over our previous work on access control [13,17,19]. We have proposed XEngine [13], which focuses particularly on performance issues addressed with XACML policies evaluation. XEngine proposes an alternative solution to brute force searching based on an XACML policy conversion to a tree structure to minimize the request evaluation time. It involves a refactoring process that transforms the global policy to a decision diagram that is then converted to forwarding tables. In this current contribution, we introduce a new refactoring process that involves splitting the policy into smaller sub-policies. Our two refactoring processes are combined to decrease dramatically the evaluation time.

6. CONCLUSION AND FUTURE WORK

In this paper, we have tackled the performance issue in access control decision making mechanism and we have proposed an automated refactoring process that enables to reduce access control policies evaluation time up to 9 times. All the reasonings about performance factors have not included hardware considerations. However, performance improvement at the software/logical level as done by our approach would complement performance improvement at the hardware level to best improve the overall performance. 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 pleads 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. In 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 automatically the most suitable splitting criterion for a given application.

7. REFERENCES

[1] IBM, Enterprise Privacy Authorization Language (EPAL), Version 1.2 . http://www.w3.org/Submission/2003/SUBM-EPAL-20031110, 2003.

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

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

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

[5] 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.

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

[7] J. Chaudhry, T. Palpanas, P. Andritsos, and A. Mana. Entity lifecycle management for okkam. In *Proc. 1st IRSW2008 International Workshop on Tenerife*, 2008.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] T. Mouelhi, Y. L. Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing, 2009.

[19] 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.

[20] 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.

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