Efficient General Policy Decision by Using Mutable Variable Aware Cache

Liangxing Liu*, Weili Han*†‡, Elisa Bertino§, Tao Zhou*, and Xinyi Zhang*
* *Software School, Fudan University, Shanghai, China*
† *Key Lab of Information Network Security, Ministry of Public Security, Shanghai, China*
‡ *China Internet Network Information Center, Beijing, China*
§ *Department of Computer Science, Purdue University, West Lafayette, USA*

***Abstract*—Performance is a key issue in the implementation of tools for policy-based management of large and complex networked systems. When a system is characterized by millions of policies, the policy decision point is usually a performance bottleneck for the whole system. Although a few researchers have proposed cache-based methods to improve the efficiency of the policy decision point, the mutable variables, e.g., time, location, and temperature, are usually hard to be dealt with. The reason is that when applicable policies contain mutable variables, the policy decision point has to re-evaluate conditions in policies for events with the same targets. This paper thus proposes a novel Mutable Variable Aware Cache mechanism, by which the entries in the cache are aware of the mutable variables. The conditions or sub-conditions that do not contain mutable variables in the applicable policies will be evaluated once during the life cycle of a policy decision point for specific events, and the evaluation results will be appended to the cache. This optimization can greatly reduce the cost of fetching the values of mutable variables and evaluating the conditions or sub-conditions. It can, therefore, improve the performance of the policy decision point. We classify the possible situations and propose key algorithms. We also conduct a performance evaluation, which shows that the Mutable Variable Aware Cache mechanism can significantly improve the efficiency of a policy decision point.**

***Keywords*-Policy-based Management, Policy Decision Point, Cache, Mutable Variable**

I. Introduction

Policy-based management [1][2][3][4][5] represents an effective approach for managing large-scale networked systems, such as cloud systems. Such an approach simplifies network and security management and supports high-level languages by which management policies can be expressed in the Event-Condition-Action form. A very well-known architectural paradigm for implementing such a policy-based management approach, proposed by standardization bodies like IETF and DMTF [6], relies on three core architectural components: the policy repository (PR) in charge of managing policies; the policy decision point (PDP) in charge of processing policies in order to generate policy decisions; and policy enforcement point (PEP) in charge of interacting with the applications requiring the enforcement of policies.

Of course, in order to achieve reasonable performance in the management of the networked system, the performance of the PDP is critical as this component typically has to process several applicable policies for each single request. A system consisting of thousands of software and hardware components may have millions of policies and thus the PDP must be designed with efficiency as a main goal.

Approaches have thus been proposed to address the efficiency of the PDP [7][8][9]. Such approaches usually leverage some cache mechanisms [7] or re-organize the policies to streamline the policy evaluation process at the PDP [8][9]. Cache-based approaches usually leverage a temporary storage to store the intermediate results for the PDP. For example, the system stores the decision result in the cache. After that, if the same event happens, the system can directly return the result fetched from the cache without further evaluation in the PDP. The latter could heavily consume time. Here, the same event means the same type of event, the same triggered object, and the same possible target.

In cache-based approaches, however, a critical issue is represented by the management of *mutable variables* in the monitored events. Examples of such variables include the *time* when the event happens, and the *location* where the event happens. Mutable variables are however critical for many policy models such as models supporting context sensitive access control model and attribute-based policy model. The main issue concerning these variables is that their values need to be continuously acquired, possibly for each processed event. A possible approach to addressing this issue is to use a time window for each cache entry. For any event happening in the time window, the decision of the PDP will be fetched from the cache upon cache hit, and returned despite the fact that the result could be incorrect due to changes of values in mutable variables. This strategy is thus not optimal and whereas it may be suitable for managing authorization policies [10], it may be not applicable to other domains.

This paper, thus, proposes a novel optimization technique, referred to as the **Mutable Variable Aware Cache (MVAC)** mechanism, for improving the efficiency of cache-based mechanisms for PDP. Under the MVAC approach the variables are divided into mutable variables and non-mutable variables. MVAC manages variables in these two categories according to different strategies. When non-mutable variables are involved in the evaluation process of the PDP, a result cache entry will be created. Otherwise, when mutable

variables are involved in the evaluation process of the PDP, the relevant conditions will be attached to the mutable variable aware cache entry, and evaluated again only when the same event happens. This strategy avoids having to always perform a full evaluation process at the PDP and thus enhances the PDP performance.

The main contributions of this paper are as follows:

- We propose a novel cache mechanism, referred to as the **Mutable Variable Aware Cache (MVAC)** mechanism, to enhance the efficiency of PDP. In MVAC, we leverage two types of cache entries: result cache in charge of the applicable policies without mutable variables and token cache in charge of the applicable policies with mutable variables. We use different strategies to deal with these two different caches when processing policies for policy decision making.
- We evaluate the effectiveness and efficiency of our proposed technique by using simulated data concerning a data center case. We compare MVAC with other PDP evaluation processes in the cases of pure policy cache and no cache. The performance evaluation shows that the proposed caching mechanism greatly improves performance. Experimental results also show how the ratio of mutable variables and size of cache pool influence cache performance.

The rest of this paper is organized as follows: Section II introduces preliminary notions and a running scenario of our proposed work. Section III describes our mutable variable cache mechanism in detail. We then present experimental setup and show experimental results in Section IV. Next, we discuss implementation issues in Drools [11][12], which is a rule reasoning engine that we have used as a policy evaluation mechanism in PDP. We also discuss about the correctness of the MVAC-based decisions in Section V. Section VI introduces related work. Finally, Section VII summarizes the paper and outlines future work.

II. Preliminary Notions and Running Scenario

A. Policy Languages

Most policy-based management approaches support a policy language for expressing policies of interest [13][14][15]. To date, there are two main categories of policy languages that have been defined for two different application purposes: security management policies, such as XACML (eXtendible Access Control Markup Language) [16] policies, and network management policies, such as SPL (Simple Policy Language) policies.

The policy languages follow the Event-Condition-Action (ECA) paradigm. The ECA paradigm means that if an *event* happens, the *action* will be executed when the *condition* is satisfied. However, some policy languages are restricted to the Condition-Action (CA) form. The CA form means that when the *condition* is satisfied, the *action* will be executed. The *action* could simply be returning a *permit* or *deny* decision. In this paper, we focus on the general form of policies, which follows the ECA paradigm.

The policies in policy-based management are designed to fulfill a specific purpose, and are usually organized in a hierarchical structure. Furthermore, some elements, such as *target* and *combination algorithm*, are designed to enhance the performance of the policy evaluation process.

B. Cache based Performance Optimization

The use of cache is a common strategy to enhance performance. There are two main types of caches in the PDP. The first one is the policy cache. When an event happens, the PDP captures the event, then retrieves the applicable policies. Identifiers of these policies would then be set into policy cache entries. Next time the same event comes, the PDP acquires the applicable policies through cache entries with no need to retrieve the policies from the PR. The second type of cache is the result cache. That is, when the event and the target are the same, the result can be directly returned from the cache without further policy evaluations. This mechanism can dramatically reduce the cost [7][17]. These cache mechanisms usually ignore the problem of policy updates, because the policies in the policy-based management are usually relevantly stable. A feasible way to count the policy update is to refresh all the cache entries.

However, when the condition in a policy contains a term determined by a mutable variable, such as $currenttemperature > 25°\mathrm{C}$, the result of the hit cache entry could be wrong because the value of *currenttemperature* is changed, for instance, from $24°\mathrm{C}$ to $26°\mathrm{C}$. In order to deal with such changes, the PDP should re-evaluate all applicable policies, or set a time window during which any change of mutable variables is ignored which may result in a wrong decision.

To combine the advantages of the two types of cache, we propose the **Mutable Variable Aware Cache** mechanism.

C. Running Scenario: Data Center Management

Our running scenario focuses on data centers which are today commonly used by many different organizations and companies. Hoelzle *et al.* [18] defined a data center as a building in which multiple servers and communication gears are co-located because of their common environmental requirements and physical security needs, and for the ease of maintenance. In order to make a data center run smoothly and stably, an administrator is faced with several tough challenges, including environmental control and safety monitoring. A well-designed environmental monitor and control system typically include basic equipments, such as temperature sensors, humidity sensors, smoke sensors, air-conditioners and fire alarms. All such equipments need to be properly managed. A policy-based management approach represents

an effective approach in such context. For example, the central control system could make certain adjustments to environmental conditions according to policies stored in the PR based on data acquired by monitor equipments. Moreover if the administrator wants to change policies, he or she just need to modify the relevant policies which is very easy to do when a high-level policy language is provided.

The following are examples of policies in our data center scenario.

Policy 1:

Event: PersonArrive
Condition:
Person is Administrator
Action: Open the door of Room A

Policy 2:

Event: TemperatureChange
Condition:
Temperature is higher than 24°C
∧ Location of Sensor A is in Room A
Action: Turn Cooler A in Room A on

Policy 3:

Event: TimeChange
Condition:
Current time is 12:00 AM
Action: Turn the ventilating fan in Room A on

Policy 1, *Policy 2* and *Policy 3* are policies from a policy set designed for the management of Room A in the data center:

- *Policy 1* specifies that when a person arrives at the door of Room A, the physical access control system checks whether the person is the administrator. If this is the case, the control system opens the door; otherwise the door is kept locked. To enforce this policy, the PDP must retrieve the role properties of the subject, and take a physical access control decision, that is, opening the door and allowing access to the room, or keeping the door locked and denying access to the room.

  ***Analysis**: The result cache can work well for this policy. That is, the PDP does not need to retrieve the role properties when the person arrives at the door again, and can directly return the correct result from the result cache.*

- *Policy 2* specifies that when the current temperature is higher than 24°C and the location of Sensor A is in Room A, the central control system turns on the cooler in Room A.

  ***Analysis**: The result cache can not work well for this policy. Once the temperature changes from 25°C to 23°C, the decision result should be changed accordingly. Hence, this part of the condition must be evaluated for each event, because the temperature is a mutable variable. On the contrary, the location of Sensor A is not a mutable variable, if the sensor is permanently installed in Room A. In such case, information about the location of Sensor A just needs to be retrieved once, and thus the latter part of the condition just needs to be evaluated once. However, if the sensor is mobile, the location is obviously a mutable variable. Thus, whether a variable is mutable or not should be specified by the administrator.*

- *Policy 3* specifies that a ventilating fan must be turned on when the current time is 12:00 AM every day.

  ***Analysis**: The result cache does not work well for this policy because the current time is a mutable variable.*

III. Mutable Variable Aware Cache Mechanism

A. Key Flow Of Mutable Variable Aware Cache Mechanism

The IETF policy working group has standardized a policy framework [19], which consists of four elements: the policy administration point (PAP), the policy decision point (PDP), the policy enforcement point (PEP), and the policy repository (PR). The PAP allows administrators to write and manage policies; the PR supports storage and retrieval of the defined policies; The PDP evaluates policies and make decisions; whereas the PEP enforces the decisions.

Figure 1 shows the flow of our proposed MVAC in the context of the IETF policy framework. Our MVAC flows are represented by solid lines. After an *event* is captured, a *request* for *decision* is sent to the PDP. The PDP tries to hit a cache entry. If a *ResultCache* is hit, a decision is directly made. If a *TokenCache* is hit, the PDP further evaluates the corresponding policies to make a decision. However, if no cache entry is hit, the PDP retrieves the applicable policies from the PR, and then evaluates these policies to make a decision. After a decision is made, a *token* set may be generated. If a *token* set is generated, a *TokenCache* will be created. When no token set is created, the PDP creates a *ResultCache* cache entry. An administrator can modify policies through the PAP, and thus invalidate some cache entries.

B. Basic Definitions

We first introduce the new concept of *Mutable Variable Aware Token* (MVAT for short), which is a data object recording whether the variables in the *Condition* part of a specific ECA policy are mutable variables, and the values of these variables when the policy applies to a policy request.

Definition 1. Mutable Variable Aware Token(MVAT):

$$MVAT := \langle TokenID,\ V,\ PolicyID \rangle$$
$$V := \{\langle variable\_name, is\_mutable, variable\_value \rangle^{+}\}$$

Here, *TokenID* refers to the identifier of a MVAT. $\langle variable\_name, is\_mutable, variable\_value\rangle$ is a triple where $variable\_name$ refers to the name of a variable, $is\_mutable$ denotes whether a variable is defined as a mutable variable, and $is\_mutable \in \{true, false\}$. *True* means that the variable is a mutable variable. $variable\_value$ refers to the value of the variable when the MVAT is created. The sign + indicates that there may be more than one triple of variables. *PolicyID* refers to a policy identifier, indicating from which policy the MVAT comes. In this case, we say this MVAT is *pointing to* the policy with identifier *PolicyID*. We will discuss how a MVAT is created in detail later.

The following is an example of MVAT.

Example:

$$\langle tid : token_1, \{\langle temperature, true, 24\rangle, \langle location, false, RoomA\rangle\}, policy_2\rangle$$

We extend the IETF policy framework by adding a *Cache* object to enhance the policy evaluation performance. We define two kinds of cache objects, namely, **ResultCache**, and **TokenCache**. We define them as follows:

Definition 2. ResultCache:

$$ResultCache := \langle Event, Decision, ExpireTime, \{PolicyID^+\}\rangle$$

Here, a result cache is keyed by *Event*, which also serves as an index for the PDP when looking for a cache entry. *Decision* is a decision made by the PDP, which is the *Action* part of an *ECA* policy, or a combination of *Actions* if the number of applicable policies is greater than one. *ExpireTime* refers to the window time when the cache entry will expire. *PolicyID* refers to policy identifier and the sign + indicates there may be more than one policy.

Definition 3. TokenCache:

$$TokenCache := \langle Event, \{Token^+\}, ExpireTime\rangle$$

Here, like in Definition 2, *Event* and *ExpireTime* refer to the key of a cache entry and expiration time, respectively. $Token^+$ is a set of tokens. The sign $+$ denotes there exists one or more tokens in a cache entry.

C. Five Situations of MVAC Creation in PDP

As we can see in Figure 1, the enhanced PDP leverages the cache module to improve its efficiency. Two kinds of cache entry may be generated, namely, a *ResultCache* entry and a *TokenCache* entry.

During the cache entry generation process, five possible cases can arise which need to be considered separately. Before we discuss these cases, we introduce some notations and settings:

- $N_{applicable+}$ denotes the number of applicable policies for an event whose conditions are evaluated true. Note that, $N_{applicable+} \geq 0$.

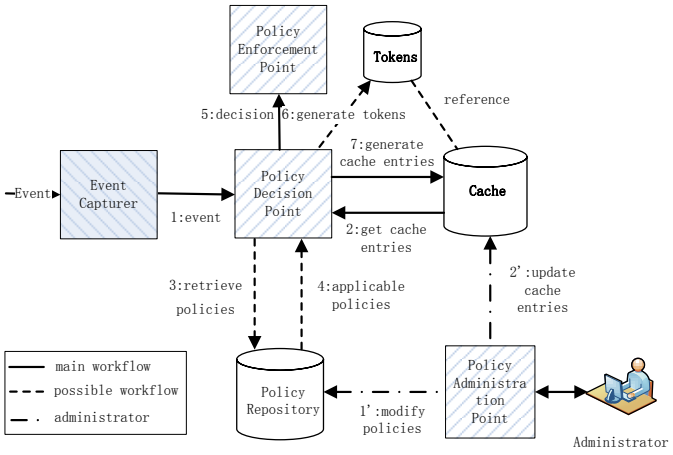

Figure 1. Key Flow Of Mutable Variable Aware Cache Mechanism

- M_i indicates whether an applicable policy ($1 \leq i \leq N_{applicable+}$) which is evaluated true contains mutable variables. If it does contain mutable variables, M_i=1; otherwise M_i=0.
- We extend existing policy languages by adding an *annotation* tag. Its value denotes whether the variables in the *Condition* of a policy are mutable variables. A system administrator can set or modify the *annotation* tag.

Assume that our example policies (see Section II.C) are expressed in a XML-based policy language. The annotations for each example are as follows:

- *Policy 1*

```
<annotations>
    <annotation type="non-mutable">
       person
    </annotation>
</annotations>
```

Note that in this policy, *person* is not defined as a mutable variable.

- *Policy 2*

```
<annotations>
    <annotation type="mutable">
       temperature
    </annotation>
    <annotation type="non-mutable">
       location
    </annotation>
</annotations>
```

- *Policy 3*

```
<annotations>
    <annotation type="mutable">
```

```
            currenttime
        </annotation>
    </annotations>
```

Figure 2 summarizes the five cases that can arise when creating cache entries in MVAC. In Figure 2, RC and TC represent $ResultCache$ and $TokenCache$ cache entries respectively. Two parameters determine the five cases: $N_{applicable+}$ and M_i. We do not show cases where $N_{applicable+}$=0 in Figure 2.

1) Case One: $N_{applicable+} = 1$, $M_1 = 0$. There exists only one applicable policy whose condition is evaluated true, and the policy does not contain any mutable variable.

- *Cache Entry Generation:* This step generates a *ResultCache* entry, whose definition is shown in Section III. Here we assume that *ExpireTime* is constant value *D*, which means an administrator can set the expiration time according to system requirements.
- *Example:* We show the *ResultCache* from Policy 1 as follows:

$$\langle PersonArrive, open\ \ door, D, \{policy_1\}\rangle$$

2) Case Two: $N_{applicable+} > 1$, $\forall M_i = 0$. There exists more than one applicable policy whose condition is evaluated true, and all of these policies contain no mutable variables.

- *Cache Entry Generation:* A *ResultCache* entry is generated. In the result cache entry, *Decision* is the combination of *Action* parts of these policies. The *PolicyID* set is a set of the identifiers of these policies.

3) Case Three: $N_{applicable+} = 1$, $M_1 = 1$. In this case, the number of applicable policies whose condition is evaluated true is one, and the policy does contain mutable variables.

- *Cache Entry Generation:* A *TokenCache* entry would be generated in this case, and its structure is as shown in Section III.
- *Example:* Policy 2 is the only applicable policy which is evaluated true, so a token would be created first:

$$\langle tid : token_1, \{\langle temperature, true, 24\rangle, \langle location, false, RoomA\rangle\}, policy_2\rangle$$

Then a token cache entry is generated:

$$\langle TemperatureChange, \{tid : token_1\}, D\rangle$$

4) Case Four: $N_{applicable+} > 1$, $\forall M_i = 1$. In this case, the number of applicable policies whose condition is evaluated true is more than one, and all these policies contain mutable variables.

- *Cache Entry Generation:* A *TokenCache* would be generated in this case. The token set is composed of tokens pointing to these policies.

5) Case Five: $N_{applicable+} > 1$, $\exists M_i = 0$, $\exists M_i = 1$. The number of applicable policies whose conditions are evaluated true is more than one, and some of these policies contain non-mutable variables, whereas others do not.

- *Cache Entry Generation:* Both *ResultCache* and *TokenCache* would be generated in this case. *ResultCache* and *TokenCache* are generated in the same way as when generated separately.

D. Decisions based on *ResultCache* and *TokenCache*

Upon arrival on an event, the PDP uses the event as a key to look up the cache entry keyed by exactly the same event. If a cache entry is hit, our cache mechanism deals with *ResultCache* and *TokenCache* in different ways in decision making.

1) ResultCache:

- *Decision Making:* If a result cache entry is hit, the PDP considers *Decision* in *ResultCache* as the appropriate decision. In this case, a decision is made directly without evaluating the variables.
- *Example:* Consider the following result cache entry:

$$\langle PersonArrive, open\ \ door, D, \{policy_1\}\rangle$$

The next time the *PersonArrive* event happens, the PDP first directly looks for a corresponding cache entry, and since the result cache is hit, the *open door* decision is taken.

2) TokenCache:

- *Decision Making:* If a token cache entry is hit, the PDP first acquires the value of *PolicyID* in the token contained in the token cache entry. Then the PDP evaluates these policies using the *Drools* reasoning engine. In this process, the PDP evaluate the non-mutable variables without acquiring their current values. By contrast, the PDP needs to acquire the current values of the mutable variables before evaluating them. Finally, a decision is made by the PDP according to evaluation results returned by Drools.
- *Example:* Suppose that the following token cache entry is hit:

$$\langle TemperatureChange, \{tid : token_1\}, D\rangle$$

The PDP then finds the token with the corresponding identifier by taking into account that for $policy_2$ only the *temperature* variable is mutable. Consequently, the PDP needs to acquire the current temperature value and evaluates $policy_2$ using Drools. During this process, there is no need for PDP to obtain the current value of the *location* variable. At last, a decision is made according to the current value of the temperature.

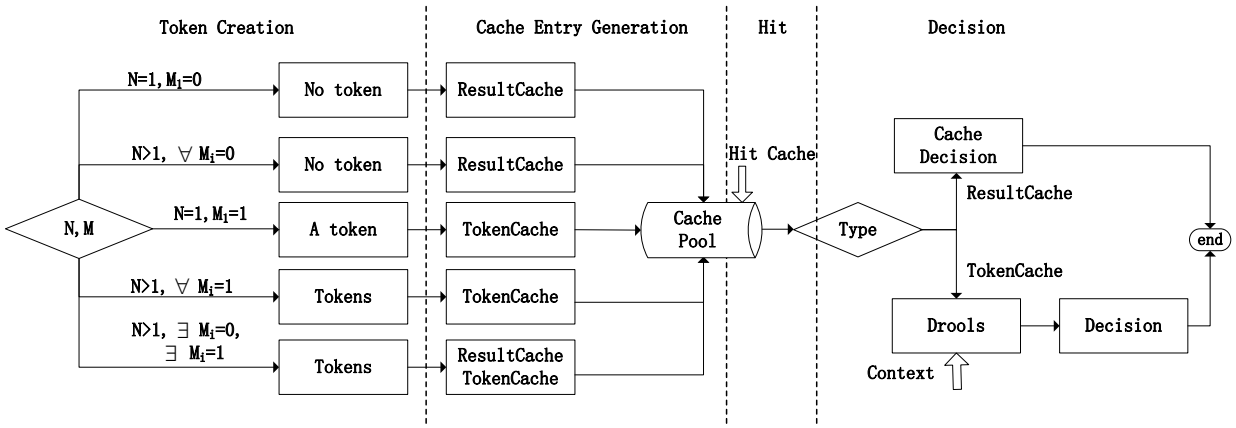

Figure 2. The Five Cases for the MVAC Creation in the PDP

E. Key Algorithms

In this section, we introduce two key algorithms for MVAC: a token creation algorithm, and a cache update algorithm.

1) Token Creation Algorithm: The pseudocode of token creation algorithm is Algorithm 1.

Algorithm 1 Token Creation.

Input: A: an annotation set of return value of GetAnnotation() with *m* elements; P, a set of *n* policies; V: an empty set of variables; C: context

Output: T: a set of generated tokens

1: $T = \emptyset$
2: **for** i=1 to n **do**
3: $V = \emptyset$
4: A:=$GetAnnotation(p_i)$
5: **for** j=1 to m **do**
6: v_j:=$A.get(j)$
7: **if** $v_j.is\_mutable = mutable$ **then**
8: $current\_value \leftarrow C$
9: v:=$\langle\, v_j.variable\_name, true, current\_value\rangle$
10: **else**
11: $current\_value \leftarrow C$
12: v:=$\langle\, v_j.variable\_name, false, current\_value\rangle$
13: **end if**
14: Add v to V
15: $t := \langle tid : token_i, V, p_i\rangle$
16: **end for**
17: Add t to T
18: **end for**
19: **return** T

First, after the associated states or parameters have been obtained and evaluated true against the *condition*, the PDP examines the applicable policies to determine the element values of a MVAT, e.g., $variable\_name$, $is\_mutable$ according to the policy annotation tag, and then generates a MVAT according to such values.

Policies, eg., XACML [16], are presented in a hierarchical and nesting structure. The root of each such policy is usually a *policy* or a *policy set*. A *policy set* is composed of a sequence of *policies* or other *policy set*. For those *policy set* nodes or *policy* nodes without an *annotation* tag, we follow the following strategy to assign an *annotation* value to them.

> *If a node does not have an annotation tag, we obtain its annotation value from its parent recursively.*

The above strategy is applied in order to obtain the annotation value of each specific policy. In Algorithm 1, the process for obtaining the annotation value is denoted by method $GetAnnotation()$. The return value of $GetAnnotation()$ is a set of pairs (Referred to as A) whose format is $\langle variable\_name, is\_mutable\rangle$. We assume that there are m elements in A, and that every element in A is denoted as v_j. C holds the current value of variables. We denote the identifier of these policies as a set P, and there are n policies in P, with each policy denoted as p_i.

After a token set is generated, a TokenCache cache entry is created afterwards using this token set.

2) Cache Update Algorithm: Algorithm 2 shows how we update cache entries.

An administrator may modify policies. Cache entries are supposed to be updated at the same time. Based on when cache entries are updated, different strategies are applied:

- This strategy is applied to the *TokenCache*. Every time changes are made to the *annotation* tags of the *policy set* nodes and *policy* nodes, the *TokenCache* entries with these tokens are invalidated.
- This strategy is applied to the *ResultCache*. Every time changes are made to the *Action* part of a policy,

Algorithm 2 Cache Update

Input: P: a set of modified policy identifiers; RC: ResultCache; TC: TokenCache

Output: RC: updated ResultCache; TC: updated TokenCache

```
1: RC, TC
2: for all p ∈ P do
3:     if isAction(p) then
4:         for all rc ∈ RC do
5:             if p ∈ rc.PolicyID then
6:                 invalidate(rc)
7:             end if
8:         end for
9:     else if isAnnotation(p) then
10:        for all tc ∈ TC do
11:            if p ∈ tc.Token.PolicyID then
12:                invalidate(tc)
13:            end if
14:        end for
15:    else if isDelete(p) then
16:        for all rc ∈ RC do
17:            if p ∈ rc.PolicyID then
18:                invalidate(rc)
19:            end if
20:        end for
21:        for all tc ∈ TC do
22:            if p ∈ tc.Token.PolicyID then
23:                invalidate(tc)
24:            end if
25:        end for
26:    end if
27: end for
28: return RC, TC
```

the *ResultCache* entries whose *PolicyID* set contains identifiers of these policies are invalidated.

- This strategy is applied to both the *TokenCache* and the *ResultCache*. Every time a *policy set* node or a *policy* node are deleted from the policy repository, some necessary *TokenCache* entries or *ResultCache* entries need to be invalidated.

In updating the *TokenCache* entries, we need to determine which entries have to be invalidated according to the tokens they contain. We take the following strategies:

- An *annotation* tag of a *policy set* node is modified. The *TokenCache* entries which contain tokens pointing to the *policy* nodes without an explicit *annotation* tag and that are descendants of the *policy set* node are updated.
- An *annotation* tag of a *policy* node is modified. *TokenCache* entries which contain tokens pointing to the *policy* node are updated.

In updating the *ResultCache* entries, as we mentioned earlier, we invalidate the *ResultCache* entries whose *PolicyID* set contains identifiers of the modified policies.

As discussed above, there are three cases in which we have to update the cache entries: the *Action* of policies has been changed, the *Annotation* has been changed, or policies have been deleted. We denote three cases as $isAction$, $isAnnotation$, $isDelete$, respectively. R is an identifier set of policies which have been modified by the administrator. The pseudocode of the cache update algorithm is Algorithm 2.

IV. Evaluation

A. Experimental Setup

We implemented a PDP extended with our MVAC mechanism. Our experimental platform consists of a PC running Windows 7 professional with 4.00 GB memory and Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz processor. We also employed Drools version 5.1.1. as our reasoning engine.

There are many factors affecting the performance of our PDP, such as *the number of policies* in the policy repository, *the ratio of mutable variables* appearing in policies, *the size of cache pool* and *cache entry replacement strategy*. In our policy evaluation experiments, we focus on how *the number of policies*, *the ratio of mutable variables* and *the size of cache pool* affect cache performance. The policies are designed to implement self-management of a data center. These policies are written in XML.

B. Performance Evaluation

In order to evaluate performance of MVAC, we conducted three experiments. The first experiment is to compare the policy evaluation times among different caches. The second experiment is to show how the ratio of mutable variables in policies affects the cache hit rate. The third experiment is to show how the size of cache pool affects cache performance. The detailed experimental processes are as follows.

First, we conducted experiments with a number of policies varying from 500 to 5,000. Every 10 seconds, a simulation of events, eg., *TemperatureChange*, *SettingTime* is captured by the PDP. We measured the time when an event is captured and the time when a decision is made by the PDP. The latter subtracted by the former was considered as the policy evaluation time. The experimental results are shown in Figure 3. We can see that, with the increase in the number of policies, the improvement of the efficiency of *TokenCache* is more pronounced. And *ResultCache* can always ensure a very efficient decision. Our PDP can achieve high performance through a combination of *TokenCache* and *ResultCache*.

Second, we conducted experiments with a ratio of mutable variables in policies varying from 0 to 1, and a total number of policies between 1,000 and 3,000 in two different experiments. Similarly, a simulation event is captured every 10 seconds. We count the *TokenCache* and *ResultCache* hit

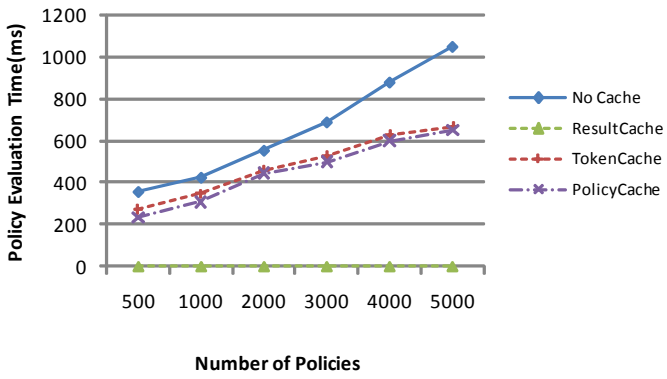

Figure 3. Comparison of Policy Evaluation Time among Different Types of Caches. *TokenCache* of MVAC is greatly more efficient comparing with when their is no cache.

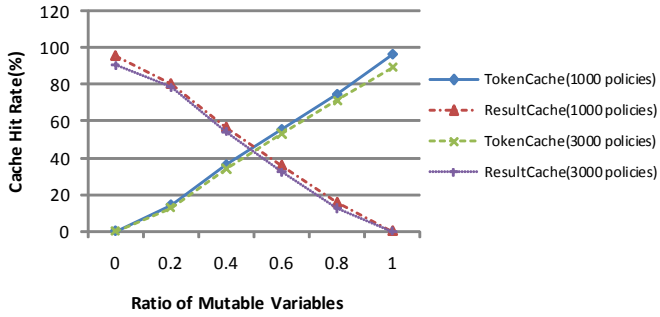

Figure 4. Effect of Ratio of Mutable Variables on Cache Hit Rate under 1,000 and 3,000 Policies. The ratio of mutable variables affects *TokenCache* and *ResultCache* hit rate, and the higher the ratio of mutable variables is, the higher the *TokenCache* hit rate is.

rate by recording whether a decision is made by means of cache. The results are shown in Figure 4. Our cache mechanism can ensure a total cache hit rate higher than 95%. With the increase of the ratio of mutable variables, *TokenCache* reaches a higher cache rate.

Third, we conducted experiments to show how the *size of the cache pool* affects cache performance. We take the maximum number of cache entries as size of the *TokenCache* and *ResultCache*. The number of policies we used in these experiments is 20. The size of the cache varies from 100 to 500 cache entries, and in addition we use two different values for ratio of mutable variables, that is, 0.2 and 0.4. The results are shown in Figure 5. In our experiments, as there are only 20 policies, the number of possible cache entries is not so large. In a cache pool with 100 cache entries, the total cache hit rate is higher than 82.6% when the ratio of mutable variables is 0.2. As the size of cache pool grows larger, cache hit rate will achieve an improvement.

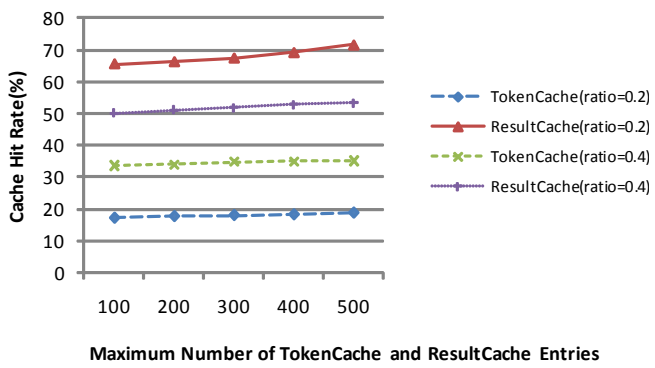

Figure 5. Effect of Maximum Number of Cache Entries on Cache Hit Rate with Ratio of Mutable Variables being 0.2 and 0.4. As the size of cache pool grows larger, cache hit rate achieves an improvement.

V. Discussion

A. Implementation Issues in Drools

Drools is a rule-based reasoning engine running on the Java platform. In our implemented PDP, we use Drools to evaluate conditions in policies. As policies in Drools are not organized in a nested structure, in order to use Drools for policy evaluation, we had to address the following issues:

- Transforming nested general policies into a Drools rule file, which is a *.drl* file. Rules in a Drools rule file are in the *when...then...* form.
- Setting environmental context as facts in Drools. Because Drools uses facts to evaluate conditions, we need to set environmental context, eg., current temperature as facts in Drools.
- Getting decisions of Drools as results of policy evaluation. We need to get the *then* part of the rules as decision results.

B. Correctness of MVAC-based Decision

The core idea of our MVAC mechanism is to ignore changes of certain variables. As a result, wrong decisions may be made in certain cases. Here are the possible cases in which the PDP may return the wrong decision.

- During the life cycle of the PDP, a variable which is defined as non-mutable has a different state from the state when a *TokenCache* is produced.
- An event happens before cache updates have taken effect.

In the former case, whether wrong decisions are made depends on the probability that a non-mutable variable changes its state. As a result, the probability of making a wrong decision depends on the ratio of non-mutable variables in policies. In other words, the higher the ratio of non-mutable variables is, the higher the probability that a

wrong decision is made would be. In the latter case, whether wrong decisions are made depends on the efficiency of cache update algorithm. An efficient cache update mechanism could decrease the probability to a great extent.

In order to reduce as much as possible the probability of making a wrong decision, when a system administrator writes policies, he or she should carefully decide whether a variable must be defined as a non-mutable variable. Therefore correct variable classification is crucial. In addition, the adoption of an efficient cache update mechanism can further decrease the probability that the PDP returns a wrong decision.

VI. Related Work

Although the performance of the PDP is a crucial aspect in policy-based management approaches, most previous researches on policies mainly paid attention to correctness issues [20], such as specification, design, and analysis of policies. Although policy correctness does play a very important role, the adoption of a policy-based management approach may not be effective if systems are not efficiently implemented [8].

There are two categories of approaches for enhancing policy evaluation performance. Approaches in the first category transform policies into equivalent expressions that are amenable to efficient evaluation [8][9]. Approaches in the second category use a cache mechanism [7][17]. Approaches in the first categories can be further classified into two types: approaches that transform policies into equivalent expressions with a simpler logical structure [8]; approaches that transform policies to other formats for processing by on-the-shelf tools for performing various logical reasoning and analysis [9].

Kevin *et al.* [7] show why performance has a major impact on policy-based management by experimentally analyzing the use of a policy-based management approach in the context of some emerging applications, such as real-time enforcement of privacy policies in a sensor network or location-aware computing environment. These applications require high throughput. However, when they conduct the experiments, they find that current policy enforcement solutions are unable to achieve the required scalability. As a result, they develop a flexible C++ framework, CPOL, to achieve high-throughput policy evaluation. The most critical performance improvement in CPOL is caching. In the location-aware privacy implementation, CPOL is able to process most requests from the cache, thus reducing the handling time to a large extent. CPOL is able to achieve such a high hit rate because it gives the application developer control over invalidation. Evaluation is conducted to test performance of CPOL, and results show that CPOL provides a good solution to the evaluation of policies.

Liu *et al.* [8] propose the first approach towards improving the XACML policy evaluation performance. In [8], they propose fast policy evaluation algorithms which can also be adapted to support various policy languages. In order to separate policy performance issues from correctness, they propose two fundamental techniques for fast policy evaluation: policy normalization and canonical representation. They implement their proposed algorithms in a policy evaluation system called XEngine [21]. They perform extensive comparison with the Sun PDP, which is the industrial standard for XACML policy evaluation. The experimental results show that XEngine is orders of magnitude faster than Sun PDP.

Additional efforts have been devoted to implementing high-performance policy evaluation, focusing on XACML, which has become the *de facto* standard for specifying access control policies. Nils *et al.* [17] propose a decision cache for XACML. The decision cache is implemented as an XACML obligation service, where a specification of the XML elements to be authorized and anonymised is sent to the PEP during the initial authorization. Further authorization check of the individual XML elements according to the authorization specification is then performed on all matching XML resources, and decisions are stored in the decision cache. The decision cache improves XACML performance significantly.

Unlike the above performance optimization mechanisms, we introduce an efficient cache mechanism for general policies. CPOL implements a cache mechanism by using simpler cache conditions which are not as precise as the original condition in policies. However, we further decrease the number of variables to be evaluated in caches by dividing variables into mutable variables and non-mutable variables. For those non-mutable variables, we directly get their values in tokens without acquiring them and accept a minimal chance that a wrong decision may be made. Our experiments show that performance is significantly improved. Liu *et al.* also introduce two policy evaluation techniques focusing on XACML, but our cache mechanism can be applied to more general policies.

VII. Conclusion and Future Work

Performance has not been the main focus for policy-based management methodologies for large-scale networked systems. However performance is a critical requirement for many deployments of policy-based management systems. In this paper, we introduced the *Mutable Variable Aware Cache (MVAC)* mechanism to achieve high-performance policy decision for general policies. The core idea of our cache mechanism is to divide variables into mutable variables and non-mutable variables and to fix time windows during which changes to non-mutable variables are ignored. We introduced a new concept of *Mutable Variable Aware Token (MVAT)* object to record states of variables. There are two kinds of cache entries: *ResultCache* and *TokenCache*. We deal with them in different ways to achieve an efficient policy

evaluation process. Experimental results showed our PDP implemented an efficient general policy decision process.

As part of future work, we would like to include in our PDP some policy combination techniques for dealing with conflicting policies. As the analysis of policy conflicts [22][23][24] is one of the key issues in policy-based management, we need to extend our PDP to include policy combining algorithms in our future work and to understand their impact on performance.

ACKNOWLEDGMENT

This paper is supported by the 863 project (2011AA100701), the Opening Projects of Key Lab of Information Network Security, Ministry of Public Security and DNSLAB of CNNIC, the project of Natural Science Foundation of Shanghai (12ZR1402600), and the project of Innovation Foundation of STCSM (12511504200). We thank anonymous reviewers for their valuable comments. Weili Han is the corresponding author.

REFERENCES

[1] M. S. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 2, no. 4, pp. 333–360, December 1994.

[2] D. C. Verma and T. J. Watson, "Simplifying network administration using policy-based management," *IEEE Network*, vol. 16, no. 2, pp. 20 – 26, Mar/Apr 2002.

[3] L. Lymberopoulos, E. C. Lupu, and M. S. Sloman, "An adaptive policy-based framework for network services management," *Journal of Network and Systems Management*, vol. 11, no. 3, pp. 277–303, September 2003.

[4] E. C. Lupu and M. S. Sloman, "Towards a role-based framework for distributed systems management," *Journal of Network and Systems Management*, vol. 5, no. 1, pp. 5–30, 1997.

[5] R. Bhatia, J. Lobo, and M. Kohli, "Policy evaluation for network management," in *Proceedings of 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2000, pp. 1107–1116.

[6] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy core information model – version 1 specification," IETF, RFC 3060, Febrary 2001, http://www.ietf.org/rfc/rfc3060;.

[7] K. Borders, X. Zhao, and A. Prakash, "CPOL: high-performance policy evaluation," in *CCS '05 Proceedings of the 12th ACM conference on Computer and communications security*, November 2005, pp. 147 – 157.

[8] A. X. Liu, F. Chen, J. Hwang, and T. Xie, "Designing fast and scalable XACML policy evaluation engines," *IEEE Transactions On Computers*, vol. 60, no. 12, pp. 1802–1817, December 2011.

[9] G. Ahn, H. Hu, J. Lee, and Y. Meng, "Representing and reasoning about web access control policies," *Proceedings of 2010 IEEE 34th the Computer Software and Applications Conference (COMPSAC)*, pp. 137 – 146, July 2010.

[10] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu, "Authorization recycling in hierarchical rbac systems," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 1, p. 3, 2011.

[11] M. Bali, *Drools JBoss Rules 5.0 developers guide*. Packt Publishing, 2009, ch. Introduction, User Guide, Rule Language Reference.

[12] JBoss, "Drools expert user guide," http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/pdf/drools-expert-docs.pdf, 2012.

[13] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, no. 1, pp. 477–489, January 2012.

[14] M. I. Yague, "Survey on XML-based policy languages for open environments," *Journal of Information Assurance and Security*, vol. 1, pp. 11–20, 2006.

[15] M. S. Sloman and E. C. Lupu, "Security and management policy specification," *IEEE Network*, vol. 16, no. 2, pp. 10–19, Mar/Apr 2002.

[16] OASIS, "Core specification: extensible access control markup language," http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.pdf, 2010.

[17] N. Ulltveit-Moe and V. Oleshchuk, "Decision-cache based XACML authorisation and anonymisation for xml documents," *Computer Standards and Interfaces*, vol. 34, no. 6, pp. 527–534, November 2012.

[18] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009, ch. Introduction, pp. 1–11.

[19] J. Strassner and S. Schleimer, "Policy framework definition language," http://www.ietf.org/proceedings/43/I-D/draft-ietf-policy-framework-pfdl-00, November 1998.

[20] K. Jayaraman, V. Ganesh, M. V. Tripunitara, M. C. Rinard, and S. J. Chapin, "Automatic error finding in access-control policies," in *ACM Conference on Computer and Communications Security*, 2011, pp. 163–174.

[21] A. X. Liu, F. Chen, J. Hwang, and T. Xie, "XEngine: a fast and scalable XACML policy evaluation engine," in *SIGMETRICS '08 Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, June 2008, pp. 265–276.

[22] J. D. Moffett and M. S. Sloman, "Policy conflict analysis in distributed system management," *Journal of Organizational Computing*, vol. 4, no. 1, pp. 1–22, 1994.

[23] E. C. Lupu and M. S. Sloman, "Conflicts in policy-based distributed systems management," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 852 – 869, November/December 1999.

[24] N. D. Griffeth and H. Velthuijsen, "Reasoning about goals to resolve conflicts," in *Proceedings of International Conference on Intelligent and Cooperative Information Systems*, May 1993, pp. 197 – 204.