Atomic HyperRect Index: Efficient Set Operation on Packet Header Space

Danyang Li, Xiaohe Hu, Linli Wan and Zhi Liu
Department of Automation
Research Institute of Information Technology (RIIT)
Tsinghua University, Beijing, China
{lidangyan16, hu-xh14, wan-ll13}@mails.tsinghua.edu.cn
zhiliu.thu@gmail.com

Jun Li
Research Institute of Information Technology (RIIT),
Tsinghua University, Beijing China.
Tsinghua National Lab for Information Science and Technology (TNList), Beijing China.
junl@tsinghua.edu.cn

***Abstract*—To achieve a logically centralized control over the whole network, Software-Defined Networking (SDN) divides the network into three planes: application plane, control plane and data plane. However, due to the application-independent feature of SDN, together with the occurrence of manual configuration, system reset and other incidents, the operation rules in SDN data plane are sometimes inconsistent with the network rules in SDN control plane, making the network unable to meet the requirements from users. To ensure that the network behaves according to the policies set by user, SDN needs to periodically check the consistency between operation rules and network rules. With the increasing diversity of SDN deployment scenarios and protocols, the number of SDN rules grows exponentially. Therefore, it is crucial to design efficient algorithms for policy consistency verification in SDN. Since the basic of policy consistency verification is set operation on policy space, the key to implement efficient policy consistency verification algorithms is to develop efficient algorithms to do set operations on policy. In this paper, a novel algorithm AHR (Atomic Hyper Rect), based on rule projection and Bitmap index, is proposed to provide basic set operations on policy space. With real-life data sets, AHR is evaluated against three existing policy space operation algorithms, including r-BDD, wildcard expression and PSA, and demonstrated two to three orders of magnitude operation time improvement, along with memory cost comparable to the wildcard expression algorithm, which has the lowest memory consumption within the three existing algorithms.**

I. Introduction

Due to the decouple of control plane and data plane in Software-Defined Networking (SDN), the way policies are defined and enforced is quite different from that of traditional network. The middlebox policy enforcement problem inSDN was first raised by work [1]. Leveraging the policy enforcement process described in this work, we can define three levels of rule in policy enforcement as follows:

- User writes natural semantic forward rules and control rules through the APIs provided by application layer, which enables the user to have a high level management on the network. We define these natural semantic rules as 'user rules';
- Control plane combines the 'user rules' provided by application layer with the configuration information of the network to map the 'user rules' to 'network rules', and 'network rules' can be seen as a simple translation of 'user rules' in the network;
- Control plane combines the 'network rules' with the topology information of the network to dispatch the 'network rules' into the 'operation rules', the 'operation rules' will be assigned to the network nodes in data plane.

According to the summary above, three levels of rules are involved in the process of enforcing policy to network nodes in SDN. These rules should share real-time consistency between each other to make sure that the whole network functions as the user's wish. However, due to the application-independent feature of SDN, together with the occurrence of manual configuration, system reset and other possible incidents, these rules are sometimes inconsistent with each other. So we should periodically check the consistency between them. Since the 'network rules' is a simple mapping of 'user rules' in the control plane, the focus of consistency checking problem should be on checking the consistency between 'network rules' and 'operation rules'. The state-of-art frameworks of policy consistency checking usually consist of two parts: algorithms doing set operation on rulesets consists of rules with priority, and graph algorithms such as reachability checking algorithm to decide the sequence of set operations on rulesets. Since the graph algorithms used in this problem are quite common, our focus of discussion is the rulesets set operation algorithm part.

Existing algorithms of doing set operation on rulesets include Packet Space Analysis tool (PSA) [1], reduced-Binary decision Diagram (r-BDD) [2]. Also in network checking tool Hassel [3], wildcard expression sets are used to represent rulesets and provide basic set operation on them.

The main contributions of this paper are as follows:

- We define the three levels of rules during the policy enforcement process in SDN, and then point out the necessity of policy consistency checking.
- We conclude existing work on policy consistency checking, and theoretically break them down to two stages: algorithms doing set operation on rulesets, and graph algorithms deciding the sequence of set operations on rulesets. We give the mathematical model of policy consistency checking and explain the design goals of

algorithms to solve this problem.
- We devise a new algorithm called AHR to provide basic set operation on rulesets based on rule projection and bitmap index, which is very time and space efficient.
- We use real-life network data set to evaluate the performance of this new algorithm, and we compare the proposed algorithm with three existing state-of-art algorithms in terms of average set operation time cost, memory cost and pre-process time cost.

The rest of this paper is organized as follows. Sec. II introduces related works on rulesets set operation and some related work on policy consistency checking problem. Sec. III gives the model and problem formulation. Sec. IV describes the proposed algorithm. Simulation results of the algorithm and the comparison with existing algorithms will be presented in Sec. V. The main conclusions will be given out in Sec. VI.

II. Related Work

Ever since SDN came into being, policy consistency checking and network properties verification have been a focus of research. There are works like Header Space Analysis (HSA) [3], Atomic Predicates Verifier (AP-Verifier) [4], etc., they provide a general framework for statically or dynamically checking network specifications and configurations. These works can generally be decomposed into two parts: 1) basic representation of policies and rulesets together with the algorithms to do set operations on them; 2) graph algorithms leveraging these set operation algorithms to achieve reachability checking, traffic isolation checking or other purpose. Among these two parts, we can easily see that the first part, algorithms doing set operation on rulesets, is the core of policy checking work, since the graph algorithms are quite common that most of the framework share the same graph algorithms to achieve certain goal. And the algorithms for set operation on rulesets should contain two main part: 1) Give the data structure to represent a ruleset; 2) Give a method to do set operation between this representations.

HSA, together with its further work real-time HSA [5], use wildcard expression sets to represent rulesets, and they do set operation on rulesets by doing set operation on the corresponding wildcard expression sets. AP-Verifier and the packet classification method purposed by Takeru [2] use r-BDD to represent rulesets, and by doing different merge operation between corresponding r-BDDs, this method provide a solution for doing set operation on rulesets. PSA [1], a fast ruleset analysis tool based on the idea of spatial index in database territory, uses hyper rectangles sets to represent rulesets.

To sum up, the mainstream algorithms for doing set operation on rulesets are r-BDD, wildcard expression and PSA. We will use these three algorithms for comparison with proposed algorithm in later part of this paper.

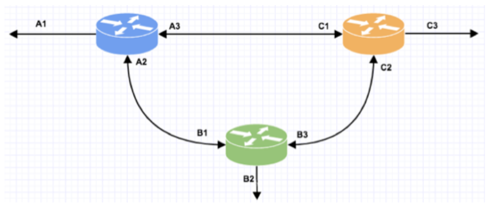
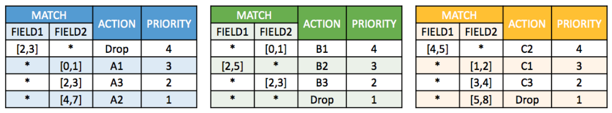


Fig. 1: Topology of the example 3-nodes network



MATCH		ACTION	PRIORITY
FIELD1	FIELD2		
[2,3]	*	Drop	4
*	[0,1]	A1	3
*	[2,3]	A3	2
*	[4,7]	A2	1

MATCH		ACTION	PRIORITY
FIELD1	FIELD2		
*	[0,1]	B1	4
[2,5]	*	B2	3
*	[2,3]	B3	2
*	*	Drop	1

MATCH		ACTION	PRIORITY
FIELD1	FIELD2		
[4,5]	*	C2	4
*	[1,2]	C1	3
*	[3,4]	C3	2
*	[5,8]	Drop	1

Fig. 2: Rulesets of the example 3-nodes network

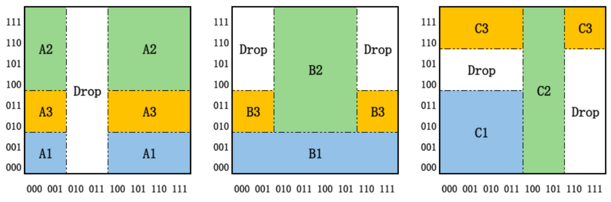


Fig. 3: The packet header space of each node in the example network

III. Model and Problem Formulation

A. *Policy Consistency Checking Model*

Most of the policy consistency checking operation can be generalized to one overall model, that is, to do union and intersect operations on certain network node's ruleset. Since most of the 'network rules' aim at defining reachability among the network (or can be broken down to define reachability), we will take reachability checking as an example, and give a simple illustration on how to perform reachability check between two nodes in the network.

Take the simple 3-nodes network in Figure 1 as an example, each network node in this simple network has one out port connected to the outside network and two inner ports connected with another two nodes in the network. Each nodes has a ruleset which contains several rules with priority. As we can see from Figure 2, each rule contains three part: a matching on the packet header in the form of ranges, corresponding action the node will apply to the matched packets and the priority of the rule.To simplify the example, we assume the network we are discussing to have two fields header, each contains 4 bits. Each rule in ruleset divides the packet header space into some subspaces, as shown in Figure 3. And normally, rules with higher priority will shadow rules with lower priority. Now, suppose we are going to check this network rule: 'Only packet with header [4,5] [2,3] will reach B from A'. We will do as follows:

We will firstly check the topology of the network and get these two path from A to B: $\{A \to B\}$ and $\{A \to C \to B\}$, then, for each path, we will look into the ruleset of each

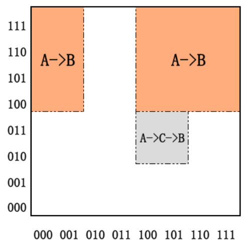


Fig. 4: Packet header set that will reach B from A

node on that path, and do intersect operation between the corresponding subspaces in the packet header space. For example, for path $\{A \to C \to B\}$, we should calculate the intersection of subspace on node A which represents the action of forwarding packet to A3 port and subspace on node C which represents forwarding packet to C2 port. Then, we should do union operation between the results of each path to find out the whole set of packet header that will reach B from A. Figure 4 shows the overall result of this example. As a result, we can see that except for packet with header [4,5] [2,3], there are also other packet that will reach B from A. So the network rule isn't in consistency with the operation rules on network node.

In general, the ruleset on each network node divides the packet header space into different subspaces, and each subspace corresponds to an different action that the network node applies to the packet. Thus we get a 'subspace-action' mapping of this network node. When we are checking some 'network rule' from higher layer, we can do basic set operation between these subspace, and find the 'subspace-network rule' mapping.

B. Problem Formulation

Theoretically, set operation on rulesets, the key to policy consistency checking problem, can be regarded as set operation between hyper rectangle sets in high-dimensional space. In this section, we will give the exact mathematical model of policy consistency checking problem.

Consider a network who has d fields matching in rules, and each fields of the matching consists of $W_i, i = 1, 2, ...d$ bits. Based on this, we define some basic concepts in policy consistency checking as follows:

Network Node, R, intermediate devices in network which have n ports, denoted as $p_1, p_2, ..., p_n$. Network nodes apply different actions to packets passing by them according to the packet's header. Possible actions contain discard, forward to one of its ports, or process(not forward).

Network Packet, P, d-dimensional vector, $P = (p_1, p_2, ..., p_d)$, the value in each dimension represent the value in corresponding field of packet header.

Packet Header Space, S, the set of all the vectors in a d-dimensional space, representing all the possible packet header. $S = \cup_{k=1}^{\infty} P_k(p_1, p_2, ...p_d), \forall p_i$. Since each field consists of $W_i, i = 1, 2, ...d$ bits, S can also be denoted as: $S = [0, 2^{W_1} - 1] \times [0, 2^{W_{[2]}} - 1] \times ... \times [0, 2^{W_d} - 1]$.

Policy Space, Φ, a subspace of packet header space S, which is used to represent the scope of a network rule or a set of operation rules. Policy space Φ_i is usually a set of hyper rectangles in packet header space S, so it can often be represent as $\Phi = \cup_{i=1}^{k} [r_{0_0}^k, r_{0_1}^k] \times [r_{1_0}^k, r_{1_1}^k] \times ... \times [r_{d_0}^k, r_{d_1}^k]$.

Network Rule, A, rules that describe the overall action the network will do to certain packet. Each network rule consists of two parts: $A_i : (\Phi_i, action_i)$, Φ_i stands for the matching of the packet header involved with this rule, and $action_i$ stands for the overall action of packets with these packet header in the network.

Operation Rule Set, F, the 'operation rule' set of every network node, consists of k operation rules $f_1, f_2, ..., f_k$, and each operation rule consists of three parts: $f_i : (match_i, action_i, pri_i)$. $match_i$ is a matching of the packet header, $action_i$ is the action that the network node will apply to matched packet, and pri_i is the priority of this rule in this ruleset. Since different rule may overlap with each other, that is, $m_i \cap m_j \neq \emptyset$, priority is crucial to decide which rule a packet should follow when its header matches multiple rules.

According to the definition above, the policy consistency checking problem can be expressed as follows: given the network topology and the operation rule sets F of every network node R in the network, find the corresponding policy space Φ of network rule A, and then compare it with the corresponding user rule's scope. And the core of this problem is to combine the operation ruleset to find the corresponding policy space, that is, to do set operation on policy space. So, the key to implement efficient policy consistency verification algorithms is to develop efficient algorithms to do set operations on policy space.

C. Design Goals for Set Operation Algorithm on Policy Space

The overall evaluation criteria for policy space set operation algorithm consists of two parts: time cost and space cost. To make clear the design goal for set operation on policy space, we may first analyze the time cost of policy consistency checking problem. The overall time cost of policy consistency checking can be expressed as follows:

$$T_{check} = T_{node\_process} + \sum_{i=1}^{n} (T_{combine}^i + T_{is\_equal}^i)$$

T_{check} is the total time cost to check n network rules, $T_{node\_process}$ is the time cost of pre-processing the operation rule sets. For each static network, pre-processing on operation rule sets only need to be performed once. $T_{combine}^i$ is the time cost of doing set operation on operation rulesets of different nodes, and $T_{is\_equal}^i$ is the time cost of comparing the result policy space with the corresponding user rule scope. In most circumstances, rules in a real-life network wouldn't change frequently, so we should focus on decreasing $T_{combine}^i$ and $T_{is\_equal}^i$.

The memory cost of this problem is the data structure used to represent policy space. So to find a space-saving data structure is also an important design goal of algorithms to solve this problem.

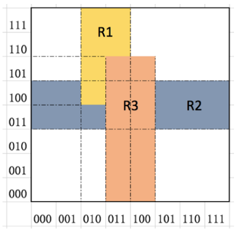
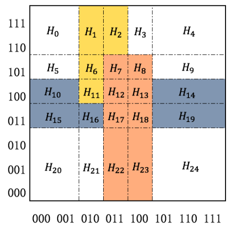


Fig. 5: Rule projection



111
110
101
100
011
010
001
000
H_0 H_1 H_2 H_3 H_4
H_5 H_6 H_7 H_8 H_9
H_{10} H_{11} H_{12} H_{13} H_{14}
H_{15} H_{16} H_{17} H_{18} H_{19}
H_{20} H_{21} H_{22} H_{23} H_{24}
000 001 010 011 100 101 110 111

Fig. 6: Packet header space atomization

This paper proposes a algorithm which demonstrates two to three orders of magnitude improvement on $T^i_{combine}$ and $T^i_{is\_equal}$ with a reasonable memory cost by carefully pre-processing the operation rulesets. This algorithm strikes a better balance between pre-processing time and set operation time to fit the special needs of policy consistency checking. The following section will explain this algorithm in detail.

IV. Atomic Hyper Rect Index Algorithm

The proposed Atomic Hyper Rect Index algorithm (AHR in following contents) is based on rule projection and bitmap index. First we use rule projection to atomize the packet header space, then we use bitmap to represent rules and policy space, and provide solution to set operation between them. In this section, the concept of rule projection and bitmap index will be explained, the realization of AHR algorithm will be given.

A. Rule Projection

Rule projection is a common idea in some network packet classification algorithms such as HyperSplit [6] and so on. It projects rules in a direction perpendicular to the coordinate axis in packet header space, dividing each dimension of the packet header space into many intervals. In this way different rules are separated. An example of rule projection is give in Figure 5. There are two dimensions in the illustrated packet header space, and the figure shows three overlapping operation rules with priority and a default rule. By projecting each operation rule along the horizontal axis and the vertical axis, we divide each axis into five intervals, so as to prepare for the next step for AHR algorithm.

B. Packet Header Space Atomization

After rule projection, we can atomize the packet header space according to the intervals get from rule projection. In this way, we divide the entire packet header space into many atomic hyper-rectangles. Because we have cut the packet header space to the finest granularity, the effect of rule overlapping is eliminated, and every single rule can be represented as a set of several hyper-rectangles.

As an example, the two-dimensional packet header space in Figure 4 will be cut into 25 atomic hyper-rectangles (rectangles in this case) after this process, as shown in Figure 6, and we can easily find that the original overlapping rules can now be represented by different set of atomic rectangles separately. For example, the grey rule can now be represent by set $\{H_{10} \cup H_{15} \cup H_{16} \cup H_{14} \cup H_{19}\}$.

C. Bitmap Index

Bitmap index is a commonly used type of index in recent years. It is most appropriate for columns having low distinct values, and demonstrate very fast query operation and set operation. In this paper, we encode the atomic hyper-rectangles in the packet header space to create a one-to-one mapping between hyper-rectangles and bits in a bit string. Every bit in this bit string serves as an index for corresponding hyper-rectangle, and if a rule's matching (or a policy space) contains this rectangle, the corresponding bit in bit string of this rule(or policy space) is set to 'True' and vice versa. In this way, we turn policy space and set operation between policy spaces into bit string and set operation between bit string. As the set operation between bit string is very fast, we can ensure an extremely high efficiency of set operations between policy spaces using this method. Again, in the example showed by Figure 6, we can map the policy space of the grey rule into bits string '0000000000100011100100000', and the policy space of the orange rules can be represented as '0000000110001100011000110'.

On the other hand, because we use one overall atomic hyper-rectangle set to represent every rule, we only have to record these atomic hyper-rectangles once, and then use memory saving bit string to represent each rule. In this way, we can avoid the heavy use of memory.

D. Atomic HyperRect Index Algorithm

The main idea of AHR algorithm is as follows: first use rule projection and packet header space atomization to pre-process the operation rulesets, then encode the atomic hyper-rectangles and generate the corresponding bit string of each rule. After translating rules into bit strings, we can easily do set operation between rules.

The detailed process of pro-processing a ruleset can be described as follows:

- First of all, project every rule's matching in the ruleset F along each dimension in the packet header space, and by this we get a set of projection points in every dimension, denoted as $\{Pt'_d[i], 1 \leq i \leq M'_d\}, d = 1, 2, ..., DIM\_MAX$, where M'_d is the number of projection points in dimension d.
- Then for each dimension d, we remove the duplicate points in Pt'_d. After this we sort the de-duplicate point set, by this we get the final set of points, denoted as $\{Pt_d[i], 1 \leq i \leq M_d\}, d = 1, 2, ..., DIM\_MAX$. And these points divide each dimension of packet header space into several segments, denoted as $\{Sg_d[i], 1 \leq i \leq M_d - 1\}, d = 1, 2, ..., DIM\_MAX$. By cutting the overall packet header space by the projection points in every dimension, we divide the packet header space into $\prod_{d=1}^{DIM\_MAX} len(Sg_d)$ atomic hyper-rectangles. And we encode these rectangles from 0 to $\prod_{d=1}^{DIM\_MAX} len(Sg_d) - 1$.

- Finally, we create a bit string $\xi_1\xi_2...\xi_N$ of length $\prod_{d=1}^{DIM\_MAX} len(Sg_d)$ for each rule in the ruleset, and set each bit's value according to the geometric relationships between the corresponding atomic hyper-rectangle and the rule's policy space.

After pre-processing, if we want to do set operation between, say, two rules R_1 and R_2, given their bit string $R_1 = \xi_1\xi_2...\xi_N$, $R_2 = \xi_1^{'}\xi_2^{'}...\xi_N^{'}$, we have: $R_1 \Delta R_2 = \xi_1\xi_2...\xi_N \Delta \xi_1^{'}\xi_2^{'}...\xi_N^{'}$, where $\Delta = |, \&, xor$.

Borrowing the 'atomic predicates' idea from AP-Verifier [4], when we are verifying the consistency between network rules and operation rules in practice, we can use the algorithm explained in Algorithm1 to construct a bit string for each distinct action to packet of one router based on its ruleset. In this way, we can turn the ruleset of an router with n actions for packets into n predicates. Then, it's even convenient to combine the corresponding predicates on path which we are checking to verify some network rules. To simplify the explanation of predicate constructing algorithm, we can define a function $set(BitString, matching, 0/1)$ to set the corresponding bits of atomic hyper-rectangles in $matching$ to be 0/1. Let the number of rules in the router's ruleset be n, the number of all possible actions to packets of the router be m, the algorithm is shown in Algorithm1.

Algorithm 1 Construct the predicate of a router

```
Input: Ruleset $F\{(match_i, action_i, pri_i)\}, i = 1, 2, ..., n$
Output: Bit string dict ActP:$\{action_j : bitstring_j\}, j = 1, 2, ..., m$
1:  function PredicateConstruct($F : list(f_j)$)
2:      Sort $F$ by $pri_i$ in increasing order
3:      for $j = 1 \to m$ do
4:          $ActP[action_j] \leftarrow$ '000...0'
5:      end for
6:      for $f_i \in F$ do
7:          for $j = 1 \to m$ do
8:              $set(ActP[action_j], f_i[match], 0)$
9:          end for
10:         $set(ActP[f_i[action]], f_i[match], 1)$
11:     end for
12:     return $ActP$
13: end function
```

V. Evaluation

In this section, we evaluate the proposed algorithm against three existing policy space set operation algorithms: wildcard-expression algorithm [3], PSA [1] and r-BDD algorithm [2], then compare their performance in three aspects: average operation time, memory cost and pre-processing time. AHR algorithm demonstrates two to three orders of magnitude improvement on operation time, along with memory space comparable to the wildcard expression algorithm, which has the lowest memory consumption within the three existing algorithms. But to achieve this improvement, AHR's pre-processing time is two orders of magnitude higher than other algorithms.

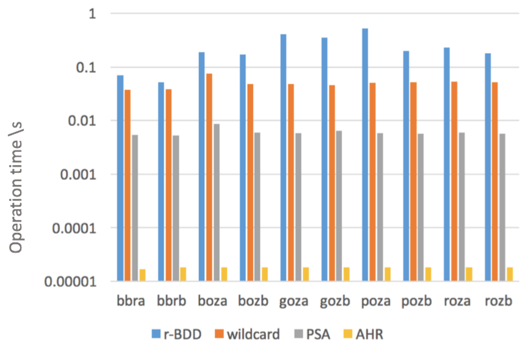


Fig. 7: Average operation time

We carry out the experiment on Stanford backbone network [3], a real-life data set used by HSA to test its efficiency. This data set contains the raw data acquired from the Stanford campus backbone network. It consists of the routing tables, access control lists and configuration lists of 16 Juniper routers.

We use the raw data parser provided by Hassel tool [7] to parse the raw data into prefix form. For each algorithm, we first pre-process the ruleset of each router, and then by randomly choosing a sequence of rules to do random set operations, we test the operation time, memory cost and pre-processing time of these four algorithms. And the results are showed in following paragraphs.

The physical configuration of the machine on which we carry out our test on these algorithms is as follows: CPU:Intel(R) Core(TM) i7-3770 CPU @3.40GHz, MEM:16G, OS:Ubuntu 12.04.4 LTS.

Figure 7 shows the average operation time of doing 500 rounds of set operations on each router's ruleset by each algorithm. We can see from the result that the operation time of AHR is much more lower than all other three algorithm. This is reasonable because AHR algorithm adopts the most fine-grained cutting on packet header space, so that it turns the set operation of rules into set operations of bit strings, which is incredibly fast compared with other three algorithm. The PSA tool is also faster than the remaining two algorithm, which has to be attributed to its reference to the clipping region management in computer graphics field [1]. As for the remaining two algorithms: r-BDD and wildcard-expression, the reason of their failure to compete with AHR or PSA is as follows: 1) The set operation itself is quite complicated. For r-BDD, it has to recursively calculate on each nodes in the diagram, and for wildcard-expression, it has to do set operation between each wildcard expression in two sets of wildcard expressions. 2) The data structure of these two algorithms will get more and more complex as the number of operations performed increases, and the time cost of doing set operation on such data will increase simultaneously.

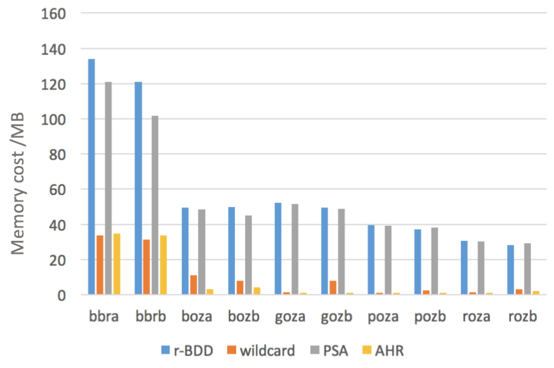


Fig. 8: Memory Usage

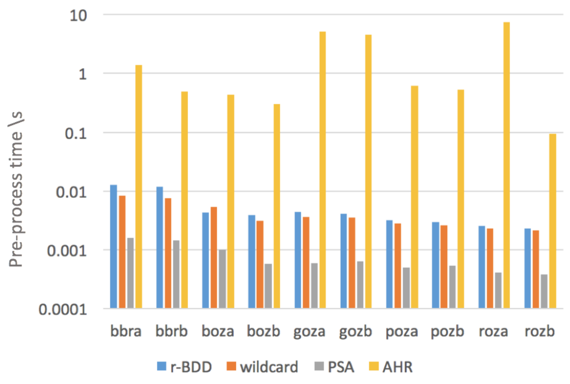


Fig. 9: Pre-process time

Another thing we must mention is that, among these four algorithms, AHR is the only algorithm that avoid the ‘cumulative effect’ in intensive set operation tests. That is, when we preform intensive set operations continuously, the size of the data structure used by other three algorithms to hold the result of these set operations will grow rapidly. As a result, algorithms who have ‘cumulative effect’ will have increasing average operation time cost when doing intensive set operations. In detail, the node number in r-BDD’s decision diagram will grow when the policy space of the result gets complex, and the number of wildcard expression used to represent the result will also grow, so as the number of hyperrects in PSA. But as for AHR algorithm, because we have already divide the packet header space into the smallest subspaces, however complex the result is, we can still use a fixed length bit string to represent it, and the time cost of doing set operation on it will not change.

Figure 8 shows the memory cost of each algorithm to deal with each router’s ruleset. We can see that AHR algorithm has a memory cost comparable to the wildcard expression algorithm, and these two algorithms have the lowest memory cost among four algorithms. This is reasonable because in AHR, we only need to store the projection points set $\{Pt_d[i], 1 \leq i \leq M_d\}, d = 1, 2, ..., DIM\_MAX$, and the bit string of each rule, and in wildcard expression algorithm, each rule will be represented by a set of wildcard expressions composed by ‘1’, ‘0’ and ‘X’. But in PSA and r-BDD, rule will be represented by a set of hyperrects and a decision graph.

Figure 9 shows the pre-processing time of each algorithm to deal with each router’s ruleset. We can see that among these four algorithms, AHR has the longest pre-processing time. This is because that by doing the most fine-grained cutting on the packet header space, AHR needs to generate too many hyper-rectangles, and then decide the value of each bit of each rule’s bit string, which sharply increases the amount of computation in pre-processing. However, due to the special application scenario of network that the rules in network won’t change frequently, so relative longer pre-processing time is still acceptable as long as we can do set operation fast.

VI. Conclusion

We define the three levels of rules during the policy enforcement process in SDN for the first time, then state the necessity of policy consistency checking. We give a mathematical model of policy consistency checking and explain the design goals of this problem.

We present a new algorithm leveraging rule projection and bitmap index to provide basic set operation on packet header space, which is much more time efficient than existing algorithms. We evaluate the performance of this new algorithm using a real life data set, and compare it with other three algorithms, r-BDD, wildcard expression and PSA. As a result, our algorithm, AHR, demonstrate two to three orders of magnitude operation time improvement, along with memory cost comparable to the wildcard expression algorithm, which has the lowest memory consumption within the three existing algorithms. Although AHR needs two orders of magnitude pre-processing time than other three algorithms, it’s still a better trade-off between operation time and pre-processing time, because for multiple policy consistency check operations, we only have to pre-process the rulesets for once as long as the configuration of the network doesn’t change. So this work shield lights on implementing faster policy consistency checking framework, and can also be used as the basic algorithm for network properties verification.

Acknowledgment

This work was supported by the National Key Technology R&D Program of China under Grant No. 2015BAK34B00.

References

[1] X. Wang, W. Shi, Y. Xiang, and J. Li, “Efficient network security policy enforcement with policy space analysis.”

[2] T. Inoue, T. Mano, K. Mizutani, S.-I. Minato, and O. Akashi, “Rethinking packet classification for global network view of software-defined networking,” in *2014 IEEE 22nd International Conference on Network Protocols*. IEEE, 2014, pp. 296–307.

[3] P. Kazemian, G. Varghese, and N. McKeown, “Header space analysis: Static checking for networks,” in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 113–126.

[4] H. Yang and S. S. Lam, “Real-time verification of network properties using atomic predicates,” *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, 2016.

[5] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real time network policy checking using header space analysis,” in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 99–111.

[6] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, “Packet classification algorithms: From theory to practice,” in *INFOCOM 2009, IEEE*. IEEE, 2009, pp. 648–656.

[7] “Header space library (hassel),” http:/stanford.edu/œkazemian/hassel.tar.gz.