J Supercomput (2012) 59:1577–1595
DOI 10.1007/s11227-011-0569-5

Towards high performance security policy evaluation

Zheng Qin · Fei Chen · Qiang Wang · Alex X. Liu · Zhiguang Qin

Published online: 19 February 2011

Abstract The Enterprise Privacy Authorization Language (EPAL) is a formal language for specifying fine-grained enterprise privacy policies. With the adoption of EPAL, especially in web applications, the performance of EPAL policy evaluation engines becomes a critical issue. In this paper, we propose Eengine, an engine for efficient EPAL policy evaluation. Eengine first converts all string values in an EPAL policy to numerical values. Second, it converts a numericalized EPAL policy specified as a list of rules following the first-match semantics to a tree structure for efficient processing of numericalized requests.

Keywords Policy evaluation · EPAL · Web server · Access control

The work of Qiang Wang is done during his visit at Michigan State University.

Z. Qin
College of Software, Hunan University, Changsha, Hunan 410082, China
e-mail: qinzheng@hnu.cn

F. Chen (✉) · A.X. Liu
Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824-1266, USA
e-mail: feichen@cse.msu.edu

A.X. Liu
e-mail: alexliu@cse.msu.edu

Q. Wang · Z. Qin
Department of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China

Q. Wang
e-mail: qinzg@uestc.edu.cn

Z. Qin
e-mail: qinzg@uestc.edu.cn

1 Introduction

The Enterprise Privacy Authorization Language (EPAL) [1] is an XML-based markup language used to describe and enforce internal enterprise privacy policies. It addresses the need of an enterprise to specify access control policies, with reference to attributes/properties of the requestor, to protect private information. EPAL is designed to enable organizations to translate their privacy policies into a structured format such that they can exchange and communicate their privacy policies.

Typical EPAL based access control works as follows. A user (e.g., a professor) wants to perform an action (e.g., change) on protected data (e.g., student grade) with a purpose (e.g., assign the grade). The user submits this request using the *EPAL authorization request language* to the *EPAL evaluation engine*. The EPAL evaluation engine checks the request with its EPAL policy and determines whether the EPAL request should be allowed or denied and enforces the decision.

This paper concerns the process of checking whether a request satisfies a policy, which we call *policy evaluation*. We refer to a policy decision point (PDP) [1] as a *policy evaluation engine*. Specifically, this paper concerns the performance of EPAL policy evaluation engines. With the adoption of EPAL, especially in online applications running on IBM servers, the performance of EPAL policy evaluation engines becomes a critical issue. When a web server needs to enforce an EPAL policy with a large number of rules, its EPAL policy evaluation engine may easily become the performance bottleneck for the server. As the number of resources and users managed by web servers grows rapidly, EPAL policies grow correspondingly in size and complexity. To enable an EPAL policy evaluation engine to process simultaneous requests of large quantities in real time, especially in face of a burst volume of requests, an efficient EPAL policy evaluation engine is necessary. However, commercial implementations of EPAL policy evaluation engines such as IBM EPAL Policy Decision Point (PDP), which performs brute force searching by comparing a request with a list of rules in an EPAL policy, still represent the state-of-the-art. To our best knowledge, there is no prior research work on improving the performance of EPAL policy evaluation engines. This paper represents the first step in exploring this unknown space. Making EPAL policy evaluation more efficient is difficult from many aspects. First, the EPAL vocabulary, which is the most important component of the EPAL policy, can have complex structures. For example, in the EPAL vocabulary, user categories (data categories, or purposes) can have hierarchical structures. Second, an EPAL policy often has conflicting rules, and they are resolved by first-match semantics. It is natural for an EPAL policy evaluation engine to examine all the rules in an EPAL policy until finding a rule that matches the request. Third, the first-match semantics for solving conflicting rules is complicated because of two factors. One is the different decisions of rules, e.g., *deny* or *allow*. The other is which user category the subject of the request belongs to. Fourth, in EPAL, a request could be an uncertainly valued request. For example, the subject of an EPAL request may have one user category "an employee," which can be "a professor" or "a secretary." Last but not the least, in EPAL policies, a request could be a compound request. For example, the subject of an EPAL request may have multiple user categories, which is not only "a professor of computer science" but also "a professor of electrical engineering." In retrospect, the

high complexity of EPAL policies makes brute force searching appear to be the natural way of processing requests. In this paper, we present *Eengine*, a fast and scalable EPAL policy evaluation engine. Eengine has two key ideas. First, Eengine converts all string values in an EPAL policy to numerical values. In processing requests, Eengine also converts the string values in a request to their corresponding numerical values. We call this technique *EPAL policy numericalization*. Second, Eengine converts a numericalized EPAL policy with a list of rules and one conflict resolution mechanism into a tree structure, and uses it to efficiently process numericalized requests. Intuitively, Eengine outperforms the standard EPAL policy evaluation engines (which we call IBM PDP) for three major reasons. First, in checking whether a request satisfies a predicate, Eengine uses efficient numerical comparison thanks to the EPAL policy numericalization technique. Second, when it receives a request, Eengine can find the correct decision without comparing the request with a list of rules in the policy due to the EPAL policy normalization technique. Third, EPAL numericalization and normalization open many new opportunities for building efficient data structures for fast request processing.

To evaluate the performance of Eengine, we compare it with the standard IBM PDP implementation. We conducted extensive experiments on synthetic policies with different number of rules. The experimental results show that Eengine is orders of magnitude more efficient than IBM PDP, and the performance difference between Eengine and IBM PDP grows almost linearly with the number of rules in EPAL policies.

As the core of access control systems, the correctness of Eengine is critical. Thus, we not only formally prove that Eengine makes the correct decision for every request based on the EPAL 1.2 specification [1], but also empirically validate the correctness of Eengine in our experiments.

In our experiments, we first randomly generate 10,000 single-valued requests and 10,000 uncertainly valued requests; we then feed each request to Eengine and IBM PDP and compare their decisions. The experimental results confirmed that Eengine and IBM PDP are functionally equivalent.

The rest of the paper is organized as follows. We first review related work in Sect. 2. In Sect. 3, we briefly introduce EPAL. In Sect. 4, we describe the policy numericalization and normalization techniques. In Sect. 5, we present the algorithms for processing requests. We prove the correctness of our the policy numericalization and normalization techniques in Sect. 6. In Sect. 7, we present our experimental results. Finally, we give concluding remarks in Sect. 8.

2 Related work

Prior work that is closest to ours is the XEngine [2], which is an XACML policy evaluation engine. Because both EPAL and XACML are similar in terms of functionality, the basic idea of XEngine and Eengine is similar, which aims to convert EPAL (or XACML) policies to tree structures and uses them to efficiently process numericalized requests. However, in terms of semantics, there are three major differences between EPAL policies and XACML policies [3], which lead to different solutions

for Eengine and XEngine. First, EPAL polices have flat structures, i.e., an EPAL policy only includes a list of rules. In contrast, XACML policies have hierarchical structures, i.e., an XACML policy consists of policies or policy sets and a policy set further consists of policies or policy sets. Second, the vocabulary of an EPAL policy is an important component and has a complex structure; in contrast, the vocabulary of an XACML policy is an optional component. The vocabulary of an EPAL policy has six elements and three of them have hierarchical structures. Third, EPAL policies have one rule combining algorithm (i.e., first-match); in contrast, XACML policies have four rule/policy combination algorithms. Therefore, the algorithms in XEngine cannot to applied to EPAL policies and we need to propose new algorithms for our EPAL policy evaluation engine.

Beyond XEngine, very few previous work focus on high performance application-level policy evaluation [4–6]. Borders et al. proposed a new access control policy language which was designed with the goal of high performance policy evaluation [4]. In contrast, we do not invent new policy languages or modify existing ones. Crampton et al. proposed a caching mechanism to speed up policy evaluation [5, 6]. In contrast, we may employ their caching mechanism as a technique at the evaluation engine level. Thus, the caching mechanism is transparent to application developers in our scheme, but not in [5, 6].

One of the relevant areas of EPAL policy evaluation is packet classification (e.g., [7, 8]), which focuses on the performance of checking a packet using a packet classifier. A classifier is a collection of rules or policies. Although similar in spirit, packet classification is different from EPAL policy evaluation. First, EPAL rules are specified using application specific string values, while packet classification rules are specified using ranges and prefixes. Second, some components of the vocabulary in EPAL policies have hierarchical structures; in contrast, the structure of packet classifiers is flat. Third, the number of possible values that a request attribute can be small, while the number of possible values that a packet field can be much bigger (e.g., 2^{32}). These differences mean that it is usually not suitable to directly apply prior packet classification algorithms to EPAL policy evaluation. Our procedures of EPAL policy numericalization and normalization take a step in bridging the two fields. Although packet classification concerns access control in the network level while EPAL policy evaluation works in the application level, they share essential characteristics.

Other works on the EPAL policy language focused on the policy analysis, extension, application, and comparison with other access control languages. Stufflebeam et al. [9] investigated how to deploy EPAL and P3P policies into real systems and measured the results against security requirements. Hung [10] briefly summarized the research issues of web services privacy technologies including the EPAL policy language. Backes et al. [11] studied unification over EPAL policies such that policy evaluation engines can support general queries and quantitative measurements. Barth et al. [12] proposed an extended version of the EPAL language, which can support guaranteed safety, local reasoning, and closure under combination. Barth and Mitchell [13] presented a data-centric model to check whether the deployed policies, e.g., EPAL policies meets the access control requirements.

3 Background

An EPAL policy consists of a vocabulary and a list of privacy rules. A *vocabulary* defines six elements:

(1) User categories that define the roles of all requestors; (2) data categories that define protected data; (3) actions that define the proposed actions; (4) purposes that define the purposes of all proposed actions; (5) containers that define external context data that can be evaluated by conditions; (6) obligations that define (normally) external requirements for each requestor to fulfill as a condition of doing the requested action (e.g., grades may be modified, but must be deleted after 3 years). Note that three elements, user categories, data categories and purposes can have hierarchical structures. For example, two user categories "professor" and "secretary" belong to one user category "university employee," which indicates that a university employee can be a professor or a secretary.

In this paper, if category c_1 belongs to another category c_2, we call category c_1 a descendant of category c_2 and category c_2 an ancestor of category c_1. While the other three elements, actions, containers and obligations have a list of items and there is no ordering on these items. A list of *privacy rules* follows first-match semantics, that is, the decision for a request is the decision of the first rule that request matches. A *rule* consists of a ruling (e.g., *deny* or *allow*), a user category, an action, a data category, and a purpose. A rule may also contain conditions and obligations.

Figure 1 shows an example EPAL policy accompanied by its associated vocabulary. In the vocabulary (lines 1–13), lines 4–6 define three user-categories, which indicates that an employee can be a professor or a secretary; line 7 defines one data category, grades; lines 8–10 define three purposes, which indicates that one purpose "manage" consists of two purposes "grading" and "reading;" lines 11–12 define two actions, change and read. The example EPAL policy (lines 14–35) includes three rules: lines 17–22 define the first (*deny*) rule, whose meaning is that a secretary cannot change grades with the purpose "grading;" lines 23–28 define the second (*allow*) rule, whose meaning is that an employee can change grades with the purpose "manage;" lines 29–34 define the third (*allow*) rule, whose meaning is that a professor can read grades with the purpose "manage." Note that "default-ruling" at line 14 indicates that if a request does not match any rule in the EPAL policy, the decision *deny* should be returned.

4 EPAL policy numericalization and normalization

The process of EPAL policy numericalization is to convert the string values in an EPAL policy into integer values. This numericalization technique enables our EPAL policy evaluation engine to use efficient integer comparison, instead of inefficient string matching, in processing EPAL requests. The process of EPAL policy normalization is to convert an EPAL policy with a list of rules into an equivalent tree structure, the policy decision diagram (PDD). This normalization technique enables our EPAL policy evaluation engine to process an EPAL request without comparing the request against all the rules until finding the rule that matches the request.

```
<epal-vocabulary>
  <vocabulary-information id="GradeManagement">
  </vocabulary-information>

  <user-category id="employee"></user-category>
  <user-category id="professor" parent="employee"></user-category>
  <user-category id="secretary" parent="employee"></user-category>

  <data-category id="grades"></data-category>

  <purpose id="manage"></purpose>
  <purpose id="grading" parent="manage"></purpose>
  <purpose id="reading" parent="manage"></purpose>

  <action id="change"></action>
  <action id="read"></action>
</epal-vocabulary>

<epal-policy default-ruling="deny">
  <policy-information id="n"></policy-information>

  <epal-vocabulary-ref id="GradeManagement"/>

  <rule id="1" ruling="deny">
    <user-category refid="secretary"/>
    <data-category refid="grades"/>
    <purpose refid="grading"/>
    <action refid="change"/>
  </rule>

  <rule id="2" ruling="allow">
    <user-category refid="employee"/>
    <data-category refid="grades"/>
    <purpose refid="manage"/>
    <action refid="change"/>
  </rule>

  <rule id="3" ruling="allow">
    <user-category refid="professor"/>
    <data-category refid="grades"/>
    <purpose refid="manage"/>
    <action refid="read"/>
  </rule>
</epal-policy>
```

Fig. 1 An example EPAL policy

There are many technical challenges in EPAL policy numericalization and normalization: noninteger values, uncertainly valued requests, compound requests, and inefficient sequential searching. Next, for each challenge, we formulate the problem, present our solution, and give an illustrating example.

4.1 EPAL policy numericalization

Problem In sequential range rules, the constraints on each attribute are specified using integers. However, in EPAL rules, the constraints on each attribute are specified using ASCII strings in the EPAL vocabulary of the EPAL policy, and three elements

in the EPAL vocabulary, user categories, data categories, and purposes, can have hierarchical structures.

Solution For user categories (data categories, or purposes), if they have a hierarchical structure, we parse and model them as a tree, where each terminal node represents only one item, and each nonterminal node represents the set of items represented by its children. Because this tree represents the structure of user categories (data categories, or purposes), we call it the structure tree of user categories (data categories, or purposes). At each node, we store the corresponding ASCII string. Next, for each structure tree, we first map the string in each terminal node to a set that contains a distinct integer, and all the mapped integers of terminal nodes in the tree should form a range. Second, we convert the string in each nonterminal node to a set of integers, where the set is a union of the sets represented by its children. Third, we can easily convert each set in the structure tree to a range.

For actions (containers or obligations) that have a list of items with no ordering issue, we can map each distinct string in actions (containers or obligations) to a distinct integer, and all the mapped integers in actions (containers or obligations) should form a range.

After the above two steps, we can convert the string on each attribute in a rule to a range. However, recall that if a request does not match any rule in the EPAL policy, the decision *default-ruling* is the decision of the request. Considering the example EPAL policy in Fig. 1, the decision *default-ruling* in line 14 is deny. To fulfill this goal, we add rule R_∞: $true \rightarrow dec$ as the default rule to make the sequence of range rules complete, where *dec* is the default-ruling of the EPAL policy. If no rule before R_∞ matches the request, the decision of the default rule is the decision of the request.

After EPAL policy numericalization, an EPAL policy becomes a sequence of range rules. Such representation is called the *sequential range rule representation*. The format of a range rule is $\langle predicate \rangle \rightarrow \langle decision \rangle$. A request has z attributes $F_1, \ldots, F_z$, where the domain of each attribute F_i, denoted $D(F_i)$, is a range of integers. The $\langle predicate \rangle$ defines a set of requests over the attributes F_1 through F_z. The $\langle decision \rangle$ defines the action (permit or deny) to take upon the requests that satisfy the predicate. The predicate of a rule is specified as $F_1 \in S_1 \wedge \cdots \wedge F_z \in S_z$ where each S_i is a range of integers and S_i is a subset of the domain of F_i. The semantics of a sequence of rules follows first-match, that is, the decision for a request is the decision of the first rule that the request matches. To serve as a security policy, a sequence of range rules must be complete, which means for any request, whose F_i value is in $D(F_i)$, there is at least one matching rule.

Example Taking the EPAL policy in Fig. 1 as an example, we convert two hierarchical elements, user categories and purposes to two structure trees as shown in Fig. 2. The normalization table of the EPAL vocabulary is shown in Fig. 3. The converted rules after mapping are shown in Fig. 4. Note that "*a*" and "*d*" in Fig. 4 denote *allow* and *deny*, respectively.

4.2 Uncertainly valued requests

Problem There are two differences between an EPAL policy and sequential range rules. First, the user category (data category or purpose) in an EPAL request may

Fig. 2 Two structure trees

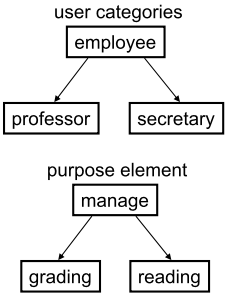


Fig. 3 Numericalization table for the EPAL policy in Fig. 1

user-category	professor: secretary: employee:	0 1 0,1
data-category	grades:	0
purpose	grading: reading: manage:	0 1 0,1
action	change: read:	0 1

Fig. 4 The numericalized rules for the EPAL policy in Fig. 1

$$R_1 : U \in [1,1] \wedge D \in [0,0] \wedge P \in [0,0] \wedge A \in [0,0] \rightarrow d$$
$$R_2 : U \in [0,1] \wedge D \in [0,0] \wedge P \in [0,1] \wedge A \in [0,0] \rightarrow a$$
$$R_3 : U \in [0,0] \wedge D \in [0,0] \wedge P \in [0,1] \wedge A \in [1,1] \rightarrow a$$
$$R_\infty : U \in [0,1] \wedge D \in [0,0] \wedge P \in [0,1] \wedge A \in [0,1] \rightarrow d$$

have multiple uncertain values. We call them uncertainly valued requests. For the example EPAL policy in Fig. 1, the user category in an EPAL request may be an "employee," which can be a "professor" or a "secretary." However, in the sequential range rule representation, rules are specified under the assumption that each attribute in a request has a singular value.

Second, the semantics of an EPAL policy and sequential range rules are different in checking whether a rule matches a request. In an EPAL policy, given a request with a specified category (user category, data category, or purpose), a *deny* rule can apply to its descendants or its ancestors in the structure tree of the category. More formally, given a *deny* rule $R_i : (F_1 \in S_1) \wedge \cdots \wedge (F_z \in S_z) \rightarrow deny$ and a request $Q : (F_1 \in S'_1) \wedge \cdots \wedge (F_z \in S'_z)$, the *deny* rule R_i can match the request Q if one of two following conditions satisfies.

1. R_i is a superset of Q if for any j $(1 \leq j \leq z)$ the condition $S_j \supseteq S'_j$ holds. In other words, *deny* rules can apply to the descendants of a specified category. For simplicity, we use $R_i \supseteq Q$ to denote that R_i is a superset of Q.
2. R_i is a subset of Q if for any j $(1 \leq j \leq z)$ the condition $S_j \subset S'_j$ holds. In other words, *deny* rules can apply to the ancestors of a specified category. For simplicity, we use $R_i \subset Q$ to denote that R_i is a subset of Q.

However, in the sequential range rules, a rule matches a request if and only if R_i is a superset of Q.

Solution To solve the first problem, we break an uncertainly valued request into multiple single-valued requests. For the example EPAL policy in Fig. 1, if a request is "an employee wants to change grades with the purpose grading," we break it into two single-valued requests: "a professor wants to change grades with the purpose grading" and "a secretary wants to change grades with the purpose grading."

To solve the second problem, for each decomposed single-valued request, our basic idea is to find all the rules that this single-valued request matches based on the semantics of sequential range rules, and then compute the decision of the original uncertainly valued request based on all these rules and their decisions. For simplicity, in the rest of this paper, the sentence "a request matches a rule" only means that a request matches a rule based on the semantics of sequential range rules. Let Q be an uncertainly valued request and $q_1, \ldots, q_k$ be the resulting single-valued requests decomposed from Q. For each q_x, we use $\mathcal{N}(q_x) = \{n_1^{q_x}, \ldots, n_{a_x}^{q_x}\}$ $(1 \leq x \leq k)$ to denote the set of sequence numbers of all rules that q_x matches. Note that we can easily sort sequence numbers in ascending order (i.e., $\forall 1 \leq i < j \leq a_x$, $n_i^{q_x} < n_j^{q_x}$). Because the last rule matches all the requests, there should be at least one common sequence number included by each set $\mathcal{N}(q_x)$ $(1 \leq x \leq k)$. Therefore, the minimum common sequence number included by each set $\mathcal{N}(q_x)$ $(1 \leq x \leq k)$ always exists. Let $n_{\min}$ denote the minimum common sequence number included by all these sets.

To find the decision of Q, first, we only consider the condition $R_i \supseteq Q$. Since $R_{n_{\min}}$ is the first rule that each q_x $(1 \leq x \leq k)$ matches, and hence Q matches, the decision of Q is the decision of $R_{n_{\min}}$. Next, we consider the condition $R_i \subset Q$. Because $R_{n_{\min}}$ is the first rule that Q matches and also a superset of Q, if we want to find a rule that is a subset of Q and is a *deny* rule, the sequence number of this *deny* rule should be less than $n_{\min}$. Therefore, as long as we can find a rule whose sequence number is less than $n_{\min}$ and whose decision is *deny*, the decision of Q is *deny*; otherwise, the decision of Q is the decision of $R_{n_{\min}}$.

Example Suppose an uncertainly valued request Q is "an employee wants to change grades with the purpose grading," and the EPAL policy to be evaluated is in Fig. 4. We first decompose this uncertainly valued request into two single-valued requests, q_1: "a professor wants to change grades with the purpose grading," and q_2: "a secretary wants to change grades with the purpose grading." Second, we compute the set of sequence numbers of all EPAL rules that q_1 or q_2 matches. $\mathcal{N}(q_1)$ is $\{2, \infty\}$ and $\mathcal{N}(q_2)$ is $\{1, 2, \infty\}$. The common minimum sequence number included by $\mathcal{N}(q_1)$ and $\mathcal{N}(q_2)$ is 2 and the sequence number that is less than 2 in $\mathcal{N}(q_1)$ and $\mathcal{N}(q_2)$ is 1. Because the decision of R_1 is *deny*, the decision of Q is *deny*.

4.3 Compound requests

Problem In the sequential range rules representation, each attribute in a request has a singular value. However, a EPAL request can be a compound authorization request, which contains a non-empty set of user-categories U, a nonempty set of data-categories D, a nonempty set of purposes P, and a nonempty set of actions A. For example, a person belongs to multiple user categories and he want to check whether there is one of his user categories that allows him to perform multiple actions for multiple purposes on multiple data categories [1].

Note that in the EPAL specification [1] the result of a compound request is a set of rules and the decision of each rule in this set is one decision of the compound request. The EPAL specification [1] does not define the procedure to compute the final decision based on this set, which gives the user flexibility to define their own strategy to compute the final decision.

Solution We solve this problem in two steps. First, we decompose a compound request into multiple single user-category requests. For example, "a person who is not only a professor of computer science, but also a professor of electrical engineering," we break it into two single user-category requests: "a professor of computer science" and "a professor of electrical engineering." Note that a single user-category request can be a single-valued request or an uncertainly valued request.

Second, to compute the final decision of the original compound request, our basic idea is to find the decision for each decomposed single user-category request and then compute the final set of ruleid-decision pairs based on the semantics of processing multiple user categories. A *ruleid-decision pair* consists of two parts, the sequence number of a rule and the corresponding decision. More formally, let CQ be a compound request with multiple user categories, and $Q_1, \ldots, Q_m$ be the resulting single user-category requests decomposed from CQ. Suppose dec_{Q_i} is the decision of Q_i and dec_{Q_i} comes from the rule $R_{n_{Q_i}}$, we first compute the set of ruleid-decision pairs of all single user-category requests decomposed from CQ, which is $RD = \{(n_{Q_1}, dec_{Q_1}), \ldots, (n_{Q_m}, dec_{Q_m})\}$. Second, find the first ruleid-decision pair with an *allow* decision and delete all the other ruleid-decision pairs with an *allow* decision in RD. Similarly, find the first ruleid-decision pair with a *deny* decision and delete all the other ruleid-decision pairs with an *deny* decision RD. Third, output RD as the result set for the compound request CQ.

Example Given a compound request CQ with two user categories, we decompose CQ into two single user-category requests Q_1 and Q_2. Suppose the ruleid-decision pair of Q_1 is $(2, allow)$ and the ruleid-decision pair of Q_2 is $(4, allow)$. Therefore, the set of ruleid-decision pairs RD is $\{(2, allow), (4, allow)\}$. Because the ruleid-decision pair $(4, allow)$ has the same decision of $(2, allow)$, we delete $(4, allow)$ from RD. The result set for the compound request CQ is $\{(2, allow)\}$.

4.4 Inefficient sequential searching

Problem After EPAL policy numericalization, the EPAL policy is converted to the sequence of range rules. To process an EPAL request, we need to conduct sequential searching, i.e., comparing the request against all the rules until finding the rule that matches the request, which is not efficient. To accelerate this operation, we want to convert the sequence of range rules to the policy decision diagram, which enables our evaluation engine to process an EPAL request without comparing the request against all the rules until finding the rule that matches the request. The key challenging issue in EPAL policy normalization is how to convert the sequence of range rules to an equivalent policy decision diagram.

Solution The policy decision diagram (similar to the firewall decision diagram in [14, 15]) is the core data structure in this conversion. A *Policy Decision Diagram* (PDD) with a decision set DS and over attributes $F_1, \ldots, F_z$ is an acyclic and directed graph that has the following five properties: (1) There is exactly one node that has no incoming edges. This node is called the *root*. The nodes that have no outgoing edges are called *terminal* nodes. (2) Each node v has a label, denoted $F(v)$. If v is a nonterminal node, then $F(v) \in \{F_1, \ldots, F_z\}$. If v is a terminal node, then $F(v) \in DS$. (3) Each edge $e{:}u \rightarrow v$ is labeled with a nonempty set of integers, denoted $I(e)$, where $I(e)$ is a subset of the domain of u's label (i.e., $I(e) \subseteq D(F(u))$). (4) A directed path from the root to a terminal node is called a *decision path*. No two nodes on a decision path have the same label. (5) The set of all outgoing edges of a node v, denoted $E(v)$, satisfies the following two conditions: (a) *consistency*: $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$; (b) *completeness*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$.

To correctly solve EPAL requests, the terminal node of the policy decision diagrams should have the capability to store multiple ruleid-decision pairs. Let RD denote the set of ruleid-decision pairs stored in the terminal node. Let $\langle R_1, \ldots, R_g \rangle$ denote the sequence of range rules after EPAL policy numericalization. We first change the decision of each rule R_i from $\langle decision \rangle$ to a ruleid-decision pair $(i, decision)$. A sequence of range rules $\langle R_1, \ldots, R_g \rangle$ and a PDD are equivalent if and only if the following two conditions hold. First, for each R_i denoted as $(F_1 \in S_1) \wedge \cdots \wedge (F_z \in S_z) \rightarrow RD$ and each decision path $\mathcal{P}$ denoted as $(F_1 \in S_1') \wedge \cdots \wedge (F_z \in S_z') \rightarrow RD'$, either R_i and $\mathcal{P}$ are non-overlapping (i.e., $\exists j (1 \leq j \leq z)$, $S_j \cap S_j{}' = \emptyset$) or $\mathcal{P}$ is a subset of R_i (i.e., $\forall j (1 \leq j \leq z)$, $S_j{}' \subseteq S_j$). In the second case, the ruleid-decision pair that consists of i (the sequence number of R_i) and R_i's decision should be included in the set RD that is stored in $\mathcal{P}$'s terminal node. Second, using $\mathcal{P}$ (or R_i) to denote the set of requests that match $\mathcal{P}$ (or R_i), the union of all the rules in $\langle R_1, \ldots, R_g \rangle$ is equal to the union of all the paths in the PDD. After converting an EPAL policy with n rules to a PDD, the processing time of a single-valued request decreases from $O(n)$ to $O(1)$. The pseudocode of the policy decision diagram conversion algorithm is in Algorithm 1. Note that in this paper we use $e.t$ to denote the node that e points to.

The EPAL policy decision diagram conversion algorithm is similar to the algorithm AllMatch2FirstMatch in XEngine [2]. The only difference between these two algorithms is the data structure for storing rules in the terminal nodes. In Eengine, such a data structure is ruleid-decision pairs, while in XEngine [2], such a data structure is origin blocks.

Example Figure 5 shows the PDD converted from the rule sequence $\langle R_1, R_2, R_3, R_\infty \rangle$ in Fig. 4.

5 The policy evaluation engine

After converting an EPAL policy to a semantically equivalent policy decision diagram, we need an efficient approach to search for the decision of a given request

Algorithm 1: PDDConversion ($\langle R_1, \ldots, R_g \rangle$)

Input: $\langle R_1, \ldots, R_g \rangle$.
Output: An equivalent policy decision diagram of range rules.

```
1  Build a decision path with root v from rule R_1, and add the rule-decision pair of
   R_1 to the terminal node of this path;
2  for i := 2 to g do Append(v, i, R_i);
3  return v;
4  Append(v, i, F_m ∈ S_m ∧ ··· ∧ F_z ∈ S_z → RD) /*F(v) = F_m,
   E(v) = {e_1, ..., e_k}*/
5  if (S_m − (I(e_1) ∪ ··· ∪ I(e_k))) ≠ ∅ then
6      add to v an outgoing edge e_{k+1} with label S_m − (I(e_1) ∪ ··· ∪ I(e_k));
7      build a decision path P from rule F_{m+1} ∈ S_{m+1} ∧ ··· ∧ F_z ∈ S_z → RD, and
       make e_{k+1} point to the first node of P;
8      add RD to the terminal node of P;
9  if m < z then
10     for j := 1 to k do
11         if I(e_j) ⊆ S_m then Append(e_j.t, i,
           F_{m+1} ∈ S_{m+1} ∧ ··· ∧ F_z ∈ S_z → RD);
12         else if I(e_j) ∩ S_m ≠ ∅ then
13             add to v an outgoing edge e with label I(e_j) ∩ S_m;
14             make a copy of the subgraph rooted at e_j.t, and make e points to the
               root of the copy; replace the label of e_j by I(e_j) − S_m;
15             Append(e.t, i, F_{m+1} ∈ S_{m+1} ∧ ··· ∧ F_z ∈ S_z → RD);
16 else if m = z then
17     for j := 1 to k do
18         if I(e_j) ⊆ S_m then add RD to terminal node e_j.t;
19         else if I(e_j) ∩ S_m ≠ ∅ then
20             add to v an outgoing edge with label I(e_j) ∩ S_m;
21             make a copy of the subgraph rooted at e_j.t, and make e points to the
               root of the copy; replace the label of e_j by I(e_j) − S_m;
22             and add RD to the terminal node e.t;
```

using PDD. The straightforward approach to processing requests is that using the PDD to find the decision, namely the *decision diagram approach*. The decision diagram approach consists of two steps. First, we numericalize the request using the same numericalization table in converting the EPAL policy to range rules. For example, a request (secretary, grades, grading, change) will be numericalized as a tuple of four integers (1, 0, 0, 0). Second, we search the decision for the numericalized re-

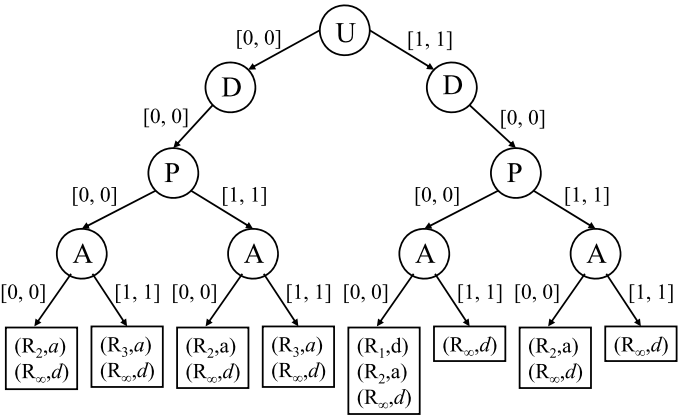
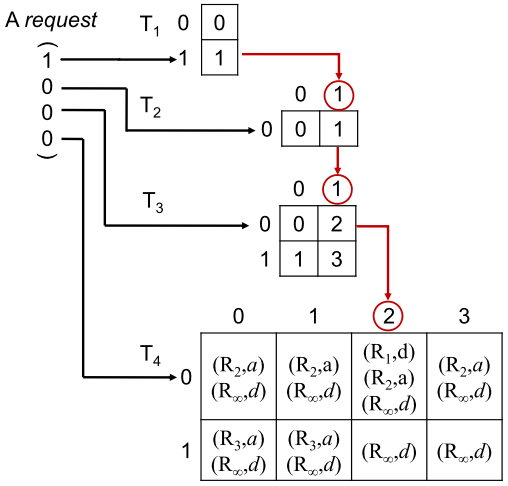


Fig. 5 PDD converted from range rules in Fig. 4

Fig. 6 Forwarding tables for PDD in Fig. 5



quest on the constructed PDD. Note that every terminal node in a PDD stores multiple ruleid-decision pairs.

To further facilitate the processing of a request, we use the *forwarding table approach* [2]. The forwarding table approach is based on the PDD that is constructed in the decision diagram approach. The basic idea of the forwarding table approach is to convert a PDD to z tables, which we call *forwarding tables*, such that we can search the decision for each single-valued request by traversing the forwarding tables in z steps. Obviously, the time complexity for processing a single-valued request is $O(z)$. Considering the same request (1, 0, 0, 0) in the above example, Fig. 6 shows the process using the forwarding table approach. We can find the correct decision for this request in the following four steps. First, we use 1 (value of secretary in the request) to find the value $T_1[1]$. Second, we use 0 (value of grades in the request) to find the value $T_2[0, T_1[1]]$. Third, we use 0 (value of grading in the request) to find the value

$T_3[0, T_2[0, T_1[1]]]$. Fourth, we use 0 (value of change in the request) to find the value $T_4[0, T_3[0, T_2[0, T_1[1]]]]$. This process continues until we find the value in T_4, which contains the ruleid-decision pair for the given request.

6 Correctness of EPAL normalization

The correctness of EPAL policy numericalization is obvious. The correctness of EPAL policy normalization follows from Lemmas 1, 2 and Theorems 3, 4, 5.

Lemma 1 *Given an EPAL policy E, for any single-valued request q, the set of ruleid-decision pairs in the terminal node of a path $\mathcal{P}$ that q matches includes all the rules that q matches in E.*

Proof Let E be $\langle R_1, \ldots, R_i \rangle$,where each R_i is a rule. We build a PDD such that for any decision path $\mathcal{P}$ in the PDD and any R_i, using $\mathcal{P}$ (or R_i) to denote the set of requests that match $\mathcal{P}$ (or R_i), if $\mathcal{P} \cap R_i \neq \emptyset$, then $\mathcal{P} \subseteq R_i$ and R_i's decision belongs to $\mathcal{P}$'s set of ruleid-pairs. For any request q, there exists at most one decision path in the partial PDD that q matches. Suppose there exists a decision path $\mathcal{P}$ that q matches. Thus, for any R_i that $q \in R_i$, then $R_i \cap \mathcal{P} \neq \emptyset$, which means that R_i's decision belongs to $\mathcal{P}$'s set of ruleid-pairs, $\mathcal{P}$'s RD contains all the rules that q matches in E. □

Lemma 2 *Given an EPAL policy E, for any single-valued request q, using $RD(q)$ to denote the set of ruleid-pairs that Q matches, the decision of the rule with the minimum sequence number in $RD(Q)$ is the same decision that E makes for q.*

Proof Let E be $\langle R_1, \ldots, R_i \rangle$,where each R_i is a rule. Lemma 2 is correct because the first-match semantics computes the decision of $RD(q)$ based on the first rule that q matches in E. □

Theorem 3 *Given an EPAL policy E and its normalized version E', for any single-valued request q, E and E' have the same decision for q.*

Proof The correctness of Theorem 3 follows from Lemma 2. □

Theorem 4 *Given an EPAL policy E and its normalized version E', for any uncertainly valued request Q, E and E' have the same decision for Q.*

Proof Let $q_1, \ldots, q_k$ be the resulting single-valued requests decomposed from Q. According to Lemma 1, the RD_i in the terminal node of a path $\mathcal{P}$ that q_i matches in E' consists of all the rules that q_i matches in E. Thus, $RD_1, \ldots, RD_k$ consists of all the rules in E that any q_x $(1 \leq x \leq k)$ matches. Therefore, the decision resolved by the method in Sect. 4.2 is the decision that E makes for Q. This is equivalent to computing all the rules that q_i matches in E. □

Theorem 5 *Given an EPAL policy E and its normalized version E', for any compound request Q, E, and E' have the same decision for Q.*

Proof Let $Q_1, \ldots, Q_m$ be the resulting single user-category requests decomposed from CQ. dec_{Q_i} is the decision of Q_i and dec_{Q_i} comes from the rule $R_{n_{Q_i}}$. According to Sect. 4.3, we compute the set $RD = \{(n_{Q_1}, dec_{Q_1}), \ldots, (n_{Q_m}, dec_{Q_m})\}$, find the first ruleid-decision pair with an *allow* decision and the first ruleid-decision pair with an *deny* decision, and finally delete all other ruleid-decision pairs in RD. This procedure follows the EPAL specification. Therefore, E and E' have the same decision for Q. □

7 Experimental results

We implemented Eengine using Java 1.6.3. Our experiments were carried out on a desktop PC running Windows Server 2003 with 8 GB of RAM and dual 2.56 GHz Dual Core AMD processors. We evaluated the efficiency and effectiveness of Eengine on synthetic EPAL policies.

In terms of efficiency, we measured the request processing time of Eengine in comparison with that of IBM PDP. For Eengine, the processing time for a request includes the time for numericalizing the request and the time for finding the decision for the numericalized request. For IBM PDP, the processing time for a request is the time for finding the decision. Note that we only considered single-valued and uncertainly valued requests in our experiments. Because in both our Eengine and IBM PDP, processing single-valued and uncertainly valued requests are two core operations for evaluating EPAL policies. Processing compound requests can be decomposed of processing multiple single-valued and/or uncertainly requests.

It is difficult to get real-life EPAL policies, as access control policies are often deemed confidential. Therefore, we generated random synthetic EPAL policies. The experimental results showed that Eengine is orders of magnitude more efficient than IBM PDP, and the performance difference between Eengine and IBM PDP grows almost linearly with the number of rules in EPAL policies. For EPAL policies of large sizes (with thousands of rules), the experimental results showed that Eengine is three to four orders of magnitude faster than IBM PDP for both single-valued and uncertainly valued requests.

We also measured the preprocessing time of EPAL policies for Eengine. The preprocessing time of an EPAL policy includes the time for numericalizing the policy, the time for normalizing the policy, and the time for building the internal data structure (of a PDD or forwarding tables). For synthetic EPAL policies (of large sizes with thousands of rules), the preprocessing takes a few seconds. For example, numericalizing and normalizing an EPAL policy of 4,000 rules takes about 0.35 second on average.

In terms of effectiveness, we compared the decisions made by Eengine and IBM PDP for each request. In our experiments, we first generated 10,000 random single-valued requests and 10,000 random uncertainly valued requests; and then fed each request to both Eengine and IBM PDP and compared their decisions.The experimental results showed that Eengine and IBM PDP have the same decision for every request.

Figure 7 shows the preprocessing time versus the number of rules in a three-layered policy for Eengine. Note that for all figures in this section, we use PDD

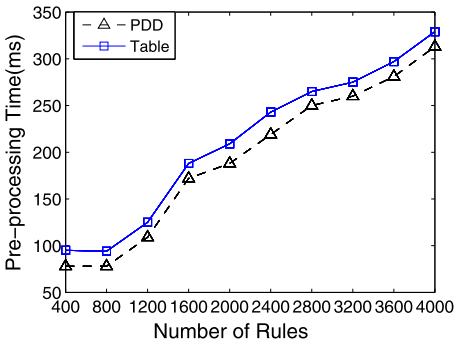


Fig. 7 Preprocessing time on synthetic EPAL policies

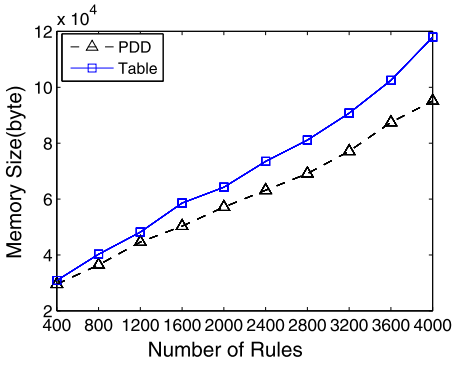


Fig. 8 Memory size on synthetic EPAL policies

to denote the decision diagram approach and use Table to denote the forwarding table approach. We observe that there is an almost linear correlation between the preprocessing time of Eengine and the number of rules, which demonstrates that Eengine is scalable in the preprocessing phrase. Figure 8 shows the memory size of Eengine versus the number of rules in a three-layered policy for Eengine. Similar to the preprocessing time of Eengine, there is an almost linear correlation between the memory size of Eengine and the number of rules.

Figure 9 shows the difference between IBM PDP and Eengine for the total processing time of 10,000 randomly generated single-valued requests. This figure shows that Eengine is orders of magnitude faster than IBM PDP and the performance difference grows almost linearly with the number of rules in EPAL policies.

Figures 10 and 11 show the experimental results as a function of the number of rules in a three-layered policy for processing single-valued requests and uncertainly valued requests, respectively. Figure 12 shows the results as a function of the number of layers for processing single-valued requests in three-layered policies, which consist of 2,000 rules. Figure 13 shows the evaluation results as a function of the number of layers for processing uncertainly valued requests in three-layered policies, which consist of 2,000 rules. Note that the vertical axes of these four figures are in logarithmic scales.

These figures demonstrate that Eengine is highly scalable and efficient in comparison with IBM PDP. For single-valued requests, under different numbers of rules, say

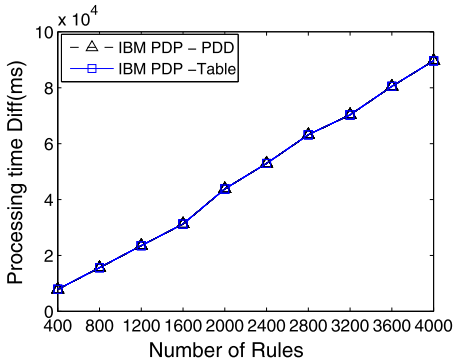


Fig. 9 Processing time difference between IBM PDP and Eengine

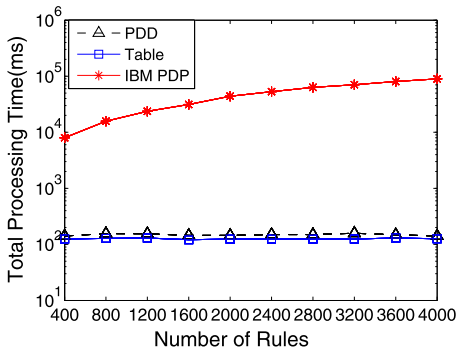


Fig. 10 Effect of number of rules for single-valued requests

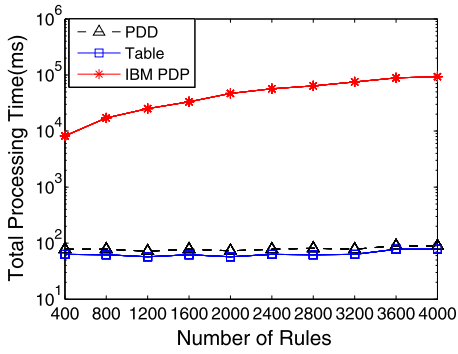


Fig. 11 Effect of number of rules for uncertainly valued requests

400, 2,000, and 4,000 rules in a three-layered policy, the forwarding table approach is 63, 354, 712 times faster than IBM PDP, respectively, and the PDD approach is 56, 303, 636 times faster than IBM PDP, respectively. For uncertainly valued requests, under different numbers of rules, say 400, 2,000, and 4,000 rules in a three-layered policy, the forwarding table approach is 128, 819, 1,195 times faster than IBM PDP, respectively, and the PDD approach is 104, 640, 1,047 times faster than IBM PDP, respectively. For EPAL policies with 2,000 rules but with various number of layers, for single-valued requests, say in a 1, 4, 8 layered policy, the forwarding table approach

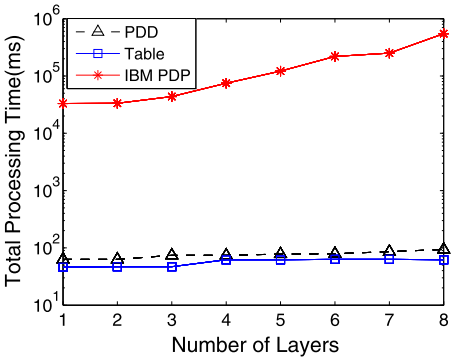


Fig. 12 Effect of number of layers for single-valued requests under 2,000 rules

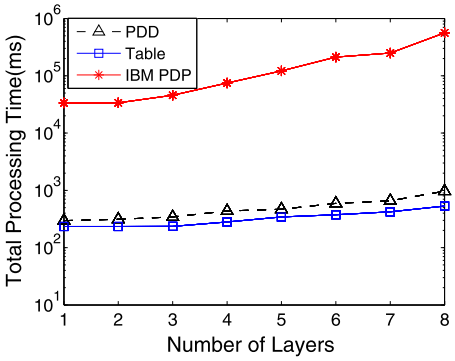


Fig. 13 Effect of number of layers for uncertainly valued requests under 2,000 rules

is 718, 1,204, 8,852 times faster than IBM PDP, respectively, and the PDD approach is 524, 1,009, 5,838 times faster than IBM PDP, respectively. For EPAL policies with 2,000 rules but with various number of layers, for uncertainly valued requests, say in a 1, 4, 8 layered policy, the forwarding table approach is 143, 266, 1,055 times faster than IBM PDP, respectively, and the PDD approach is 113, 170, 578 times faster than IBM PDP, respectively.

8 Conclusions

In this paper, we present Eengine, an EPAL policy evaluation engine. We make three major contributions. First, we propose algorithms that build efficient data structures for EPAL policy evaluation. Second, we propose optimization techniques for uncertainly valued requests and compound requests. Third, we conducted extensive experiments and demonstrated Eengine's efficiency and effectiveness on large-scale EPAL policies. Our experimental results show that Eengine is orders of magnitude faster than the standard IBM implementation.

Acknowledgements This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-0845513. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National

Science Foundation. The work of Zheng Qin is partially supported by the National Science Foundation of China under Grant No. 61070194, the Information Security Industrialization Fund from NDRC of China in 2009 (No. [2009]1886), the major achievements transfer projects of MOF and MIIT of China in 2010 (No. CJ [2010] 341), and NSF of Hunan Province, China (No. 09JJ3124).

References

1. IBM (2003) Enterprise Privacy Authorization Language (EPAL). http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/
2. Liu AX, Chen F, Hwang J, Xie T (2008) Xengine: a fast and scalable XACML policy evaluation engine. In: Proceedings of the ACM of international conference on measurement and modeling of computer systems, Sigmetrics, pp 265–276
3. Anderson AH (2006) A comparison of two privacy policy languages: EPAL and XACML. In: Proceedings of the 3rd ACM workshop on secure web services, pp 53–60
4. Borders K, Zhao X, Prakash A (2005) CPOL: high-performance policy evaluation. In: Proceedings of the ACM conference on computer and communications security, CCS, pp 147–157
5. Wei Q, Crampton J, Beznosov K, Ripeanu M (2008) Authorization recycling in RBAC systems. In: Proceedings of the ACM symposium on access control models and technologies, SACMAT
6. Crampton J, Leung W, Beznosov K (2006) The secondary and approximate authorization model and its application to Bell-Lapadula policies. In: Proceedings of the ACM symposium on access control models and technologies, SACMAT
7. Dong Q, Banerjee S, Wang J, Agrawal D, Shukla A (2006) Packet classifiers in ternary CAMs can be smaller. In: Proceedings of the ACM Sigmetrics, pp 311–322
8. Qiu L, Varghese G, Suri S (2001) Fast firewall implementations for software-based and hardware-based routers. In: Proceedings of the 9th international conference on network protocols, ICNP
9. Stufflebeam WH, Antón AI, He Q, Jain N (2004) Specifying privacy policies with P3P and EPAL: lessons learned. In: Proceedings of the ACM workshop on privacy in the electronic society, pp 35–36
10. Hung PCK, Ferrari E, Carminati B (2004) Towards standardized web services privacy technologies. In: Proceedings of the IEEE international conference on web services, pp 174–181
11. (2004) Unification in privacy policy evaluation—translating EPAL into prolog. In: Proceedings of the IEEE international workshop on policies for distributed systems and networks, pp 185–188
12. Barth A, Mitchell JC, Rosenstein J (2004) Conflict and combination in privacy policy languages. In: Proceedings of the ACM workshop on privacy in the electronic society, pp 45–46
13. Barth A, Mitchell JC (2005) Enterprise privacy promises and enforcement. In: Proceedings of the 2005 workshop on issues in the theory of security, pp 58–66
14. Gouda MG, Liu AX (2004) Firewall design: consistency, completeness and compactness. In: Proceedings of the IEEE international conference on distributed computing systems, ICDCS, pp 320–327
15. Gouda MG, Liu AX (2007) Structured firewall design. Comput Netw J 51(4):1106–1120