Rule Anomalies Detecting and Resolving for Software Defined Networks

Pengzhan Wang,Liusheng Huang,Hongli Xu,Bing Leng,Hansong Guo
Email: {pzwang,lengb,guohanso}@mail.ustc.edu.cn,{lshuang,xuhongli}@ustc.edu.cn
School of Computer Science and Technology, University of Science and Technology of China, China

***Abstract*—Software Defined Network (SDN) is facilitating rapid innovation of network by providing a programmable network infrastructure. However, managing SDN flow rules, especially among multiple modules and administrators, has become complex and error-prone. Different controller modules with diverse objectives may be installed on the SDN controller, which can lead to anomalies among policies and rules. In this paper, we propose ADRS(Anomaly Detecting and Resolving for SDN) to solve this problem. Firstly, we analyse the rule-level anomalies that may occur in SDN based on OpenFlow protocol. Then we present an interval tree model for rapid rule scanning and a *share* model for network privilege allocating. By applying these models, we provide an automatic algorithm to detect and resolve the anomalies among SDN modules. Moreover, a rule-recovery mechanism is presented to avoid modification faults. We also implement and evaluate our system in the OpenDayLight controller.**

I. Introduction

TABLE I: Example for SDN Rule

rule	*prot*	*src_ip*	*src_p*	*dst_ip*	*dst_p*	action
r1	TCP	192.168.2.1	*	*.*.*.*	80	drop
r2	TCP	192.168.2.*	*	*.*.*.*	80	outport:8
r3	TCP	*.*.*.*	*	216.2.2.3	80	outport:8
r4	TCP	192.168.2.*	*	216.2.2.3	80	drop
r5	TCP	*.*.*.*	*	216.2.2.*	80	outport:8
r6	TCP	192.170.*.*	*	216.2.1.*	80	drop

Software Defined Network has gained great attention in industry by providing significant flexibility in programming the network. Based on the idea of separating control and data plane, SDN provides a new network solution to efficiently tackle the increasing demands of the challenging network landscape. The SDN operators implement network functions and intelligent applications through installing a set of flow rules on the OpenFlow switches. As flow tables are shared by independent controller modules, it's inevitable that some unanticipated anomalies will occur. For example, in a data center network, a load-balance module may install a routing entry like rule 5 in Table 1, meanwhile, rule 4 is implemented by a firewall module to deny certain insecure network services. In such a scenario, the rule with lower priority will turn invalid while its administrators are kept unaware. If rule 4 keeps a lower priority, all the matched packets will still be directed to port 8 by rule 5, resulting in a severe security problem. Besides, there are likely redundant rules, such as rule 3 and rule 5, if rules are manually configured. It will cause a waste of the limited flow table. Therefore, it's an urgent problem that must be solved before we can take full advantage of SDN.

Fortunately, this problem has already got some attention in recent years [1] [2] [3]. Most of these studies are based on a high-level solution, which focuses on avoiding conflicts among network resources or polices. [1] and [3] define the semantics of policy and resolve conflicts with user-defined conflict-resolution operators. When a new policy is added, new operators should be defined and the semantics need to be modified accordingly. In [2], each SDN module needs to implement some specific API to identify and install resource allocations. In a way, these methods are tightly-coupled with modules, which makes it complex to extend new modules. Relatively, a rule-level mechanism will be much more practical for SDN developers and operators for some reasons. Firstly, a real-time anomaly detecting of the rules is indispensable and significant for a SDN module developer, which is really useful for debugging. Secondly, every policy should be implemented through basic rules, it's feasible to avoid policy anomalies by reasonable configuration of rules. What's more, a rule-level mechanism is detached from modules and much more works can be done besides conflict resolving, e.g., rule redundancy elimination.

There are also similar studies in the firewall policy anomaly detecting problem [4] [5] [6] [7] [8]. But their solutions are not absolutely suitable for the problem in SDN for the following reasons: (1) The rules in SDN are much more complex than firewall rules, e.g., an OpenFlow [9] rule may contain more than 5 kinds of actions. (2) The size of the flow table is limited on the OpenFlow switch due to the design of TCAM. (3) Much more functions need to be implemented in SDN, whose rules will be more unmanageable. (4) Rules are updated much more frequently in SDN, therefore, a higher performance is required to detect and handle anomalies.

In a word, the problem in SDN is much more challenging because all of the above constraints should be taken into account. In order to avoid the anomalies without adding complexity in modules, we present a reasonable scheme named ADRS to detect and resolve rule-level conflicts and redundancy. Firstly, we formally define the relationships as well as anomalies between two OpenFlow rules. Based on the analysis of anomalies, a state transition diagram is designed to detect the abnormal rules. We also improve the detecting performance through an interval tree model. After that, we eliminate anomalies by resetting the rule's priority or removing certain rules. Besides, we take full account of the potential errors caused by the operations of the abnormal rules. At last, we implement and test ADRS in the OpenDayLight controller. The evaluation result shows that ADRS achieves a satisfactory performance in anomaly detecting and resolving. The main contributions of this paper are list as follows:

- Both conflicts and redundancies are detected within a appropriate processing time.
- We allocate privileges to each module to optimize the management of SDN resources.
- Anomalies are resolved without increasing the size of

978-1-4799-5952-5/15/$31.00 ©2015 IEEE

table and the complexity of modules.
- A rule recovery mechanism is presented to improve the robustness and validity of our system.

The rest of paper is organized as follows. In Section II, we discuss the related work both in SDN and firewalls. Then we analyse the relations among OpenFlow rules and classify the potential rule anomalies in Section III. Next we describe the rule-based mechanism to detect and resolve the anomalies in Section IV. The simulation results are presented in Section V. At last, we conclude the paper in Section VI.

II. Related Work

A. Relevant Researches in SDN.

HFT [1] defines the semantics of policy and resolves conflicts with user-defined conflict-resolution operators. An improved algorithm is proposed in PANE [3], which presents the design and implementation of an API for applications to control a SDN. PANE uses a global *share tree* to limit the authority of principles. And both of these frameworks are focused on the policy conflict prevention.

Athens [2] resolves resource conflicts among SDN based on a family of voting procedures. Two properties are defined to measure the voting rate: precision and parity. However, the value of voting rate is difficult to measure accurately and consistently across modules.

FortNOX [10] provides a way to check OpenFlow rule contradictions including those with *set* and *goto* actions. This work only gives a formalization description of OpenFlow rule conflict and a solution based on authorization role. In this paper, we will present a much more detailed scheme for rule conflict detecting and resolving.

B. Rule Conflict Problem in Firewall.

The similar problem of conflict detecting and resolving in firewalls has been studied in the past years. In [4] [5], a firewall policy tree model is presented, which provides a simple representation of the filtering rules and at the same time allows for easy discovery of anomalies among these rules. We optimize this tree model by combining some intersectant branches to a interval node.

The study in [6] [7] resolves the rule conflict by splitting conflict rules into disjoint rules, which is at the cost of extra rules. Obviously, it's not suitable in OpenFlow switches as the table size is limited. [8] makes an analysis of conflicts and resolves them through elimination of some rules and reordering the remaining rules. All of the above works are designed to handle the rules with only two actions, while much more actions should be considered in OpenFlow rules.

III. Modeling Of Rule Anomalies

In this section, we formally describe the model of SDN policy in three aspects: the formalization of filtering rule, formally definition of rule relations and classification of rule anomaly.

A. Filtering Rule Format

We formalize the filtering rule based on the OpenFlow protocol, which has been widely used in SDN industry. Each OpenFlow rule specifies a matching field $\mathcal{M}$, an action set $\mathcal{A}$ and a priority field $\mathcal{P}$. For simplicity, we consider the five common fields as rule match fields: protocol type $prot$, source IP address src_ip, source port src_port, destination IP address dst_ip and destination port dst_port. Note that both src_ip and dst_ip only use subnet mask to define the effective IP address. As defined in OpenFlow, the action set mainly consists of $output$, $drop$, $group$, and $meter$. The $output$ action forwards a packet to a specified port, $drop$ discards a packet, $group$ points to a group entry to represent additional methods of forwarding, $meter$ enables flows to implement various simple QoS operations, such as rate-limiting. Each of these actions can only be executed once in a rule, and besides $drop$, the others can coexist with another. If the action set is none, then it will default to $drop$. The $\mathcal{P}$ field defines the matching precedence and the rule with the larger value of $\mathcal{P}$ will be matched preferentially. We give an example of OpenFlow rule in Table 1. Formally, a rule can be defined as follows:
$\mathcal{R} = \{\mathcal{M}, \mathcal{A}, \mathcal{P}\}$.
$\mathcal{M} = \{prot, src\_ip, src\_port, dst\_ip, dst\_port\}$.
$\mathcal{A} \subseteq \{output, drop, group, meter\}$.

B. Formalization of Rule Relations

In this subsection we define all the possible relations that may exit between filtering rules. Only the match field is taken into account to determine the relation between two given rules.

Definition 1 : Rule R_x and R_y are *disjoint* if at least one field in R_x is not a subset nor a superset nor equal to the corresponding field in R_y. Formally,

$$R_x \mathfrak{R}_D R_y \Leftrightarrow \exists i : R_x[i] \not\bowtie R_y[i]$$

where $\bowtie \in \{\subset, \supset, =\}$ and i $\in \mathcal{M}$.

Definition 2 : Rule R_x and R_y are *exactly matching* if every field in R_x is equal to the corresponding field in R_y. Formally,

$$R_x \mathfrak{R}_{EM} R_y \Leftrightarrow \forall i : R_x[i] = R_y[i]$$

where $i \in \mathcal{M}$.

Definition 3 : Rule R_x and R_y are *inclusively matching* if they do not exactly match and if every field in R_x is a subset or equal to the corresponding field in R_y. Formally,

$$R_x \mathfrak{R}_{IM} R_y \Leftrightarrow \forall i : R_x[i] \subseteq R_y[i] \ and \ \exists j : R_x[j] \subset R_y[j]$$

where $i, j \in \mathcal{M}$.

Definition 4 : Rule R_x and R_y are *correlated* if some fields in R_x are subsets or equal to the corresponding fields in R_y and the rest of the fields in R_x are supersets of the corresponding fields in R_y.Formally,

$$R_x \mathfrak{R}_C R_y \Leftrightarrow \forall i : R_x[i] \bowtie R_y[i]$$
$$and \ \exists i, j : R_x[i] \subset R_y[i] \ and \ R_x[j] \supset R_y[j]$$

where $i, j \ \in \mathcal{M}$ and $i \neq j$.

C. Anomaly Classification

In this subsection we describe and formally define the possible policy anomalies. Before that, we define an operator between two action set A and B : $\mathcal{C}$. If A $\mathcal{C}$ B, then all of the actions contained in A and B are coexisting, also A $\not\mathcal{C}$ B means some incompatible actions exist between A and B. Let $\mathcal{I}$ be the action parameter, $\mathcal{I}(output)$ means the forwarding port number, $\mathcal{I}(group)$ means the group entry ID which a packet will be directed to, also $\mathcal{I}(meter)$ means the specified meter entry ID. Formally,

$$A \ \mathcal{C} \ B \Leftrightarrow A = B = \{drop\}$$
$$or \ \forall \alpha : \alpha \in A \cap B, \mathcal{I}(A.\alpha) = \mathcal{I}(B.\alpha) \ \ and \ \ drop \notin A \cup B$$

It should be noted that this anomaly classification only

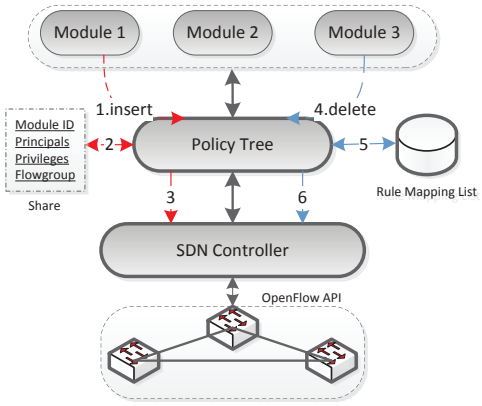

Fig. 1: The architecture of ADRS

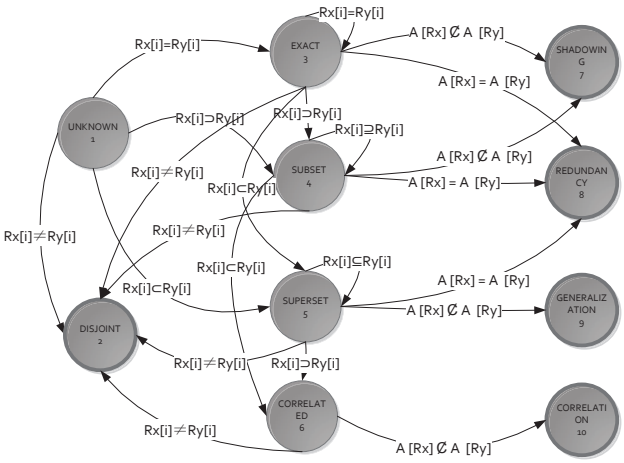

Fig. 2: State transition diagram for detecting anomaly

considers the match and action field in the rule, we will take priority field into account when resolving the anomalies. All of the anomalies are defined as follows:

Shadowing : A rule is shadowed when an existing rule with incompatible actions matches all the packets that match this rule. For example, rule 4 is shadowed by rule 3 in Table 1. This may cause an implemented policy to be invalid while the administrator is kept unwitting. Besides, these shadowed rules also waste the limited flow table resource. Formally,

$$R_y \mathfrak{C}_S R_x \Leftrightarrow R_y \mathcal{R}_{IM} R_x \ and \ \mathcal{A}[R_x] \ \not\subset \ \mathcal{A}[R_y]$$

where $\mathcal{A}[R]$ = action set of rule R.

Redundancy : A rule is redundant if there is another rule that produces the same matching and actions such that it makes possible to combine these rules into a single one. As shown in Table 1, rule 3 is redundant to rule 5. A redundancy will not cause any policy error, however, it's meaningful to discover and eliminate the redundant rules timely to relieve the pressure of the limited flow table. Formally,

$$R_x \mathfrak{C}_R R_y \Leftrightarrow R_x \mathcal{R}_{IM} R_y \ or \ R_y \mathcal{R}_{IM} R_x \ or \ R_y \mathcal{R}_{EM} R_x \\ and \ \mathcal{A}[R_x] \ = \ \mathcal{A}[R_y]$$

Correlation : Two rules are correlated if they have different filtering actions, and the first rule matches some packets that match the second rule and the second rule matches some packets that match the first rule. It's a severe problem when correlation occurs, which will result in confusion in packet processing. Referring to Table 1, rule 1 is correlated with rule 3. Formally,

$$R_y \mathfrak{C}_C R_x \Leftrightarrow R_x \mathcal{R}_C R_y, \mathcal{A}[R_x] \ \not\subset \ \mathcal{A}[R_y]$$

Generalization : A rule is a generalization of a preceding rule if they have incompatible actions, and the first rule can match all the packets that match the second rule. For example, rule 2 is a generalization of rule 1 in Table 1. Generalization also leads to a conflict in processing packets and a waste of flow table. Formally,

$$R_y \mathfrak{C}_G R_x \Leftrightarrow R_x \mathcal{R}_{IM} R_y, \mathcal{A}[R_x] \ \not\subset \ \mathcal{A}[R_y]$$

IV. Anomaly Detection and Resolving

In this section, we describe the framework of ADRS and present our algorithms to detect and resolve the potential anomalies among OpenFlow rules. In the first step, we describe the detailed process to discover the anomaly between two rules. To accelerate the detecting process, we represent an interval tree model to scan the pre-existing rule set. When an anomaly is detected, we make a solution depending on the priority field and the action set. In addition, considering that the SDN modules may repeal or modify some rules, we provide a mechanism to make sure all the conflicted rules can be resumed after the anomaly resolving process. The overall structure of ADRS is shown as Fig.1, which is mainly composed of functional modules, detecting layer, SDN controller and OpenFlow switches. We will describe the components of detecting layer in detail in this section.

A. Anomaly Detecting Algorithm

In section III we have analysed all the possible anomalies based on the relations of match and action fields between OpenFlow rules. To check whether there is any anomaly between two given rules, we need to compare each match and action field.

The state diagram in Fig.2 illustrates the rule anomaly detecting process for any two OpenFlow rules: R_x and R_y, where R_y follows R_x. Based on the classification of anomalies, five final-states are defined in this diagram: Disjoint, Shadowing, Redundancy, Generalization and Correlation. The state transition starts with the Unknown state, then each field in R_y is compared to the corresponding field in R_x. If a disjoint field is found between R_x and R_y, then we confirm that no anomaly exists between them. If every field in R_y is a subset or equal to the corresponding field in R_x and the action fields are compatible, R_y is redundant to R_x, while if the action fields are incompatible, R_y is shadowed by R_x. If every field in R_y is a superset or equal to the corresponding field in R_x and the action fields are compatible, R_y is redundant to R_x, while if the action fields are incompatible, R_y is a generalization of R_x. If some fields in R_y are subsets or equal to the corresponding fields in R_x, while the other fields in R_y are supersets to the corresponding fields in R_x and their actions are incompatible, then R_y is correlated with R_x.

Consider the example of rule 1 and rule 2 in Table 1, the processing state transits as $1 \rightarrow 3 \rightarrow 5 \rightarrow 5 \rightarrow 5 \rightarrow 5 \rightarrow 9$, which returns a generalization anomaly. The detail anomaly detecting algorithm is presented in Algorithm 1.

Algorithm 1 resolves the problem of anomaly detecting

Algorithm 1 Detecting-Anomaly(R_x, R_y)

```
relation ← UNKNOWN
anomaly ← DISJOINT
for all field i in M do
    if R_x[i] = R_y[i] then
        if relation = UNKNOWN then
            relation ←EXACT
        end if
    else if R_x[i] ⊂ R_y[i] then
        if relation ∈ {SUBSET, CORRELATED } then
            relation ← CORRELATED
        else
            relation ← SUPERSET
        end if
    else if R_x[i] ⊃ R_y[i] then
        if relation ∈ {SUPERSET, CORRELATED } then
            relation ← CORRELATED
        else
            relation ← SUBSET
        end if
    else
        return anomaly
    end if
end for
if relation =CORRELATED and A[R_x] ⊄A[R_y] then
    anomaly ← CORRELATION
else if relation =SUPERSET then
    if A[R_x] = A[R_y] then
        anomaly ← REDUNDANCY
    else if A[R_x] ⊄A[R_y] then
        anomaly ← GENERALIZATION
    end if
else if relation ∈ {SUBSET,EXACT } then
    if A[R_x] = A[R_y] then
        anomaly ← REDUNDANCY
    else if A[R_x] ⊄A[R_y] then
        anomaly ← SHADOWING
    end if
end if
return anomaly
```

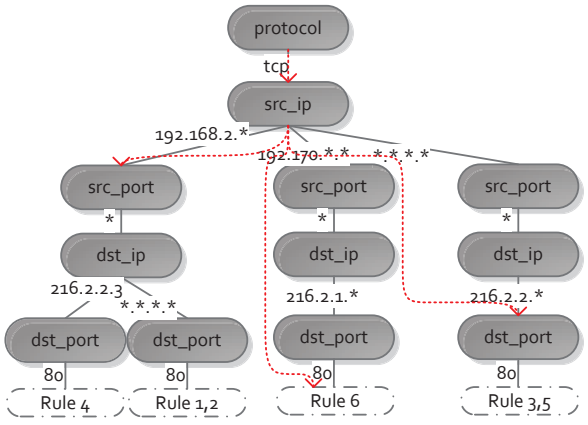

Fig. 3: The interval tree for OpenFlow rules

between two rules. However, there may exist thousands of rule entries in an OpenFlow switch. Comparing every rule directly with a new rule will lead to a high latency. Therefore, we design an interval tree model showed as Fig.3 to accelerate the detecting process. The main idea of this model is to narrow down the rule set to be detected by partitioning each matching field. Each node is corresponding to a match field and each leaf node contains the associated actions. The node's branches are sorted in ascending order of the field value and the intersecting fields will share the same branch. The value of each field will be saved in its common branch node. In this way, major disjoint rules can be ignored in the scanning process, as we only need to find the related branches among the ordered branches. For example, rule 1, rule 2 and rule 4 have the same prefix of 192.168.2.*, then these rules share the same branch of the src_ip node. It should be noted that if the node contains a field with the value of any, an associated branch should be added at the last of the branch list.

The basic idea for discovering anomalies is to determine whether any two rules coincide in their policy tree paths. If the path of a rule coincides with the path of another rule, there is a potential anomaly that can be determined based on the Algorithm 1. If rule paths do not coincide, then these rules are disjoint and they have no anomalies. Before a new rule is inserted into the flow table, we need to check it along the policy tree. The detecting process is started at the root of policy tree meanwhile the transition routine described in Algorithm 1 is invoked. If the current field of the inserted rule is a subset or equal to an existing branch field, then the detecting process will be continued along this branch and the other branches excepting the last one will get ignored. If the field of the new rule is a superset of the current branch value, we will check the next branch until all of the related branches are found, after that, the new rule should be compared with the rules contained in the related branches. For an instance, when a rule with the value of {TCP, 192.170.3.*, *, 216.2.1.7, 80, drop} is added in Table.1. The checking path is illustrated as the red dotted line in Fig.3. In this case, the times of compare operation are reduced from 19 to 11, which almost achieved 50% reduction compared with scanning in sequence.

B. Anomaly Resolving

OpenFlow policy is considered incorrect if any anomaly declared in Section III is detected. As the individual self-contained modules increase, the possibility of anomaly will increase. If all modules can design their own rules without any standard or constraint, it will be terrible for network administrators because any possible rules may be created by an unknown module. In this case, it's impossible to find the responsible principal for a certain anomaly. Therefore, it's necessary to make rules relevant to the corresponding modules and design constraints for rules. As for ADRS, the main idea of anomaly resolving includes two aspects: anomaly avoidance by distinct privilege allocation and anomaly elimination after detecting.

The PANE system [3] presents a *share* model to limit the authority of principals, which contains three components: its principals, privileges, and flow-group. A *share* states who(which principals) can say what(which messages) about which flows in the network. In this paper, we apply the *share* model to associate the rules with its principals and privileges.

We have observed that most of the anomalies arise from devised modules with different functions. If we delegate certain authority to each module based on its influence and functional-

ity, then some anomalies can be avoided through a hierachical design of modules. For example, a firewall module can get a more permissive privilege than a load balance module, and certain conflicts between these modules can be automatically eliminated by hierachical rules. As shown in Fig.1, every SDN module should create a $share$ entry containing the three components and associate its rules with this entry. A $share$ defines the principals, privileges as well as the flows to be controlled for each module. When a module tries to install new rule in the OpenFlow switch, the information in its $share$ entry can be taken into account to decide the rule's priority field.

On the other hand, a set of solutions are needed to eliminate the existing anomalies of OpenFlow rules. In order to reduce the usage of the limited flow table, we prefer to settle conflicts by reconfiguring rule priority or removing certain rules. In this case, it's necessary to give notice to the modules' principals when their rules are changed. Three types of message are defined: M_{mod}(a rule get modified), M_{cov}(a rule is covered), M_{rm}(a rule is removed). The anomalies with different priorities can be easily resolved as no conflict exists. When anomalies occur with the same priority, the priority of the inserted rule may get modified and certain rules may be removed. On this occasion, a warning message should be sent to the associated principals.

The anomaly resolving process is detailed in Algorithm 2. We define that R_y is inserted after R_x and an anomaly is detected through Algorithm 1. We handle the exception based on the principle of not increasing the flow table size as well as matching more packets. When redundancy is found between R_x and R_y, we will remove the subset one to make more packets matched(line1-5). If rules share the same priority and the anomaly is diagnosed as generation or correlation, we decrease R_y's priority to make a packet match both rules and apply the installed rule's action in the conflict region. Besides, a warning message should be sent to R_y's principal as this rule has been modified (line 8-12). If R_y is shadowed by R_x and has a higher or equal priority, it will be removed and the associated principal also gets notified (line 14-16). If R_y is a generalization of R_x and has a lower priority, then R_x will be removed as it will never get applied (line 21-22). In the other cases, there is no need to change rules but M_{cov} must be sent if a rule is covered by another one with higher priority (line 18,24).

Algorithm 2 Anomaly-Resolving(R_x,R_y,anomaly)

```
1: if anomaly = REDUNDANCY then
2:    if R_x ℜ_IM R_y then
3:       remove(R_x)
4:    else
5:       remove(R_y)
6:    end if
7: else
8:    if P[R_x] = P[R_y] then
9:       if anomaly ∈ {GENERALIZATION,CORRELATION}
         then
10:         P[R_y] ← P[R_y] − 1 and send M_mod(R_y)
11:      else
12:         remove(R_y) and send M_rm(R_y)
13:      end if
14:   else if P[R_x] > P[R_y] then
15:      if anomaly = SHADOWING then
16:         remove(R_y) and send M_rm(R_y)
17:      else
18:         send M_cov(R_y)
19:      end if
20:   else
21:      if anomaly = GENERALIZATION then
22:         remove(R_x) and send M_rm(R_x)
23:      else
24:         send M_cov(R_x)
25:      end if
26:   end if
27: end if
```

C. Rule Deletion and Modification

The above subsections describe the way to detect and resolve anomalies among the OpenFlow rules. As stated in Algorithm 2, when conflicts occur among rules, a rule may be modified or removed from the flow table to make the rule set harmonious. For example, module A and B produce R_x and R_y respectively, while R_x is redundant to R_y which contains the same action set, then R_y will be removed and R_x is kept in the table. The policy state is intact and correct for both A and B. However, when the principal of R_x try to delete R_x, then both R_x and R_y are removed, which leads to a rule miss for module B. Therefore, a delete or modify operation may result in a severe error in the network and this error will be indiscoverable for administrators.

To avoid this fault, we present a policy recovery mechanism to check the rule before it is deleted or modified. Firstly, we define a rule mapping list, whose entry is associated with a pair of abnormal rules which may lead to an anomaly we have mentioned in Section III. And each entry maps the original rules to the anomaly-resolved rules. For instance, the rule R_x and R_y is associated with a entry of $\{R_x, R_y\} \rightarrow \{R_x\}$. The rule deleting process is shown as the blue dotted line in Fig. 1. When a deleting instruction for R_y is issued from a certain module, it will be directed to the rule-mapping list first, then the associated entry is found. As the conflict rule pair $\{R_x, R_y\}$ is converted as $\{R_x\}$, this deleting result should be $\{R_x\}$. Then the decision will be transfered to the policy tree to remove the corresponding information of $\{R_y\}$. After that, the deleting process is over.

V. Implementation and Evaluation

In order to evaluate the performance and functionality of ADRS, we implement the related algorithms and technologies in the Java language. The elements of our test-bed are listed as below: (1) A host computer with a Core i5-3470 processor and 4GB of RAM. (2) A commercial OpenFlow switch. (3) The implementation of ADRS embedded in a specifically designed OpenFlow management open-source program called OpenDayLight.

To study the performance of the OpenFlow rule anomaly detecting algorithm, we produced three sets of OpenFlow rules. The first set contains rules that are different in source address only. The second set is the worst case in detecting anomaly, in which there's no disjoint rules existing, in other words, all fields in the rule will be checked. The third set includes rules that are randomly selected from the two previous sets in order to present the average case scenario.

The results we obtained are shown in Fig. 4(a). The set 1 shows the least processing time because the checking process

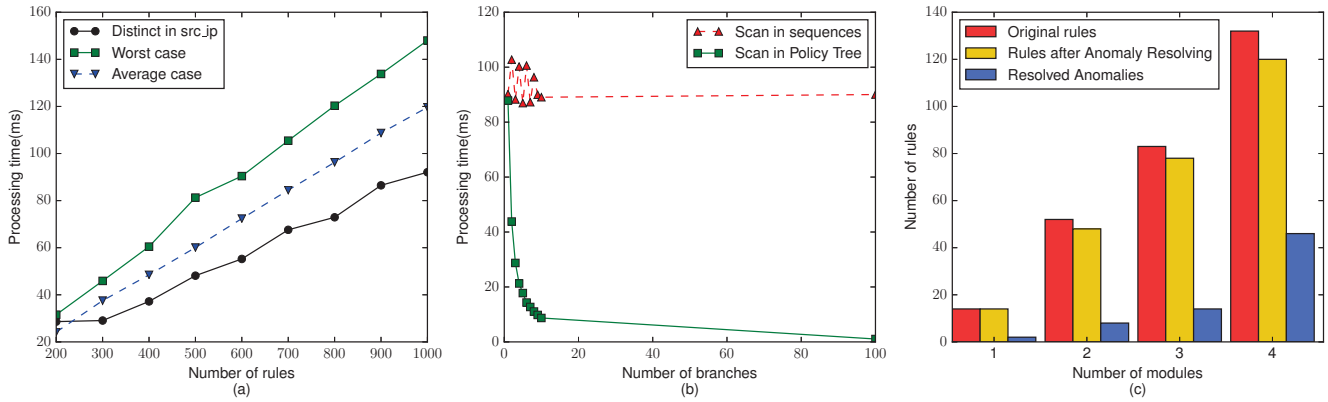

Fig. 4: Evaluation of Anomalies Detecting and Resolving

will end in the second match field and the rules are aggregated in one branch excepting src_ip field. Set 2 achieves the worst performance in the test, as each rule should be checked in all fields and more branches are contained in the policy tree. Set 3 shows the average processing time and represents a more practical scenario. Also, it shows that the processing time increases linearly as the number of rules increases.

We observed that the performance of anomaly detecting is affected by the policy tree structure, especially its branch numbers. Then we conduct a test based on a fixed-size rule set and changes the match fields according to the number of branches in src_ip node. For simplicity, each branch of the src_ip node contains the same number of subsets. The results are presented in Fig. 4(b). The red line denotes the processing time of rule scanning in sequence, which is less affected by the branch. The processing time of sequence checking keeps around 90 ms. The green line is measured by policy tree detecting. It can be observed that the processing time decreases greatly as the branch number increases because only one branch needs to be checked completely and the other rules will be ignored through limited comparison.

We implement 4 different modules to test the validity of ADRS in detecting and resolving rule anomalies. Each module's rule set is configured manually by specific operators. In Fig. 4(c), the red bar denotes the original rules' number, the yellow one means the number of rules after anomaly resolving and the purple one denotes the number of detected anomalies. As we can see from this chart that the anomalies increase as modules increase and the rule number is reduced after anomaly resolving.

VI. Conclusion

SDN provides an open infrastructure to program the network for all kinds of functional modules, resulting in some anomalies among rules such as rule redundancy and conflict. These errors are hard to detect by performing a manual or visual inspection. For this reason, automatic management of SDN rules is necessary. ADRS defines a framework and algorithms to alleviate this problem in two aspects: privilege allocations among modules to reduce the possibility of anomaly and a rule-level mechanism to detect and resolve anomalies. We accelerate the detecting process by an interval tree model. Besides, the potential exception caused by rule changes is also prevented through a rule recovery mechanism. We conduct a series of experiments in the OpenDayLight controller, and the results show ADRS works well in anomaly detecting. In the future, we will try to extend our scheme to adapt to the scenario with more match fields and actions in distributed OpenFlow switches.

VII. Acknowledgements

This paper is supported by the National Science Foundation of China under No. U1301256, 61170058, 61272133, 61472383 and 51274202, Special Project on IoT of China NDRC (2012-2766), Research Fund for the Doctoral Program of Higher Education of China No. 20123402110019, and the Natural Science Foundation of Jiangsu Province in China under No. BK2012632.

References

[1] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Hierarchical policies for software defined networks," in *HotSDN*. ACM, 2012.

[2] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul, "Democratic resolution of resource conflicts between sdn control programs," in *CoNEXT*. ACM, 2014.

[3] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An api for application control of sdns," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 327–338.

[4] E. S. Al-Shaer and H. H. Hamed, "Modeling and management of firewall policies," *Network and Service Management, IEEE Transactions on*, vol. 1, no. 1, pp. 2–10, 2004.

[5] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict classification and analysis of distributed firewall policies," *Selected Areas in Communications, IEEE Journal on*, vol. 23, no. 10, pp. 2069–2084, 2005.

[6] M. Abedin, S. Nessa, L. Khan, and B. Thuraisingham, "Detection and resolution of anomalies in firewall policy rules," in *Data and Applications Security XX*. Springer, 2006, pp. 15–29.

[7] H. Hu, G.-J. Ahn, and K. Kulkarni, "Detecting and resolving firewall policy anomalies," *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 3, pp. 318–331, 2012.

[8] S. Ferraresi, S. Pesic, L. Trazza, and A. Baiocchi, "Automatic conflict analysis and resolution of traffic filtering policy for firewall and security gateway," in *Communications, 2007. ICC'07. IEEE International Conference on*. IEEE, 2007, pp. 1304–1310.

[9] "Openflow switch specification,version 1.3.2," http://www.cs.princeton.edu/courses/archive/fall13/cos597E/papers/openflow-spec-v1.3.2.pdf.

[10] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu, "A security enforcement kernel for openflow networks," in *HotSDN*. ACM, 2012.