AHI: Efficient Policy Space Set Operations

Danyang Li*†, Xiaohe Hu*†, Linli Wan*, Zhi Liu*†, Jun Li*†‡
* Department of Automation, Tsinghua University, Beijing, China
† Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, China
‡ Tsinghua National Lab for Information Science and Technology (TNList), Tsinghua University, Beijing, China
{lidangyan16, hu-xh14, wan-ll13, zhi-liu12}@mails.tsinghua.edu.cn, junl@tsinghua.edu.cn

***Abstract*—With the fast industrial deployment of software-defined networking (SDN) and network function virtualization (NFV) technologies, network function policy enforcement in large scale virtualized networks becomes a key challenge for network management. Distributed policy enforcement heavily involves network policy space analysis, where set operations consume most of the computation. Based on spatial projection and bitmap indexing, a novel algorithm AHI (Atomic Hyper-Rectangle Indexing) is proposed for fast policy space set operations. Experiments with real datasets demonstrated that AHI improves set operation speed by two to three orders of magnitude and achieves the same least space cost, comparing to existing state-of-the-art algorithms r-BDD, wildcard expression, and PSA.**

***Index Terms*—Policy Enforcement; Policy Space; Set Operation**

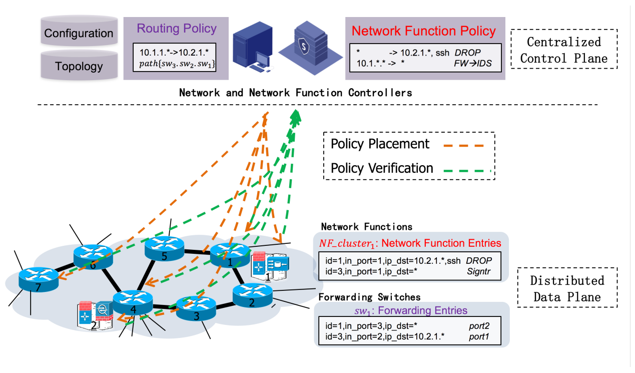
Configuration
Topology
Routing Policy
10.1.1.*->10.2.1.*
$path\{sw_3.sw_2.sw_1\}$
Network Function Policy
* -> 10.2.1.*, ssh DROP
10.1.*.* -> * FW→IDS
Centralized Control Plane
Network and Network Function Controllers
Policy Placement
Policy Verification
Network Functions
$NF\_cluster_1$: Network Function Entries
id=1,in_port=1,ip_dst=10.2.1.*,ssh DROP
id=3,in_port=1,ip_dst=* Signtr
Forwarding Switches
sw_1: Forwarding Entries
id=1,in_port=3,ip_dst=* port2
id=3,in_port=2,ip_dst=10.2.1.* port1
Distributed Data Plane
1
2
3
4
5
6
7

Fig. 1. Network Function Policy Enforcement Architecture

I. Introduction

Facing the great demands for network agility, software-defined networking (SDN) is rapidly adopted by the industry, followed by network function virtualization (NFV). Decoupling the control plane and the data plane, SDN exposes logically centralized control interfaces for network operators to write policies and deploy applications. Furthermore, SDN incorporating NFV forms a flexible and scalable framework to provide network function services [1]–[3], *e.g.*, Access Control, Firewall, IDS, WAN Optimization, for end users. However, in this new architecture, network function policy enforcement becomes heavy-loaded, distributed and much more complicated than traditional middlebox policy configuration, which was usually done by hands.

A. Network Function Policy Enforcement

The network function policy enforcement architecture and its key parts are depicted in Fig. 1. In the control plane, one *network function policy* is a set of policy entries that specifies an end-to-end packet header set, represented by the term *policy space* in this paper, and its corresponding actions, the network function symbols, *e.g.* DROP, Firewall, IDS, or chains of these functions. The control plane *routing policy* also states a policy space and its connectivity properties. Network function policies and routing policies of the control plane are mapped to comprehensive *network function entries* and *forwarding entries* of the data plane, which contain additional local information such as the *in_port* and *out_port*. *Network function policy enforcement* is the mapping process of the control plane network function policies and the data plane network function entries. The typical components of the policy enforcement are *policy placement* and *policy verification*.

On policy placement, the global network function policies are partitioned by the controller and placed to corresponding network nodes considering the traffic distribution decided by routing policies and network function locations in the topology, aiming to achieve load-balance, replication minimization, or other objectives. Existing policy placement methods are classified to two groups, one is to place policies while changing routing, *e.g.*, DIFANE [4] and vCRIB [5], and the other is to place policies with respecting routing, *e.g.*, Palette [6], One-Big-Switch [7] and MBPE [8].

On the other hand, periodical policy verification is needed to make sure that the entries in the data plane nodes are consistent with the original policies in the control plane, as some incidents such as system resets or manual configuration mistakes might happen. By combining the forwarding entries on paths and the network function entries on the related nodes, policy verification generates the splicing policies, then checks the consistency between the splicing policies and the original network function policies. Existing work of policy verification includes Header Space Analysis (HSA) [9], Net Plumber [10], and Atomic Predicates Verifier (AP-Verifier) [11].

B. Policy Space Set Operation

Both the partition-then-assign process of policy placement and the combine-then-compare process of policy verification heavily involve network policy analysis, and most of the analysis relies on policy space set operations such as intersection, union, and difference. For example, the policy spaces of network function policies should be intersected with the policy spaces of relevant routing policies to heuristically choose efficient policy placement nodes [8], and the policy space of one global network function policy should be compared to the union of the policy spaces of its corresponding network function entries in data plane to verify if the policy is correctly placed [9]. Therefore, the theoretical and practical mechanism to conduct set operations of policy spaces lays a foundation for policy enforcement in future networks.

However, realizing the policy space set operation is a challenging problem. First, given that the policy space of a network function policy entry can be a set of discrete sub-spaces for multi-dimensional packet headers, it is complex to organize these sub-spaces. Second, since arbitrary overlaps frequently occur between sub-spaces of two policy entries, set operations on those intermittent multi-dimensional sub-spaces is highly computation-intensive [11]. Third, as the amount of the policies and the size of packet header dimensions grow exponentially with the increasing diversity of SDN/NFV deployment scenarios and protocols, the scalability of the set operations also requires critical algorithm design.

As a consequence, it is urgent and crucial to analyze existing methods and develop a more efficient and scalable policy space set operation mechanism. This paper proposes a novel algorithm of the policy space set operation, AHI (Atomic Hyper-Rectangle Indexing), which provides an efficient data structure to represent the policy space and the basal set operations on this structure to support complex policy analysis. By carefully pre-processing policy spaces leveraging the mechanisms of spatial projection and bitmap indexing, AHI performs operations faster than previous work by two or three orders of magnitude while maintaining the same least space cost.

C. Main contributions and Organization of this Paper

The main contributions of this paper are:

- Conclude policy space set operation as a foundation component in policy enforcement in SDN/NFV, then present the mathematical model of the problem.
- Devise a time and space efficient algorithm called AHI to provide basal policy space set operations for policy analysis.
- Use real-life network datasets to evaluate the performance of the proposed algorithm, comparing with three existing state-of-the-art algorithms in terms of average set operation time and memory usage.

The rest of this paper is organized as follows. Sec. II gives the model of the basic concepts and formulates the policy space set operation problem. Sec. III introduces the policy space set operation methods used by existing state-of-the-art policy placement and verification works. Sec. IV elaborates the algorithm design of AHI. Experiment results of AHI and existing algorithms are presented in Sec. V. Sec. VI concludes this paper and shows the future work.

II. Problem Statement

In this section, the model primitives in policy analysis are first presented from a computation geometry perspective, then the policy space set operation problem is formulated based the given model.

As shown in Fig. 1, policies in the SDN/NFV scenario have different variants in terms of the control plane and the data plane. To conduct a universal analysis, this paper unifies the policy analysis objects as policy (set) and entry (element):

Entry consists of two parts, a set of consecutive packet header values and its corresponding action.

Policy is equal to a set of numbered entries. The ranking of each entry in the policy represents the entry priority. When two entries overlap with each other, *i.e.* containing the same part of packet header values, and the corresponding actions conflict, the entry with the higher priority(smaller ranking) effects.

Both of entry and policy specify a set of packet header value and corresponding actions, and the set operations are ultimately executed on the packet header sets[1].

A. Model Primitives

Consider a network with d fields in the packet header, and each field consists of $W_i, i = 1, 2, ...d$ bits. Leveraging the computation geometry theory, the model of the primitives are indicated as follows:

Packet Space, $P = (p_1, p_2, ..., p_d)$, a d-dimensional vector. $p_i, i = 1, 2, ..., d$ represents the value of packet header in the corresponding field. A packet space P stands for a point in the d-dimensional geometric space.

Universe Space, $U = \{P = (p_1, p_2, ..., p_d) \mid 0 \leq p_i \leq 2^{W_i - 1}, \forall i \in \{1, 2, ..., d\}\}$, the universal set of all the packet vectors. The universe space is the whole d-dimensional geometric area.

Entry Space, $E = \{P = (p_1, p_2, ..., p_d) \mid t_i \leq p_i \leq k_i, \forall i \in \{1, 2, ..., d\}\}$, $t_i, k_i, i = 1, 2, ..., d$ are constants. An entry is set of consecutive packet vectors and represents a hyper-rectangle in the universal space.

Policy Space, $S = \cup_{i=1}^{n} E_i$. One policy is a union of n arbitrary entries. Also, one policy stands for a union of n arbitrary hyper-rectangles $\cup_{i=1}^{n} E_i$, which can be converted to a union of m non-overlapped hyper-rectangles $\cup_{i=1}^{m} E'_i$, where $m \geq n$. To keep the priority semantic, each entry space holds an index as its priority, and the converted entry spaces also hold the corresponding original priority. The overlapped sub entry space inherits the highest priority of the original entry spaces.

[1]This paper doesn't contain operations on actions of the same packet header, such as combining different security network functions to enhance the shield. Recent research works, *e.g.* PGA [12], model the service function chain operations, which can be complementary to this work.

B. Problem Formulation

Given the mathematical model and the computation geometry view, the following formulation of the policy space set operation problem is easy to understand.

First, the set operation of the entry space can be viewed as geometrical union, intersection and difference of two hyper-rectangles. For entry space $E = \{P = (p_1, p_2, ..., p_d) \mid t_i \leq p_i \leq k_i, \forall i \in \{1, 2, ..., d\}\}$ and entry space $E' = \{P = (p_1, p_2, ..., p_d) \mid t'_i \leq p_i \leq k'_i, \forall i \in \{1, 2, ..., d\}\}$, the union operation denotes $E \cup E' = \{P = (p_1, p_2, ..., p_d) \mid t_i \leq p_i \leq k_i, \forall i \in \{1, 2, ..., d\}$ or $t'_i \leq p_i \leq k'_i, \forall i \in \{1, 2, ..., d\}\}$, and the intersection operation is $E \cap E' = P = (p_1, p_2, ..., p_d) \mid \max\{t_i, t'_i\} \leq p_i \leq \min\{k_i, k'_i\}, \forall i \in \{1, 2, ..., d\}\}$, and the difference operation indicates if $E \cap E' = \emptyset$, then $E \setminus E' = E$, else $E \setminus E' = E \setminus (E \cap E') = \{P = (p_1, p_2, ..., p_d) \mid t_i \leq p_i \leq k_i,$ and $p_i < t''_i,$ and $p_i > k''_i, \forall i \in \{1, 2, ..., d\}\}$.

Then, the set operation of two policy spaces is equal to multiple set operations of the entry spaces of two policy spaces. For policy space $S = \cup_{i=1}^{n} E_i$ and policy space $S' = \cup_{i=1}^{m} E'_i$, the union operation represents $S \cup S' = (\cup_{i=1}^{n} E_i) \cup (\cup_{i=1}^{m} E'_i)$, and the intersection operation indicates $S \cap S' = (\cup_{i=1}^{n} E_i) \cap (\cup_{i=1}^{m} E'_i)$, and the difference operation is $S \setminus S' = (\cup_{i=1}^{n} E_i) \setminus (\cup_{i=1}^{m} E'_i)$.

III. Related Work

Ever since SDN and NFV came into being, policy enforcement has been a focus of research. As introduced in the primary section, various frameworks are proposed to realize policy placement or policy verification. Each of these frameworks has its own methods for conducting basal policy space set operations. This section summarizes and analyzes the used operation mechanisms.

Policy space set operation algorithms used by existing policy placement and verification frameworks can be summed up into the following three categories:

1) **Decision Diagram:** This kind of algorithms translate policy spaces into decision diagrams where each bit of packet header serves as a decision node, and provide set operations of policy spaces based on this representation. Since the size of decision diagram grows exponentially with the number of bit in packet header, algorithms such as [13] [14] adopt reduction or aggregation process to improve the scalability. AP-Verifier uses reduced Binary Decision Diagram (r-BDD) [15] to represent policy space. Due to the reduce process in the construction and operation of r-BDD, which removes the duplicated branches of the original diagram, the scale of the diagram is sharply reduced, and the time and memory cost of r-BDD is respectively lower.
2) **Wildcard expression:** In this category, policy spaces are represented by lists of wildcard expressions. The masks of packet header can be easily transformed into wildcard expressions, which consist of '0', '1' and 'X'. Set operations of policy space are converted into set operation wildcard expression lists. HSA and its further work Net Plumber, use this method as the basis of their design. Although the representations and operations on wildcard expressions are simple, operations on two wildcard expression can result in multiple expressions and the linear operations on expression lists are inefficient.
3) **Decision Tree:** PSA [8] constructs a decision tree structure R-Tree to realize fast policy space set operation. Based on range represented hyper-rectangles and spatial index borrowed from database territory, PSA leverages a set of non-overlapping hyper-rectangles indexed by R-Tree to represent the policy space, and implements set operations with clipping region management methods. PSA is used in MBPE to conduct efficient partitions on control plane policy spaces.

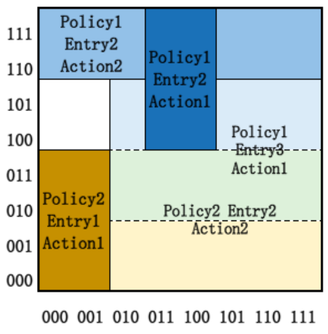

Fig. 2. Policies

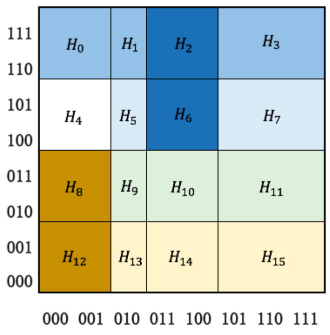

Fig. 3. Atomization

All the above set operation algorithms apply set operations on policy spaces with less and inefficient indexing design. Given that the preparation of policy changes in policy enforcement is usually at human time scales, it is practical to add further pre-processing strategies on relevant policy spaces to accelerate the intensive set operations. Although it is a trade-off between pre-processing time and operation time, a substantial improvement of the set operation time can be achieved with reasonable pre-processing cost by careful and efficient design. This paper proposes an algorithm with more efficient indexing structure design to accelerate the set operations. The details are explained in the next section.

IV. Atomic Hyper-rectangle Indexing Algorithm

The proposed AHI algorithm constructs the policy space indexing structure based on spatial projection and bitmap indexing and provides digestible and fast set operations benefiting from the indexing structure. In this section, the concept of spatial projection and bitmap indexing are explained, followed by the elaboration of AHI design and implementation.

A. Spatial Projection and Universal Space Atomization

Spatial projection is a common idea in some network packet classification algorithms such as HyperSplit [16] and so on. It projects entries in a direction perpendicular to the coordinate axis in universal space, dividing each dimension of universal

Algorithm 1 Universal Space Atomization

```
1: function SpaceAtomization($S\{S_j = \cup_{i=1}^{n_j} E_i^j\}, j = 1, 2, ..., m$)
2:     Let $Pt[d]$ and $Seg[d], d = 1, ..., D$ be lists
3:     for $S_j \in S$ do
4:         for $E_j^i \in S_j$ do
5:             for $d = 1 \rightarrow D$ do
6:                 Add end points of range $E_j^i[hyper-rectangle][d]$ to $Pt[d]$
7:     for $d = 1 \rightarrow D$ do
8:         De-duplicate list $Pt[d]$
9:         Sort $Pt[d]$ in increasing order
10:        for $i = 0 \rightarrow len(Pt[d]) - 1$ do
11:            $Seg[d].append(range([Pt[d][i], Pt[d][i+1]))$
12:    return $Seg[d], d = 1, ..., D$
```

Algorithm 2 Construct Bit Vector

```
1: function BitVectorCon($S\{S_j = \cup_{i=1}^{n_j} E_i^j\}, j = 1, 2, ..., m, Seg[d], d = 1, ..., D$)
2:     Let $BV_j, j = 1, 2, ...m$ be dictionaries
3:     for $S_j \in S$ do
4:         Sort $E_i^j$ by $pri_i^j$ in increasing order
5:         let $ActionSet$ be an empty set
6:         for $E_i^j \in S_j$ do
7:             Add $E_i^j[act]$ to $ActionSet$
8:         De-duplicate $ActionSet$
9:         for $act \in ActionSet$ do
10:            $BV_j[act] \leftarrow \underbrace{0000000...0}_{\text{number of 0s:}\prod_{d=1}^{D}(len(Seg[d]))}$
11:        for $E_i^j \in S_j$ do
12:            $IndexSet_i^j \leftarrow index$ of atomic hyper-rectangles contained in $E_i^j[hyper-rectangle]$
13:            for $act \in ActionSet$ do
14:                $BV_j[act][IndexSet_i^j] \leftarrow 0$
15:            $BV_j[E_i^j[action]][IndexSet_i^j] \leftarrow 1)$
16:    return $BV_j, j = 1, 2, ..., m$
```

space into many intervals. After spatial projection, whole universal space can then be atomized by cutting it along each interval get from the former process. In this way, universal space is divided into many atomic hyper-rectangles. Since universal space has been cut to the finest granularity, the effect of space overlapping is eliminated, and every single entry space or policy space can now be represented as a set of several atomic hyper-rectangles.

An example of spatial projection and universal space atomization is given in Fig. 2. There are two dimensions in the illustrated universal space, and the figure shows two overlapping policy spaces, *Policy*1 with 3 entries and *Policy*2 with 2 entries. By projecting each entry of each policy along the horizontal axis and the vertical axis, the universal space is divided into 16 atomic hyper-rectangles (rectangles in this case), as shown in Fig. 3. In this case, it's easy to conclude that the original overlapping policy spaces can now be represented by different sets of atomic rectangles separately. For example, the policy space of the blue policy can now be represented by set $\{H_0 \cup H_1 \cup H_2 \cup H_3 \cup H_5 \cup H_6 \cup H_7 \cup H_9 \cup H_{10} \cup H_{11}\}$. And if divided by different *action*, the policy space of the blue policy can be divided into two sub policy spaces: *action*1's policy space can be represented by set $\{H_2 \cup H_5 \cup H_6 \cup H_7 \cup H_9 \cup H_{10} \cup H_{11}\}$, and *action*2's policy space can be represented by set $\{H_0 \cup H_1 \cup H_3\}$.

B. Bitmap Indexing

After the universal space atomization, a concise representation of atomic hyper-rectangle set is needed. Bitmap indexing is a commonly used type of indexing in recent years, which is the perfect choice for representing atomic hyper-rectangle set. On the other hand, bitmap indexing is most appropriate for columns having low distinct values (in this scenario only '1' and '0' to represent 'contain' and 'not contain') and demonstrate very fast query operation and set operation, which is exactly what we need for implementing fast policy space set operations.

In this paper, the atomic hyper-rectangles of the universal space are encoded to create a one-to-one mapping between hyper-rectangles and bits in a bit vector. Every bit in this bit vector serves as an index for corresponding hyper-rectangle, and if a policy space contains one atomic hyper-rectangle, the corresponding bit in bit vector of this policy space is set to '1' and vice versa. In this way, policy spaces are transformed into bit vectors, and set operation between them can be done by set operation between bit vectors. As bit vector set operation is very fast, an extremely high efficiency of set operations between policy spaces can be reached using this method. On the other hand, since one overall atomic hyper-rectangle set is used to represent every policy space, these atomic hyper-rectangles have to be recorded for only once. In addition, the bit vector representation of policy space itself is memory saving enough, which serves as a double insurance for space efficiency.

Continue with the example shown before, by encoding the atomic hyper-rectangle set from 0 to 15, each policy space can be represented by a bit vector. In Fig. 3, the blue policy's policy space can be mapped into bit vector '1111011101110000'.

C. Indexing construction on policy space in AHI

Leveraging spatial projection and bitmap indexing, AHI first pre-processes policy space. Basically, pre-processing can be divided into two parts: Universal Space Atomization and Bit Vector Construction. The pseudo code of each part is shown in Algorithm 1 and 2.

In Universal Space Atomization, AHI projects the entry $E_i^j : (Hyper-rectangles, action, priority)$ of each policy along every dimension d in the universal space U to get the interval lists $Seg[d], d = 1, 2, ..., D$ on each axis. These intervals divide U into $\prod_{d=1}^{D}(len(Seg[d]))$ atomic hyper-rectangles, and these atomic hyper-rectangles can be encoded by dimensions. In detail, the index of the atomic hyper-rectangles formed by $Seg[1][i_1] \times Seg[2][i_2] \times ... \times Seg[D][i_D]$ can be calculated as $\sum_{d=1}^{D}(\prod_{k=1}^{d-1} len(Seg[k]) \cdot i_k + i_k)$.

Then in Bit Vector Construction part, for each policy space S_j, AHI sorts its entry E_i^j by priority in increasing order. Then for each unique action of one policy, AHI creates a bit vector of $\prod_{d=1}^{D}(len(Seg[d]))$ bits, all initialized to 0. Then for each entry E_i^j in the sorted entry list of policy S_j, AHI finds the corresponding index set $IndexSet_i^j$ of the atomic hyper-rectangles contains by $E_i^j[hyper{-}rectangles]$. After this, firstly AHI sets the bits with indexes in $IndexSet_i^j$ of every action's bit vector in S_j to be 0, secondly AHI sets the bits in $IndexSet_i^j$ of $E_i^j[action]$'s bit vector to be 1. In this way, AHI generates a bit vector for each unique action in each policy.

D. Set Operations on policy space in AHI

After pre-processing, set operation between policy space can be straight-forward enough. Since the bit vector set BV_j of policy S_j can be generated by the previous described pre-processing algorithm, which contains the policy space of each unique action of S_j, the policy space denoted by bit vector of S_j can be calculated by doing union operation on each action's bit vector: $BitVec_j = \cup_k BV_j[action_k]$. Then set operation between policy S_{j1} and S_{j2} can be calculated by $BitVec_{j1} \Delta BitVec_{j2}$, where $\Delta = \cup, \cap, \backslash$.

However, in policy analysis, set operation between policy spaces of different actions of policies is performed much more frequently than it between policy spaces of policies. In this case, set operation is even more simple: just apply the set operation on the corresponding bit vectors of related actions. This simplicity guarantees the efficiency of AHI algorithm.

V. AHI Evaluation

In this section, the proposed algorithm AHI is evaluated against three existing policy space set operation algorithms: r-BDD, wildcard-expression, and PSA. Their performances are compared in two aspects: average operation time, memory cost. AHI demonstrates two to three orders of magnitude improvement on operation time, along with memory space comparable to wildcard expression, which has the lowest memory consumption among the three existing algorithms.However, to achieve this improvement, the AHI pre-processing time of current implementation version is higher than other algorithms, which is being optimized in the on-going work.

The experiment was carried out on the Stanford backbone network [9], a real-life data set used by HSA to test its efficiency. This data set contains the raw data acquired from 16 Juniper routers in the Stanford campus backbone network. It consists of the routing tables, access control lists, and configuration lists. The physical configuration of the experiment machine on which the tests of these algorithm run is as

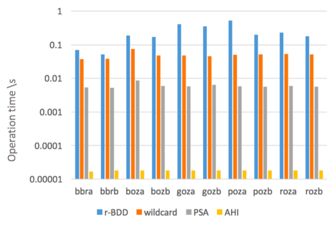

Fig. 4. Average operation time

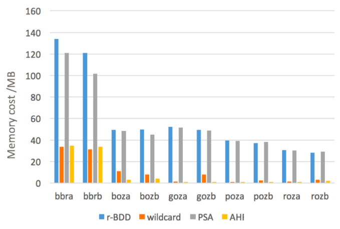

Fig. 5. Memory Usage

follows: CPU: Intel(R) Core(TM) i7-3770 CPU @3.40GHz, MEM: 16G, OS: Ubuntu 12.04.4 LTS.

The evaluation uses the raw data parser provided by Hassel [17] to parse the raw data into the prefix form. For each algorithm, the evaluation first pre-processes the policy space of policies on each network function, and then by randomly choosing a sequence of policy spaces to conduct random set operations, the operation time and the memory cost of these four algorithms are tested.

Fig. 4 shows the average operation time of doing 500 rounds of set operations on policy space of policies on each network function by each algorithm. Results show that the operation time of AHI is much lower than all other three algorithms. This is reasonable since AHI algorithm adopts the most fine-grained cutting on universal space so that it turns the set operation of policy spaces into set operations of bit vectors, which is incredibly fast compared with other three algorithms. The PSA tool is also faster than the remaining two algorithms, which has to be attributed to its reference to the clipping region management in computer graphics field [8]. As for the remaining two algorithms: r-BDD and wildcard-expression, the reason of their failure to compete with AHI or PSA is as follows: 1) The set operation itself is quite complicated. For r-BDD, it has to recursively calculate on each node in the diagram, and for wildcard-expression, it has to do set operation between each wildcard expression in two lists 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 structure will increase simultaneously.

Another thing must be mentioned is that, among these four algorithms, AHI is the only algorithm that avoids the 'cumulative effect' in intensive set operation tests. That is, when we perform intensive set operations continuously, the size of the data structure used by other three algorithms to hold the result policy space of these set operations will grow rapidly. As a result, algorithms which 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 result policy space gets complex, and the number of wildcard expressions used to represent the result will also grow, so as the number of hyper-rectangles in PSA. But as for AHI algorithm, because we have already divided the universal space into the smallest atomic hyper-rectangles, however complex the result is, we can still use a fixed length bit vector to represent it, and the time cost of doing set operation on it will not change.

Fig. 5 shows the memory cost of each algorithm in the test. We can see that AHI 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 AHI, we only need to store the projection points set $\{Pt_d[i], 1 \le i \le M_d\}, d = 1, 2, ..., DIM\_MAX$, and the bit vector of each policy space. In wildcard expression algorithm, each policy space will be represented by a set of wildcard expressions composed by '1', '0' and 'X'. But in PSA and r-BDD, policy spaces will be represented by sets of hyper-rectangles and decision diagrams, which is rather complex and memory-consuming.

However, in this early implementation of AHI, the pre-processing step to generate the atomic hyper-rectangles is somekind time-consuming. This is because of by doing the most fine-grained cutting on the whole packet header space, AHI generates too many hyper-rectangles, and the process of deciding the value of each bit in each bit vector sharply increases the amount of computation in pre-processing. We are working on some methods to decrease the pre-processing time by pruning the bit-deciding process in our on-going work.

VI. Conclusion

Addressing on the distributed policy enforcement problem in the recent overwhelming SDN/NFV architecture, this paper designs an efficient algorithm leveraging spatial projection and bitmap indexing to provide basal policy space set operations for policy analysis. AHI is evaluated with real life datasets, compared with three other algorithms, r-BDD, wildcard expression and PSA. The results show that AHI achieves two to three orders of magnitude operation time improvement, with the same least space cost. This work shields lights on implementing fast and scalable policy enforcement framework with the proposed policy space set operation mechanism.

The on-going research contains reducing the pre-processing time of AHI by adopting better bit vector generation algorithm, taking a step further in speeding up the AHI operations by leveraging the CPU vector instructions, and integrating AHI into current policy enforcement frameworks.

Acknowledgment

This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000102, and the National Key Technology Research and Development Program of China under Grant No. 2015BAK34B00. And the authors would like to thank the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving the final version of the paper.

References

[1] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 121–136.

[2] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[3] Z. Liu, X. Wang, and J. Li, "From cia to pdr: A top-down survey of sdn security for cloud dcn," *ZTE Communications*, vol. 1, p. 011, 2016.

[4] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[5] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "Scalable rule management for data centers," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 157–170.

[6] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 545–549.

[7] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 13–24.

[8] X. Wang, W. Shi, Y. Xiang, and J. Li, "Efficient network security policy enforcement with policy space analysis," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2926–2938, 2016.

[9] 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.

[10] 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.

[11] 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.

[12] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 29–42, 2015.

[13] Z. Liu, X. Wang, B. Yang, and J. Li, "Bitcuts: Towards fast packet classification for order-independent rules," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 339–340.

[14] 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.

[15] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on computers*, vol. 100, no. 8, pp. 677–691, 1986.

[16] 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.

[17] "Header space library (hassel)," https://bitbucket.org/peymank/hassel-public/wiki/Home.