Building a Security Policy Tree for SDN Controllers[1]

S. Morzhov
Department of theoretical information science
Yaroslavl State University
Yaroslavl, Russia
E-mail: smorzhov@gmail.com

V. Sokolov
Department of theoretical information science
Yaroslavl State University
Yaroslavl, Russia
E-mail: valery-sokolov@yandex.ru

M. Nikitinskiy
Department of Innovative Development
A-Real Group, Energiya-Info Inc.
Yaroslavl, Russia
E-mail: man@a-real.ru

D. Chaly
Department of information and network technologies
Yaroslavl State University
Yaroslavl, Russia
E-mail: dmitry.chaly@gmail.com

***Abstract* — A firewall is a network security system that monitors and controls the incoming and outgoing network traffic based on predetermined security rules often called security policy. Managing firewall rules, especially for large enterprise networks, is complex and error-prone task. Firewall filtering rules have to be written carefully and organized in order to implement the security policy correctly. In addition, inserting or modifying a filtering rule requires thorough analysis of the relationship between this rule and other rules in order to determine the proper order of this rule.**

In this paper, the authors propose their classification of collisions that may occur among the rules of the security policy. In addition, the authors present their new efficient algorithm for detecting and resolving collisions in firewall rules on the example of the Floodlight SDN controller. This algorithm can be used to find security holes in the rules set, to minimize the number of rules in the existing security policy or to prevent appearance of any collisions in real time.

Keywords — software-defined network, firewall, PreFirewall, SDN, ACL, access control list

I. Introduction

The task of ensuring security in SDN is divided into two large parts. The first subtask is to ensure the secure functioning of all basic network infrastructure components, i.e. SDN controller and its applications, switches, managed by it and communication channels between the controller and the switches. The second subtask is to ensure the security of the end devices, servers and storage systems, i.e. classic network components.

Firewall is a classical network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules, often called security policy. In traditional networks, it is usually configured on the edge network devices. The SDN approach allows us to manage and configure network equipment, which makes it possible to apply firewall policies not only on the edge devices, but also on the rest of the switching equipment of the local network. When firewall is enabled on the network device close to another device, which generates prohibited traffic, the load on the local network is reduced; the resources of the network devices are freed up because the packets do not pass through the entire local network.

The management of modern networks is often difficult and confusing, since the network administrator needs to configure the network equipment (routers, switches) and software, which provides network services such as Network Address Translation (NAT), load balancing system server, intrusion detection system (IDS), IP-telephony server, etc. Each sources described above can generate and add new firewall rules. Since they do not coordinate with each other, collisions between added rules may occur. Two rules are collided when they are overlapped or shadowed by each other [2]. Security policy with collided rules is very difficult to analyze and maintain. Network administrator has to spend more time sorting out the rules, some of which may be meaningless. This monotonous and complicated work is error-prone and leads to holes and gaps appearance in security policy, which directly affect the functioning of the network as a whole.

II. Relation between rules

Firewall rule of SDN controller has the following structure (Table I) [1].

To resolve collisions that may occur after adding a new rule to the firewall, we need to compare the added rule

[1] This work is partially supported by RFBR under the grants 17-07-00823-a and 16-07-01103a.

978-1-5386-9456-5/18/$31.00 ©2018 IEEE

Table 1. Rule fields of SDN controller

Filed name	Possible value	Description
switchid	xx: xx: xx: xx: xx: xx: xx: xx	Switch identification number
src-inport	short	Input switch port number
src-mac	xx: xx: xx: xx: xx: xx	Source MAC-address
dst-mac	xx: xx: xx: xx: xx: xx	Destination MAC-address
dl-type	ARP or IPv4	protocol
src-ip	A.B.C.D/M	Source IP-address
dst-ip	A.B.C.D/M	Destination IPaddress
nw-proto	TCP or UDP or ICMP	protocol
tp-src	short	Source port number
tp-dst	short	Destination port number
priority	int	Priority of the rule (less is more important)
action	allow or deny	Allow or deny set of network flows which are match rule

with all existing ones. To compare two rules, a pairwise comparison of the values of the corresponding rules attributes is performed. Since the value of the attribute is a finite set, then for comparison it is necessary to use relations over sets. All possible relations between the sets of values of the corresponding attributes can be represented as follows:

(a) Set A and set B are not intersected, $A \cap B = \varnothing$.

(b) Set A and set B are equal, $A = B$.

(c) Set B is a subset of the set A, $B \subset A$.

(d) Set A and set B are intersected, $A \cap B \neq \varnothing$, but $A \not\subset B$ and $B \not\subset A$.

Comparison of two rules can be defined as follows [3, 4].

Definition 1. Two rules r and s are disjoint ($\mathfrak{R}_D$), if they have at least one attribute with disjoint values. Formally, it can be written as:

$$r\mathfrak{R}_D s, \text{ если } \exists a \in attr,\ a_r \cap a_s = \varnothing.$$

For example, rules 1 and 2 below are disjoint, because they have different values for the attribute “src-port” (21 and 9050):

1. tcp, 193.168.*, 192.168.0.1, 21, allow.
2. tcp, 193.168.*, 192.168.0.1, 9050, allow.

Definition 2. Two rules r and s are exactly matched ($\mathfrak{R}_{EM}$), if the corresponding values of all their attributes are equal. Formally, it can be written as:

$$r\mathfrak{R}_{EM} s, \text{ если } \forall a \in attr,\ a_r = a_s.$$

For example, rules 1 and 2 below are exactly matched since the corresponding values of all their attributes are equal:

1. tcp, 193.168.*, 192.168.0.1, 21, allow.
2. tcp, 193.168.*, 192.168.0.1, 21, allow.

Definition 3. Two rules r and s are inclusively matched ($\mathfrak{R}_{IM}$), if the rule r has at least one attribute whose value is a subset of the corresponding attribute of the rule s, and the remaining attributes of the rules are equal. Formally, it can be written as:

$$\exists a \in attr,\ a_r \subset a_s \text{ и } \forall b \in \{attr \setminus a\},\ b_r = b_s.$$

For example, rule 1 is a subset of rule 2, since all the attributes of rule 1 are equal to the corresponding attributes of rule 2, except for the attribute “src-ip”. In rule 1, the value of the attribute “src-ip” is a subset of the value of the “src-ip” attribute of rule 2:

1. tcp, 193.168.0.1, 192.168.0.1, 21, deny.
2. tcp, 193.168.*, 192.168.0.1, 21, deny.

Definition 4. Two rules r and s are correlated ($\mathfrak{R}_C$), if they are not disjoint, equal or inclusively match each other. Formally, it can be written as:

$$r\mathfrak{R}_C s, \text{ если } (r \neg \mathfrak{R}_D s) \wedge (r \neg \mathfrak{R}_{IM} s) \wedge (s \neg \mathfrak{R}_{IM} r).$$

For example, rules 1 and 2 are correlated, since the values of the attributes “nw-proto” and “src-port” are equal. For rule 1, the value of the attribute “src-ip” is a subset of the value of the corresponding attribute of rule 2, and the value of the attribute “dst-ip” of rule 1 is a superset of the value of the corresponding attribute of rule 2:

1. tcp, 193.168.*, *, 21, allow.
2. tcp, *, 192.168.0.1, 21, deny.

Lemma 1. Any two rules that have two attributes can be in one of four relations: $\mathfrak{R}_D$, $\mathfrak{R}_{EM}$, $\mathfrak{R}_{IM}$ or $\mathfrak{R}_C$.

Proof. Consider rules $R_x = \langle x_1, x_2 \rangle$ and $R_y = y_1, y_2$. The relation between them is determined by the relation between the corresponding values of their attributes, that is, $x_i \mathfrak{R} y_i$, where $\mathfrak{R} \in \{=, \subset, \supset, \bowtie\}$, $i = 1,\ 2$ The operator $\bowtie$ is defined as follows:

$$x \bowtie y \Leftrightarrow x \neq y \wedge x \not\subset y \wedge x \not\supset y \wedge x \cap y \neq \varnothing.$$

Consider all possible relations between attributes R_x and R_y.

If $x_1 = y_1$ и $x_2 = y_2$ then $R_x \mathfrak{R}_{EM} R_y$

If $x_1 = y_1$ и $x_2 \subset y_2$ then $R_x \mathfrak{R}_{IM} R_y$

If $x_1 = y_1$ и $x_2 \supset y_2$ then $R_x \mathfrak{R}_{IM} R_y$

If $x_1 = y_1$ и $x_2 \bowtie y_2$ then $R_x \mathfrak{R}_C R_y$

If $x_1 = y_1$ и $x_2 \mathfrak{R} y_2$ then $R_x \mathfrak{R}_D R_y$

If $x_1 \subset y_1$ и $x_2 = y_2$ then $R_x \mathfrak{R}_{IM} R_y$

If $x_1 \subset y_1$ и $x_2 \subset y_2$ then $R_x \mathfrak{R}_{IM} R_y$

If $x_1 \subset y_1$ и $x_2 \supset y_2$ then $R_x \mathfrak{R}_C R_y$

If $x_1 \subset y_1$ и $x_2 \bowtie y_2$ then $R_x \mathfrak{R}_C R_y$

If $x_1 \subset y_1$ и $x_2 \mathfrak{R} y_2$ then $R_x \mathfrak{R}_D R_y$

If $x_1 \supset y_1$ и $x_2 = y_2$ then $R_x \mathfrak{R}_{IM} R_y$

If $x_1 \supset y_1$ и $x_2 \subset y_2$ then $R_x \Re_C R_y$

If $x_1 \supset y_1$ и $x_2 \supset y_2$ then $R_x \Re_{IM} R_y$

If $x_1 \supset y_1$ и $x_2 \bowtie y_2$ then $R_x \Re_C R_y$

If $x_1 \supset y_1$ и $x_2 \Re y_2$ then $R_x \Re_D R_y$

If $x_1 \bowtie y_1$ и $x_2 = y_2$ then $R_x \Re_C R_y$

If $x_1 \bowtie y_1$ и $x_2 \subset y_2$ then $R_x \Re_C R_y$

If $x_1 \bowtie y_1$ и $x_2 \supset y_2$ then $R_x \Re_C R_y$

If $x_1 \bowtie y_1$ и $x_2 \bowtie y_2$ then $R_x \Re_C R_y$

If $x_1 \bowtie y_1$ и $x_2 \Re y_2$ then $R_x \Re_D R_y$

If $x_1 \Re y_1$ и $x_2 = y_2$ then $R_x \Re_D R_y$

If $x_1 \Re y_1$ и $x_2 \subset y_2$ then $R_x \Re_D R_y$

If $x_1 \Re y_1$ и $x_2 \supset y_2$ then $R_x \Re_D R_y$

If $x_1 \Re y_1$ и $x_2 \bowtie y_2$ then $R_x \Re_D R_y$

If $x_1 \Re y_1$ и $x_2 \Re y_2$ then $R_x \Re_D R_y$

Thus, it was shown that R_x and R_y are always in one of four relations: $\Re_D$, $\Re_{EM}$, $\Re_{IM}$ or $\Re_C$ ■.

Lemma 2. Adding one attribute to any two rules R_x and R_y that are in relation $\Re = \Re_D, \Re_{EM}, \Re_{IM}, \Re_C$, leaves the rules R_x and R_y in their former relation $\Re$ or translates them into the new relation $\Re' = \Re_D, \Re_{EM}, \Re_{IM}, \Re_C$.

Proof. Consider rules $R_x = \langle x_1, ..., x_k \rangle$ and $R_y = \langle y_1, ..., y_k \rangle$. Let is add an attribute x_{k+1} to the rule R_x and an attribute y_{k+1} to the rule R_y. Let is denote the new rules as R'_x and R'_y respectively. If rules R_x and R_y were in relation $\Re = \Re_D, \Re_{EM}, \Re_{IM}, \Re_C$ then rules R'_x and R'_y can be in one of the following relations:

If $R_x \Re_D R_y$ and $x_{k+1} = y_{k+1}$ then $R'_x \Re_D R'_y$

If $R_x \Re_D R_y$ and $x_{k+1} \subset y_{k+1}$ then $R'_x \Re_D R'_y$

If $R_x \Re_D R_y$ and $x_{k+1} \supset y_{k+1}$ then $R'_x \Re_D R'_y$

If $R_x \Re_D R_y$ and $x_{k+1} \bowtie y_{k+1}$ then $R'_x \Re_D R'_y$

If $R_x \Re_D R_y$ and $x_{k+1} \Re y_{k+1}$ then $R'_x \Re_D R'_y$

If $R_x \Re_{EM} R_y$ and $x_{k+1} = y_{k+1}$ then $R'_x \Re_{EM} R'_y$

If $R_x \Re_{EM} R_y$ and $x_{k+1} \subset y_{k+1}$ then $R'_x \Re_{IM} R'_y$

If $R_x \Re_{EM} R_y$ and $x_{k+1} \supset y_{k+1}$ then $R'_x \Re_{IM} R'_y$

If $R_x \Re_{EM} R_y$ and $x_{k+1} \bowtie y_{k+1}$ then $R'_x \Re_C R'_y$

If $R_x \Re_{EM} R_y$ and $x_{k+1} \Re y_{k+1}$ then $R'_x \Re_D R'_y$

If $R_x \Re_{IM} R_y$ and $x_{k+1} = y_{k+1}$ then $R'_x \Re_{IM} R'_y$

If $R_x \Re_{IM} R_y$ and $x_{k+1} \subset y_{k+1}$ then $R'_x \Re_{IM} R'_y$

If $R_x \Re_{IM} R_y$ and $x_{k+1} \supset y_{k+1}$ then $R'_x \Re_C R'_y$

If $R_x \Re_{IM} R_y$ and $x_{k+1} \bowtie y_{k+1}$ then $R'_x \Re_C R'_y$

If $R_x \Re_{IM} R_y$ and $x_{k+1} \Re y_{k+1}$ then $R'_x \Re_D R'_y$

If $R_x \Re_C R_y$ and $x_{k+1} = y_{k+1}$ then $R'_x \Re_C R'_y$

If $R_x \Re_C R_y$ and $x_{k+1} \subset y_{k+1}$ then $R'_x \Re_C R'_y$

If $R_x \Re_C R_y$ and $x_{k+1} \supset y_{k+1}$ then $R'_x \Re_C R'_y$

If $R_x \Re_C R_y$ and $x_{k+1} \bowtie y_{k+1}$ then $R'_x \Re_C R'_y$

If $R_x \Re_C R_y$ and $x_{k+1} \Re y_{k+1}$ then $R'_x \Re_D R'_y$

Thus, it was shown that R'_x and R'_y are in one of the following relations $\Re_D$, $\Re_{EM}$, $\Re_{IM}$ or $\Re_C$ ■.

Theorem. Relations $\Re_D$, $\Re_{EM}$, $\Re_{IM}$, $\Re_C$ form a universal set of relations between two rules R_x and R_y.

Proof. We will prove the theorem by the method of mathematical induction.

Base step. Showing that theorem is true for $k = 2$, $R_x = \langle x_1, x_2 \rangle$, $R_y = \langle y_1, y_2 \rangle$ is trivial. It follows from the lemma 1.

Step case. Let the theorem be true for $R_x = \langle x_1, ..., x_k \rangle$, $R_y = \langle y_1, ..., y_k \rangle$ and $R_x \Re R_y$, where $\Re \in \{\Re_D, \Re_{EM}, \Re_{IM}, \Re_C\}$. Let is add a new attribute x_{k+1} to the rule R_x, an attribute y_{k+1} to the rule R_y and denote the new rules as R'_x and R'_y. Since the rules R'_x and R'_y were obtained by adding one new attribute to the rules R_x and R_y such that $R_x \Re R_y$ then, by lemma 2, $R'_x \Re' R'_y$, where $\Re' \in \{\Re_D, \Re_{EM}, \Re_{IM}, \Re_C\}$ ■.

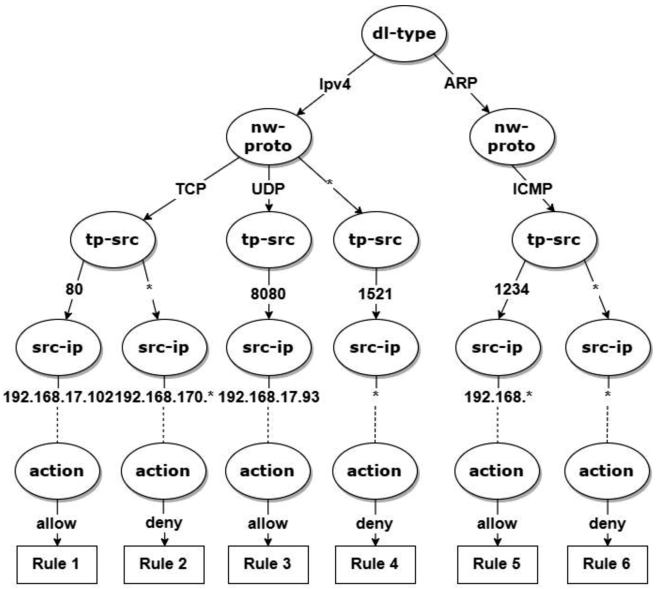

Fig. 1. Policy tree

Thus, a collision between two rules is understood as the relations between the rules given in definitions 2, 3 and 4.

III. Building policy tree

We will call a rule "good" if security policy remains free from all possible collisions, defined in chapter II, after adding this rule into it. If some collision occurred after adding a rule, then the rule will be called "bad". It is reasonable to assume that "good" rules are added into security policy more often than "bad" ones. In addition, it is worth noting that to understand whether the rule is a "good" or a "bad" one is impossible without comparing it with the rules, which were added previously to the security policy. Considering this, there is an obvious fact. To understand that a new rule is "good" is usually computationally not easier, but rather more difficult than to

understand that this rule is “bad”, because, in general, we can conclude that the rule is “good” only after we check that it does not collide with every rule added to the security policy earlier. Thus, the operation of adding a “good” rule should be as simple as possible.

Consider a tree-like data structure [5]. We will show that it is able to satisfy all conditions that have been set, and that it is optimal for our task. The rule consists of 12 attributes (see TABLE I). Each attribute can take one value from a certain set. For example, the attribute “dl-type” can only take values from the set {ipv4, arp, *} whose power is 3, and the attribute “src-ip” can be equal to any of $(2^8 + 1)^4$ IP addresses (possible wildcards[1] are taken into account). Then, we sort out all attributes except “priority” and “action” in order of increasing powers of the sets of their possible values and put “priority” and “action” attributes at the end of the received sequence.

So, as a result of ordering, a 12-dimensional tuple is obtained. Since the rules will be stored in the tree, the 12-dimensional tuple must correspond to one path from the root to the leaf in this tree, and each path from the root to the leaf in the tree must correspond to one 12-dimensional tuple representing the security policy rule. We establish a one-to-one correspondence between the values of the 12-dimensional tuple and the levels of the tree. The first value of the tuple corresponds to the root of the tree; the second one — to the vertices on the first level; the third — to the vertices on the second level etc. The twelfth element of the tuple will correspond to the leaves at the eleventh level of the tree (see Fig. 1). Since the “dl-type” attribute can have only three values, it will correspond to the root of the tree. The attribute “nw-proto” has four different values. It will correspond to the vertices of the first level of the rules tree. The leaves at the eleventh level of the tree will contain one of two possible values — allow for the resolving rule and deny for the forbidding rule.

Let each vertex of the rules tree contain a hash table (key-value pair). Keys in the hash table will contain the values corresponding to the attribute level of the security policy rules. Values in the hash table will contain memory addresses of the adjacent vertex on a lower level. Thus, the hash table of the tree root, shown in Fig. 1, will contain two keys: “ipv4” and “arp”. The values of these keys are the addresses to the left and right subtrees, respectively.

Note that the order of the elements in the tuple that correspond to a rule of the security policy was not chosen by chance. All attributes, except of “priority” and “action”, were ordered by increasing the powers of the sets of their possible values in order to minimize the lengths of the descending paths that arise when rules are added to the tree. Let us show that this approach ensures the most efficient use of memory necessary for storing the rules tree.

Suppose there are two rule trees. Let the first tree be constructed as described above, and the second one be constructed in the same manner, but with the only difference that the attributes in the tuple are ordered in descending order of the powers of the sets of their possible values. Thus, the first tree expanding gradually from the root to the leaves and the second tree in the general case becomes wide immediately. We estimate the maximum possible number of hash tables for the first and second trees. In evaluating, for simplicity of calculations, without loss of generality, let is suppose that both trees store rules with five attributes: “dl-type”, “nw-proto”, “tp-src”, “src-ip”, and “action”.

Consider the first tree. Maximum three entries — “tcp”, “udp”, “icmp” and “*” — can be in the hast-table of the first level. Therefore, on the third level, there can be $3 * 4 = 12$ tables having 2^{16} entries each because to store the port number (the third level of the tree corresponds to the attribute “tp-src” — the source port number), 16 bits of memory are allocated. At the fourth level, ip-addresses are stored, which means that the hash table can contain up to $(2^8 + 1)^4$ entries, considering possible wildcard characters. Thus, the first tree will contain

$$1 + 3 + 3 * 4 + 3 * 4 * 2^{16} = A$$

hash tables. Similarly, by calculating the number of hash tables for the second tree, we get

$$1 + (2^8 + 1)^4 + (2^8 + 1)^4 * 2^{16} + (2^8 + 1)^4 * 2^{16} * 4 = B,$$
$$B \ggg A.$$

Obviously, no matter how the attributes are ordered, the number of hash tables A in the first rule tree will be the smallest.

IV. Colisions resolution

Based on the relationship between the rules as well as the possible collisions defined in chapter II, an algorithm for resolving collisions was developed. The main idea of the algorithm is that the new rule have to be added to the rules tree, which is free of all collisions. If the path in the tree corresponding to the new rule coincides with some path in the tree representing the previously added rule, then a collision that must be resolved was found. Otherwise, adding a new rule will leave the security policy free from all kinds of collisions. This logic is implemented in the algorithm DiscoverCollision.

DiscoverCollision(rule, field, node, collision_state):

```
1. if field ≠ ACTION then
2. value_found = FALSE
3. if field in node.keys then
4.   if collision_state = NOCOLLISION then
5.     collision_state = REDUNDANT
6.     DiscoverCollision(rule, field.next,
```

[1] A wildcard character is a kind of placeholder represented by a single character, such as an asterisk (*), which can be interpreted as a number of literal characters or an empty string.

```
        field[node], collision_state)
7.  else
8.    if rule.field.value ⊃ branch.value
9.      if collision_state = GENERALIZATION then
10.       DiscoverCollision (rule, field.next, field[node],
            CORRELATION)
11.     else
12.       DiscoverCollision (rule, field.next, field[node],
            SHADOWING)
13.   else if rule.field.value ⊃ branch.value
14.     if collision_state = SHADOWING then
15.       DiscoverCollision (rule, field.next,
            field[node], CORRELATION)
16.     else
17.       DiscoverCollision (rule, field.next,
            field[node], GENERALIZATION)
18.     end if
19.   end if
20. else
21.   node.add(field, NULL)
22.   DiscoverCollision (rule, field.next, field[node],
        NOANOMALY)
23. else // action field was reached
24.   DecideCollision(rule, field, node, collision_state)
```

The DecideCollision algorithm that makes the final verdict regarding the presence or absence of collisions is called when the path of the rule added in the rule tree has been fully defined and the "action" attribute has been reached.

DecideCollision(rule, field, node, collision):

```
1.  if node in branch_list then
2.    branch = node.branch_list.first()
3.    if collision = CORRELATION then
4.      if rule.action ≠ branch.value then
5.        report rule rule.id is in correlation with rule
            branch.rule.id
6.      else if collision = GENERALIZATION and
7.        rule.action ≠ branch.value then
8.        report rule.id is a generalization of rule
            branch.rule.id
9.      else if collision = GENERALIZATION and
10.       rule.action = branch.value then
11.       branch.rule.setCollision(REDUNDANCY)
12.       report branch.rule.id is redundant to rule rule.id
13.     else if rule.action = branch.value then
14.       collision = REDUNDANCY
15.       report rule.id is redundant to rule branch.rule.id
16.     else if rule.action ≠ branch.value then
17.       collision = SHADOWING
18.       report rule.id is shadowed by rule branch.rule.id
19.     end if
20.   end if
21. rule.setCollision(collision)
```

After the collision type has been defined, a new rule can be added to the security policy (no collisions), not added (generalization or redundancy collision), or added partly (correlation). The latter case is the most interesting and complex, so it should be considered separately.

If it was determined that the two rules are correlated, they can be "broken up" into disjoint parts, which will be added to the security policy instead of adding the original correlated rule. For this purpose, first, we need to find a set of rules' attributes with disjoint values. After that, for each of the found attributes, the Split algorithm, listed below, is called. It modifies the rules r and s so that they do not correlate.

The Split algorithm receives two correlated rules r and s and an attribute a, whose value is disjoint for these rules. As can be seen from Fig. 2 the common part of the value of the attribute always starts with max{r.a.start, s.a.start} and ends with min{r.a.end, s.a.end}. The non-intersecting part before the common part always starts with min{r.a.start, s.a.start} and ends with max{r.a.start, s.a.start} – 1. The non-intersecting part after the common part always starts with min{r.a.end, s.a.end} + 1 and ends with max{r.a.end, s.a.end}. The disjoint parts of r and s rules can be added to the rules tree via the DiscoverCollision algorithm, as it is not guaranteed that they do not conflict with pre-existing rules.

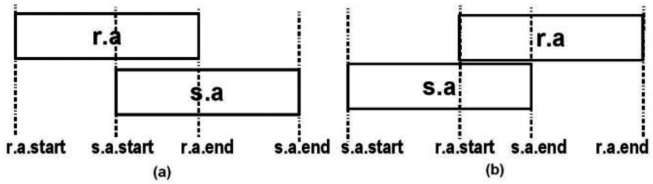

Fig. 2. Splitting an attribute.
(a) r.a.start < s.a.start & r.a.end < s.a.end, thus, the interval can be split on [r.a.start, s.a.start – 1], [s.a.start, r.a.end] and [r.a.end + 1, s.a.end]. (b) r.a.start > s.a.start & r.a.end > s.a.end, thus, the interval can be split on [s.a.start r.a.start – 1], [r.a.start, s.a.end] and [s.a.end + 1, r.a.end].

In the Split algorithm, the common part of the two correlated rules is calculated firstly, and then the disjoint parts are found and added to the rules tree as new ones.

Split(*r*, *s*, *a*):

```
1.  left = min(r.a.start, s.a.start)
2.  right = max(r.a.end, s.a.end)
3.  common_start = max(r.a.start, s.a.start)
4.  common_end = min(r.a.end, s.a.end)
5.  if r.a.start > s.a.start then
6.    DiscoverCollision(⟨(left, common_start-1), s'rest
        attributes⟩, first_filed, first_node, NOANOMALY)
7.  else if r.a.start < s.a.start then
8.    DiscoverCollision (⟨(left, common_start-1), r's rest
        attributes⟩, first_filed, first_node, NOANOMALY)
9.  if r.a.end > s.a.end then
10.   DiscoverCollision (⟨(common_end+1, right),
        r's rest attributes⟩, first_filed, first_node,
        NOANOMALY)
11. else if r.a.end < s.a.end then
```

```
12.  DiscoverCollision (⟨(common_end+1, right),
       s'rest attributes⟩, first_filed, first_node,
       NOANOMALY)
13. r = ⟨(common_start, common_end), r's rest attributes⟩
14. s = ⟨(common_start, common_end), s'rest attributes⟩
```

After the algorithms are completed, the rules tree will be free of all collisions, defined in chapter II.

V. Conclusion and future work

The main advantage of the collision resolution algorithm presented in chapter III is its, in general case, constant complexity. However, this algorithm has one significant drawback. Without cardinal changes, it is difficult to adapt it for working with rules containing wildcards. In this case, the word "difficult" should be understood in the sense that it is difficult to adapt the algorithm while maintaining the constant complexity. If we will not consider the requirement of maintaining constant complexity, then the collision resolution algorithm becomes trivial (see DiscoverCollision2).

DiscoverCollision2(rule, field, node, collision_state):

```
1.  if field ≠ ACTION then
2.  value_found = FALSE
3.  for each branch in node.branch_list do
4.    if branch.value = rule.field.value then
5.      value_found = TRUE
6.      if collision_state = NOANOMALY then
7.        collision_state = REDUNDANT
8.        DiscoverCollision(rule, field.next,
            branch.node, collision_state)
9.      else
10.       if rule.field.value ⊂ branch.value
11.         if collision_state = GENERALIZATION then
12.           DiscoverCollision (rule, field.next,
                branch.node, CORRELATION)
13.         else
14.           DiscoverCollision (rule, field.next,
                branch.node,  SHADOWING)
15.       else if rule.field.value ⊃ branch.value
16.         if collision_state = SHADOWING then
17.           DiscoverCollision (rule, field.next,
                branch.node, CORRELATION)
18.         else
19.           DiscoverCollision (rule, field.next,
                branch.node, GENERALIZATION)
20.       end if
21.     end if
22.   end for
23.   if value_found = False then
24.     new_branch = new TreeBranch(rule, rule.field,
          rule.filed.value)
25.     node.branch_list.add(new_branch)
26.     DiscoverCollision (rule, field.next,
          new_branch.node, NOANOMALY)
27.   end if
28. else. // action field was reached
29. DecideCollision(rule, field, node, collision_state)
```

The main difference between the DiscoverCollision2 algorithm and the DiscoverCollision algorithm is that at each level of the tree, before adding (or not adding) a new value, we compare it with all previously added ones. This ensures that all wildcards will be processed correctly. At the same time, this change leads to the loss of meaning in the use of hash tables. Moreover, complexity of the algorithm increases due to for each loop (line 3). In the future, we are planning to conduct research in this direction with a view to improving DiscoverCollision2 algorithm.

Among other things, the issue of the need to resolve all types of collisions remains open. Often, system administrators intentionally include rules that are correlated with other ones in security policy. The resolution of collisions in such a case is an error. Thus, it can be argued that the most important and priority remains an issue of the need to resolve collision, because sometimes it is not good to be unexpectedly too clever or pretend to be more clever than a qualified network administrator.

References

1. Floodlight SDN OpenFlow Controller, Web: https://github.com/floodlight/floodlight
2. Sergey Morzhov, Igor Alekseev and Mikhail Nikitinskiy, "Firewall application for Floodlight SDN controller," IEEE Control and communications (SIBCON), International Siberian Conference. Moscow, pp. 1–5, June 2016.
3. E. Al-Shaer and H. Hamed. "Design and implementation of firewall policy advisor tools," Technical Report CTI-techrep0801, School of Computer Science Telecommunications and Information Systems, DePaul University, August 2002.
4. M. Abedin, S. Nessa, L. Khan, B. Thuraisingham, "Detecting and Resolving Firewall Policy Anomalies," IEEE Transactions on Dependable and Secure Computing, Vol. 9, pp. 318–331.
5. Morzhov S.V. and Nikitinskiy M.A., "A new approach for detecting and resolving anomalies in security policy of the external firewall module of the Floodlight SDN controller," Modeling and Analysis of Information Systems. Yaroslavl, vol. 25, pp. 251–256, June 2018.