MBBrick: Unified Middlebox Design and Deployment in Software Defined Network

Xin He*, Qing Li*‡, Mingwei Xu†, Yong Jiang*, Lei Wang*
* Graduate School at Shenzhen, Tsinghua University, Shenzhen, China
† Tsinghua University, Beijing, China

Abstract—**Due to the expensive hardware and complex management of the traditional middlebox, a concerted effort towards the virtualized middlebox has been launched in both academia and industry. In this paper, we propose a unified middlebox model, MBBrick, which is composed of three operation modules (classifier, rewriter, forwarder) and a control module (the mcontroller). We then design a language, MG, to describe the MBBrick process. By concatenating different modules, MBBrick can achieve the functions of different middleboxes or service chains in one single super middlebox. We design the corresponding algorithms to optimize the MBBrick deployment and schedule the traffic. The experiment results show that MBBrick significantly reduces the processing latency and improves the network throughput.**

I. Introduction

The middleboxes are now an intrinsic and fundamental part of current Internet [1], which can improve network security (e.g., firewall), implement load balance (e.g., load balancer) and reduce bandwidth cost (e.g., WAN optimizers). The traditional hardware middlebox has a higher processing efficiency, but it is expensive and inflexible in deployment and management [2]–[5].

To shift the middlebox functions from hardware to software, Network Function Virtualization (NFV) has been recently proposed [6], [7]. Besides, Software Defined Networking (SDN) provides a flexible approach to manage these software middleboxes [2], [8]. These schemes effectively solve the problem of hardware middleboxes and avoid the redundant hardware deployment. However, the different middleboxes may have the same packet processing stages and replicated functional modules (e.g., packet classification), which causes resource waste [9]–[11]. In addition, the service chain makes the middlebox management and deployment more complicated [12].

To solve these problems, OpenBox [10] presents a software-defined framework for centralized middlebox management. It splits the middlebox into different modules (classifiers, modifiers, etc.). An efficient algorithm is provided for the control plane of OpenBox to achieve the diverse middlebox functions for the corresponding packets by merging some modules together. However, it does not propose a unified model or language to express all types of middleboxes. Besides, the processing path of OpenBox can be further optimized.

In this paper, we propose a universal model of middleboxes (MBBrick), which implements multiple functions in one single consolidated virtual box. MBBrick significantly facilitates the middlebox management and deployment. Besides, it increases the resource utilization and reduces the link transmission latency.

First, we propose the framework of MBBrick, which is composed of the control module (the mcontroller) and three data plane modules (classifier, rewriter, forwarder). We design MG language to describe the process of MBBrick. Each middlebox of MBBrick has one corresponding mcontroller. The mcontroller manages the middlebox by translating the commands from the SDN controller. The classifier is responsible for packet classifying. The rewriter is used to rewrite the packet (NAT, proxy, etc.). The forwarder forwards a packet to the next node or drops it according to the action. Most general middleboxes can be implemented by concatenating the modules of classifiers, rewriters and forwarders in different orders. What's more, a service chain including multiple middleboxes can also be implemented by combining the modules in one single MBBrick middlebox. In this way, MBBrick reduces resource usage. Furthermore, the same operation is done only once by adjusting the order.

Then, we propose the algorithms to deploy MBBrick middleboxes and schedule dynamic traffic in the network to reduce the latency and improve the throughput. We formalize the whole problem mathematically and show the non-trivial complexity accordingly. Therefore, we divide the problems into two stages. In the first stage, given the network topology and the historical traffic, the offline algorithm of MBLDecision is employed to find the proper positions for a certain number of MBBrick middleboxes. In MBLDecision, we aim to minimize the transmission overhead of all historical flows. In the second stage, for the online dynamic flows, the algorithm of LLFSchedule is designed to reduce the processing latency by determining the processing positions for the flows. When necessary, it also needs to update the modules in the corresponding MBBrick middlebox.

To demonstrate the performance of MBBrick, we construct comprehensive experiments on the topologies from Topology Zoo (Geant and Cerent) [13] and Rocketfuel (AS1221, AS1239) [14]. The results show: 1) MBBrick reduces the processing latency of service chains compared with OpenBox; 2) MBBrick increases the throughput greatly with the same number of middleboxes; and 3) MBBrick reduces the transmission latency for the dynamic traffic and increases the resource utilization.

‡ Corresponding author

II. Background and Motivation

Middlebox management is a significant issue for network performance, which attracts lots of attention. There are many attempts to unify middleboxes. In [15], Dilip Joseph, *et al.*, try to use a universal language to describe the middlebox processing. But it does not solve the problem of the service chain and each middlebox is still deployed and managed independently. CoMb [16] consolidates the software middlebox applications to run on the hardware platform and conducts centralized control. However, this framework only reuses modules at the application granularity and is policy-dependent.

Generally, the software middlebox is deployed in the VM running on a server [17]. In [18], Xiaoqiao Meng, *et al.*, propose a scheme to improve the network scalability by optimizing VMs placement on host machines and traffic patterns among VMs. Stratos [19] deploys and schedules flows based on the service chain to reduce communication overhead at the virtual machine level. Slick [20] takes better use of network resources by allowing the placement of fine-grained programmable functions. However, in these schemes, a packet with a long service chain will be processed across multiple servers or VMs. Therefore, the communication overhead among servers or VMs is still non-trivial. It impacts the processing time of the packet with a long service chain.

In this paper, we propose the unified middlebox, MBBrick[1]. Each MBBrick is composed of multiple modules (threads) that can be concatenated together to achieve the functions of different traditional middleboxes. Therefore, a packet with a long service chain can also be processed in the same MBBrick (corresponding to a physical server). In this way, the communication overhead among servers or VMs is avoided and the network processing latency is reduced.

OpenBox [10] presents a software-defined framework for centralized middlebox management. It splits middlebox into different modules (classifiers, modifiers, etc.). However, it does not propose a unified model or language to express all types of middleboxes. Besides, the processing path of OpenBox can be further optimized. In MBBrick, we classify packets at the first step and process them in parallel, also start several rewriter threads to increase the throughput.

For example, suppose there are multiple user networks (with NAT) sharing the same firewall and proxy (as they may be expensive). Therefore, for each packet, the service chain is $NAT \rightarrow firewall \rightarrow proxy$. The resulted process chains of MBBrick and OpenBox are shown in Fig. 1. We can see that in OpenBox, there are some replicated modules, which are eliminated in MBBrick.

Furthermore, suppose we deny the network connection of h_a but allow the connection of h_b. They request the same content from a proxy and then pass through a firewall. The service chain is $proxy \rightarrow firewall$. However, if the content is cached in the proxy after the request of h_b, h_a can also obtain the content from the proxy before denied by the firewall. While MBBrick can avoid this problem as the packet classification is made at the first stage.

[1]In our paper, MBBrick means the scheme or the middlebox of MBBrick.

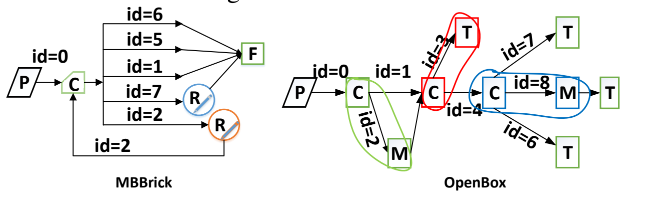

Fig. 1. An Example of MBBrick and OpenBox. (P: the input packet, C: Classifier, R/M: Rewriter/Modifier, F: Forwarder, T: Terminal)

III. MBBrick Middlebox Design

In this section, we propose MBBrick, which is a unified model to achieve middlebox functions at a fine granularity. In MBBrick, MG language is designed to describe packet processing in the middleboxes.

Tab I. Notation List

Notation	Definition
I, I'	input and output flow classification IDs
ϕ_j^k	the k^{th} thread function in j^{th} MBBrick middlebox, including classifier C_j^k, rewriter R_j^k, forwarder F_j^k
X_n^i, Y_n^i	the i^{th} input/modified packets of the n^{th} flow
$\pi_j^k(I)$	the service chain of the flow I
φ_j^k	the super controller function: create thread Cr, update resource Up, insert In or delete De rules for ϕ_j^k
$r_j^k(I)$	the rule for the flow I in ϕ_j^k
J, T	command number from controller, rule residence time
$S(J)$	the traditional single middlebox, e.g., firewall, NAT
$R(J)$	the rule of the $S(J)$

A. Model Overview

The process steps of the middlebox can be concluded as receiving, processing, and forwarding. Therefore, we define three types of operation modules (classifier, rewriter, forwarder), whose combination can achieve different middlebox functions. In our scheme, we assume each MBBrick module shares the same storage, which is used for storing the cache information or intermediate states of the middlebox. The model of MBBrick is shown in Fig. 2.

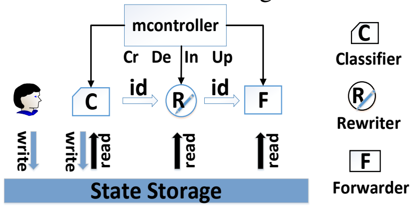

Fig. 2. MBBrick Model

$$\phi_j^k(X_n^i, r_j^k(I)) \rightarrow Y_n^i, I', \pi_j^k(I) \;\; I, I' \in \{I\}$$
$$r_j^k(I) : I \rightarrow I \times Action \qquad (1)$$
$$\pi_j^k(I) : I \rightarrow I \times SingleMB$$
$$\varphi_j^k(X_n^i, S(I), R(J)) \rightarrow X_n^i, r_j^k(I), \pi_j^k(I), T_j^k \qquad (2)$$

Before diving into the details of each module of MBBrick, we first define the notations, as shown in Tab. I. The ϕ_j^k function composes of C_j^k, R_j^k and F_j^k, and is simplified as $\phi_j^k(r_j^k(I))$. Only C_j^k can change the input ID. For R_j^k, F_j^k function, we have $I = I'$ and $\{I\}$ denote the ID set. MG language describes the operation processing of MBBrick middlebox and the mcontroller instructs modules as above.

B. Module Specification

The mcontroller manages the middlebox by translating the commands from the SDN controller. It also collects the status of the MBBrick middlebox for the SDN controller. The mcontroller executes the Cr, In, De, Up actions and adjusts T for each MBBrick middlebox.

The classifier module classifies packets according to the rules from the mcontroller, such as the IP/HTTP header classifier, the payload classifier, etc. For example, in firewalls, prefix matching is often used to test whether an IP address contains a prefix. This module classifies the IP addresses into different classes based on the matching rules. The function of a classifier is formulated as Equation 3. To keep the consistent format of function modules, we set the initial class id to I_0.

$$C_j^k(X_n^i, r_j^k(I_0)) \rightarrow X_n^i, I', \pi_j^k(I_0) \quad I_0, I' \in \{I\} \tag{3}$$

The rewriter module is used to rewrite the packet, such as the IP/HTTP header/payload rewriters, etc. For example, some load balancers rewrite the destination IP address to control the routing paths; some proxies construct a response for a URL request by rewriting both the IP header and the HTTP payload. This type of module is shown as Equation 4.

$$R_j^k(X_n^i, r_j^k(I')) \rightarrow Y_n^i, I', \pi_j^k(I') \quad I' \in \{I\} \tag{4}$$

The forwarder module includes forwarding and dropping. For example, the firewall and IPS drop malicious packets, and load balancers forward packets according to the specific policies. For simplicity, we assume the middlebox has only one ingress/egress port. Thus, $r_j^k(I) = 0/1$ indicates dropping (0) or forwarding (1).

$$F_j^k(X_n^i, r_j^k(I')) \rightarrow X_n^i, I', \pi_j^k(I') \quad I' \in \{I\} \tag{5}$$

C. Unified MBBrick Model

MBBrick, as a unified middlebox, uses the approach of module merging to avoid module replication. Although unification can improve the management of middleboxes, it is a difficult task as there are diverse middleboxes with different structures. We analyze all types of middlebox structures and use classifier, rewriter and forwarder to implement the functions of the middlebox and the service chain. Some examples are shown in Tab. II.

1) Module merge operation: We take firewall, NAT and proxy as examples. We first use $C_j^k \rightarrow R_j^{k+1} \rightarrow F_j^{k+2}$ to describe a single middlebox. Then we concatenate the modules together to achieve the function for the service chain such as $firewall \rightarrow NAT \rightarrow proxy$.

A firewall can be converted into a $C_j^k(IP) \rightarrow F_j^{k+1}$ pipeline. A NAT converts the internal address/port to the external one, which is represented as $C_j^k(IP) \rightarrow R_j^{k+1}(IP)$. We show the detailed process of MBBrick with MG language to implement the NAT function.

For NAT, an IP header rewriter is used to rewrite the source address/port. The modules of MBBrick share the same state

Tab II. Several Middlebox Models

middlebox	C	R	F	Layer	Storage
Firewall	Ip	X	√	2	none
NAT	Ip	IP,Port	√	2	none
Proxy	Ip,Http	Ip,Http	√	7	URL
IPS	Ip Content	X	√	2	info
NAT-PT	Ip	IP,Port	√	2	none
Socks GW	X	X	√	4	none
Load Bal.	Ip,Port	IP,Mac	√	4	Mac

storage. The process is presented as Equation 6 and shown in Fig. 3.

$$\begin{aligned}
C_j^k(X_n^i, r_j^k(I_0)) &\rightarrow X_n^i, I', \pi_j^k(I) \quad I_0, I' \in \{I\} \\
R_j^{k+1}(X_n^i, r_j^{k+1}(I')) &\rightarrow Y_n^i, I', \pi_j^{k+1}(I') \quad I' \in \{I\} \\
F_j^{k+2}(Y_n^i, r_j^{k+2}(I')) &\rightarrow Y_n^i, I', \pi_j^{k+2}(I') \quad I' \in \{I\} \\
r_j^k(I_0) &: I_0 \rightarrow I_0 \times classify \\
r_j^{k+1}(I') &: I' \rightarrow I' \times rewrite \\
r_j^{k+2}(I') &: I' \rightarrow I' \times forward \\
\pi_j^k(I_0) &: I_0 \rightarrow I_0 \times NAT \\
\pi_j^{k+1}(I') &: I' \rightarrow I' \times NAT \\
\pi_j^{k+2}(I') &: I' \rightarrow I' \times NAT
\end{aligned} \tag{6}$$

The proxy is a little more complex. When the packets arrive, the proxy determines whether they are client requests or server responses by the IP header classifier. For the server response packet, the URL is added to the cache stage storage. To redirect the contents to the client, an HTTP forwarder is used. For the requests, the proxy first classifies the packets by the HTTP header classifier. If the URL is missed, the proxy writes the destination-related states into the shared storage by the IP header classifier and forwards the content to the destination server by the HTTP forwarder. Otherwise, the proxy writes the source-related states into the storage and responses to the client with reconstructed packets. The processing procedure is shown in Fig. 3.

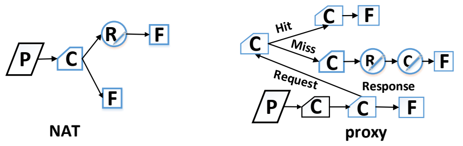

Fig. 3. NAT and Proxy Model

Given the service chain like $NAT \rightarrow firewall \rightarrow proxy$, we combine these modules to achieve the corresponding function, as shown in Fig. 4. The gray blocks implement the firewall function and the orange blocks change packet IPs and ports to the public addresses. The rest part in the figure is the function of proxy. The dotted black lines are centralized on a classifier module. The module can be reused by the consequent classifiers if it has enough idle resources. The forwarder module is analogous.

2) Merge operation optimization: Through the last part's processing, we concatenate all modules together to achieve the functions of the service chain. However, the duplicate modules exist and the corresponding states are still complex. In this part, we optimize the service chain.

Before optimization, we define some rules to ensure the correctness of the MBBrick. First, each module is decidable, i.e., with the correct states and class ID, the module works correctly. Second, before the packets processing, the corresponding states must be prepared. e.g., the related rules have been inserted into the module. Last, the classification ID should be unique and in the right subset.

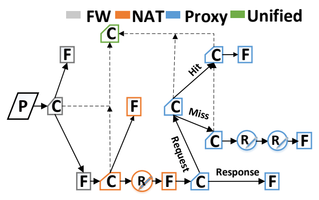

Fig. 4. Concatenated MBBrick Model

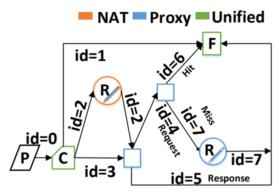

Fig. 5. Optimal MBBrick Model

Now, we assume that the classifier outputs the right class ID. There are two classifiers outputting I_1 and I_2, belonging to two class ID set set_j, set_k respectively.

$$\begin{gathered} C_j^k(X_n^i, r_j^k(I_0)) \rightarrow X_n^i, I_1, \pi_j^k(I_1) \\ C_j^{k+1}(X_n^i, r_j^{k+1}(I_1)) \rightarrow X_n^i, I_2, \pi_j^{k+1}(I_2) \\ I_1 \subseteq set_j \qquad\qquad I_2 \subseteq set_k \end{gathered} \tag{7}$$

If we put the classifier ahead, we should ensure that the set relationships are kept. This process is shown as Equation 8.

$$\begin{gathered} C_j^k(X_n^i, r_j^k(I_0)) \rightarrow X_n^i, I_1, \pi_j^k(I_1) \\ C_j^{k+1}(X_n^i, r_j^{k+1}(C_j^k(X_n^i, r_j^k(I_0)))) \rightarrow X_n^i, I_2, \pi_j^{k+1}(I_2) \\ I_2 \subseteq set_j \qquad I_2 \subseteq set_k \end{gathered} \tag{8}$$

According to the above three rules, we concatenate all these same functional modules by adjusting the processing order. We unify these rules to classify packets at the first stage. Then the service chain like $NAT \rightarrow firewall \rightarrow proxy$ is optimized as Fig. 5. In the figure, we insert some blank blocks whose functions can be achieved by the first classifier module.

IV. Traffic Schedule with MBBrick

In this part, we propose the algorithms to deploy MBBrick middleboxes and schedule dynamic traffic in the network to reduce the latency and improve the throughput. We formalize the whole problem mathematically and show the non-trivial complexity accordingly.

A. Problem Formulation

We describe a flow by the triple (src, $dest$, MBa, MBb, MBc), which means the flow from src to $dest$ should be processed by three types of middlebox (a, b, c) [21]. Some notations are described in the Tab III.

Given a topology, we choose proper locations to deploy MBBrick middleboxes. Each MBBrick middlebox is supported by a multi-core server and in a MBBrick middlebox, each module runs in an individual thread. We set that each MBBrick middlebox works within its capacity, and each module is analogous (Equation 10, 11). The bandwidth consumption for each link should be smaller than the B_s (Equation 12). We express the network latency by the sum of the t_{ij} and the p_{ij}, the objective is shown as Equation 9.

$$\text{minimize} \quad \sum_{i=0}^{N}\sum_{j=0}^{M} x_{ij}(t_{ij}+p_{ij}), \quad x_{ij} \in \{0,1\} \tag{9}$$

$$\text{subject to} \quad \sum_{k=0}^{T} \omega_j^k \leq \delta_j, \tag{10}$$

$$\sum_{k=0}^{T}\sum_{i=0}^{N} y_{jk}^i \sigma_{jk}^i \leq \omega_j^k, \quad y_{jk}^i \in \{0,1\} \tag{11}$$

$$\sum_{s \in L} b_{f_s} \leq B_l, \qquad l \in L \tag{12}$$

Tab III. Notation List

Notation	Definition
i, j, k	the i^{th} flow, the j^{th} MBBrick, the k^{th} thread
N, M, T	the number of flows, MBBrick middleboxes, threads
σ_{jk}^i, ϱ_i	resources needed by flow i (the k in j , all threads)
δ_j, ω_j^k	the resources of the j, the resource of the k in the j
$L=\{1,.,L\}$	the network link
b_i, f_l	the flow sent rate, the flow passing through link l
b_{f_s}	the bandwidth consumption of the s^{th} path
t_{ij}, p_{ij}	the transmission, processing latency of i through j
G, B_l	the network topology, the link bandwidth, $l \epsilon L$
$flow$,$hflow$	the flow tuple, historical flow distribution
$\{U\}$, v_n	the normal node set, the normal node
$\{MB\}$, v_m	the MBBrick set, the node deployed with MBBrick
$d(v_n, v_m)$	the path length from v_n to the nearest v_m
$maxlen$	the maximum from v_n to its nearest v_m
N_{fre}^v	the passing frequency for each node
N_{deg}^v	the degree of each node

In order to solve the above problem, we choose k locations each time to deploy MBBrick, and set these k locations into an aggregate expressed as {K}, the size of {K} set is described as K, then we can know the number of {K} is $C(M, K)$, the result is $\frac{M!}{(K!*(M-K)!)}$. When the network topology is complicated, it's difficult to solve this combinational optimization problem.

B. Algorithm of the Two Stages

We design the $BrickSchedule$ algorithm to solve the above problem which is composed of two stages. First, the $MBLDecision$ algorithm is applied to find the proper positions for a certain number of MBBrick middleboxes based on the network topology and the historical traffic. Second, the $LLFSchedule$ algorithm aims at reducing the processing latency by determining the processing positions for the flows and updating the modules in the MBBrick middlebox.

$$\text{minimize} \quad \sum flow_n^m * maxlen \tag{13}$$

$$\text{subject to} \quad maxlen = maximize\ d(v_n, v_m), \tag{14}$$

$$\forall v_n \in \{U\},\ \forall v_m \in \{MB\} \tag{15}$$

$$correlation^v = N_{fre}^v + \alpha * N_{deg}^v \tag{16}$$

We deploy MBBrick middleboxes to get the minimum link overhead by the $MBLDecision$ algorithm, shown as Equation 14. This problem is expressed as getting the global minimum of $flow_n^m * maxlen$. We mark each node based on its degree N_{deg}^v and frequency N_{fre}^v and express the normalization result as $correlation^v$. $Correlationlist$ is used to donate the sequence of $correlation^v$, which is associated with the value of α, expressed as Equation 16. Based on the k-center algorithm [22], we select non-adjacent nodes and compare $flow_n^m * maxlen$ each time to get the least target value. This optimization reduces the computational space.

Algorithm 1 MBLDecision

```
Input: $G$, $B_l$, $b_i$, $hflow$
Output: $\{MB\}$
 1: adjust $\alpha$, $correlation^v = N^v_{fre} + \alpha * N^v_{deg}$, $v \in G$
 2: normalize $Correlationlist$
 3: if $j^{th}, k^{th}$ non-adjacent, $j, k \in Correlationlist$ then
 4:   if $j^{th}, k^{th}$ capacity within $\delta_j$, $\delta_k$ then
 5:     $\{MB\} \Leftarrow$ the $j^{th}$ node, calculate $maxlen$
 6:     if $flow_n^m * maxlen$ is not the minimum then
 7:       remove $j$ , insert $k$, update $j^{th}, k^{th}$
 8:     end if
 9:   end if
10: end if
```

The $LLFSchedule$ algorithm distributes modules for each MBBrick middlebox and schedules dynamical flows to gain the minimum latency and balance the network load. The $LLFSchedule$ algorithm has two conditions. First, no MBBrick has been allocated with modules, so we schedule flows by the algorithm of the minimum cost flow. We schedule the same operation flows into the same optimal MBBrick middlebox. Second, there are some modules working in MBBrick middlebox, we choose a proper MBBrick middlebox to make the most of the existed modules based on the ϱ_i. If the operation cost of MBBrick middlebox and other factors (e.g., path length, link bandwidth) are the same, the $LLFSchedule$ algorithm will choose the MBBrick middlebox with the largest idle resources to implement better MBBrick load balance.

Algorithm 2 LLFSchedule

```
Input: $G$, $B_l$, $\{MB\}$, $flow$
Output: $t_{ij}$, $p_{ij}$
 1: if MBBricks are idle then
 2:   conduct $Cr_j^k$, update $\delta_j$, $\omega_j^k$
 3:   $pair$ = ($flow$, $MBBrick$), $match$ = none
 4:   while $match$ is none do
 5:     $pair.adjust$, $match$ = $Hungarianmethod$
 6:   end while
 7: else
 8:   if $Op_i = Op_{i-1}$ or $Op_i = Op_j$ then
 9:     $MB_i = MB_{i-1}, MB_i = MB_j$
10:   else if existing 2 MBBrick options then
11:     $j$ = the nearest MBBrick, conduct $Cr_j^k$
12:   else
13:     $score_{ij}$ on (link, MBBrick state), j$\in \{MB\}$
14:     sort $score_{ij}$, $pair.get$, calculate $t_{ij}$, $p_{ij}$, set $T$
15:   end if
16: end if
```

V. Experiment and Analysis

In this section, we evaluate the performance of MBBrick and the corresponding algorithms.

A. Experimental Environment

The MBBrick middleboxes are deployed in the SDN. We execute the $MBLDecision$ algorithm offline and install the $LLFSchedule$ algorithm in the SDN controller, with a server providing resources for each MBBrick. MBBrick gets the rules from the controller and returns its status by southbound API supported by OpenNF [7] platform without any modification. Our algorithm is running on a server with an Intel i5, CPU 3.20GHz, 16GB RAM and Linux Ubuntu 14.04 system. To demonstrate the performance of MBBrick, we construct comprehensive experiments on the topologies from Topology Zoo (Geant, Cerent) [13] and Rocketfuel (AS1221, AS1239) [14], shown in Tab IV.

Tab IV. Experiment Topology Properties

Topo	Node	Edge	MB	Topo	Node	Edge	MB
Geant	22	72	5	AS1221	44	162	10
Cernet	41	116	8	AS1239	52	112	15

B. Experimental Results and Analysis

We compare the processing latency of MBBrick and OpenBox based on two examples as shown in Fig. 6. For the $firewall \rightarrow IPS$ instance, the latencies are similar, as these two frameworks both concatenates all classified operations into an action. For the $NAT \rightarrow firewall$ instance, MBBrick reduces the network latency by 30%, as OpenBox can not move the modifiers to the front of the classifiers, but MBBrick conducts all classifier action in the first step to reduce the redundant modules.

We aim to get the optimal throughput performance under the least server resources, then we determine the number of the MBBrick middleboxes for each topology and take AS1221 topology as an example, shown in Fig. 7. MBBrick increases the throughput greatly with the same number of middleboxes.

We compare the resource usage of the MBBrick middlebox and the traditional middlebox at the same throughput condition (2K flows) and the result is shown in Fig. 9. There are three middleboxes in each network to implement the functions of two different service chains. The MBBrick reduces the resource usage by 40%. As MBBrick avoids the replicated modules and actions.

Next, we analysis the performance of the $BrickSchedule$ algorithm. The $MBLDecision$ algorithm is executed to get the least transmission overhead. We get the proper $correlationlist$ by adjusting the value of α. $D1$ and $D2$ are the selection sequences of the MBBrick middlebox getting from $MBLDecision$ and Violent Enumeration Method respectively. Fig. 8 represents the comparison of these two methods, the ordinate value is the ratio of $flow_n^m * maxlen$ getting from $D1$ to $D2$. We find that the optimal value of α is 1 or 1/2 and we get the maximum similarity between 80% and 90% for all these topologies and the minimum traffic cost. It also represents that the network topology is as important as the historical traffic. The algorithm gets the optimal MBBrick location sequences under the optimal value of 1.

Fig. 10 is the resource utilization and the load distribution for each MBBrick middlebox, with dynamic traffic scheduled by the $LLFSchedule$ algorithm. The load distribution is not absolutely balanced, as the $LLFSchedule$ algorithm reduces the redundant modules and increases the resource utilization by scheduling flows according to the same function needs firstly. Also, the resource utilization and the load distribution are not positively correlated, as each module achieves the different functions and consumes the different resources.

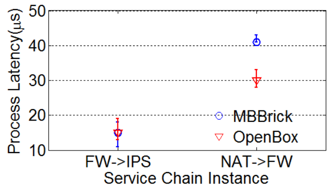

Fig. 6. The Latency Comparison

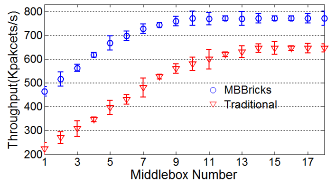

Fig. 7. The Throughput under the Different Number of Middleboxes

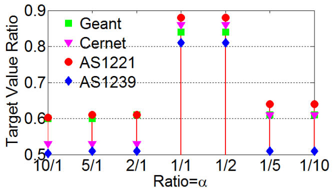

Fig. 8. The Target Value and α

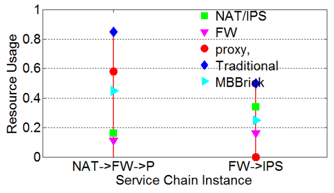

Fig. 9. The Resource Usage

Fig.11 is the comparison result of the latency of the $LLFSchedule$ algorithm (MBBrick based on the $D1$ sequences) and the $SIMPLE$ algorithm [2] (traditional middleboxes at the random location) for different topologies. For these different topologies, $LLFSchedule$ algorithm performs better, which reduces the transmission latency by 20%.

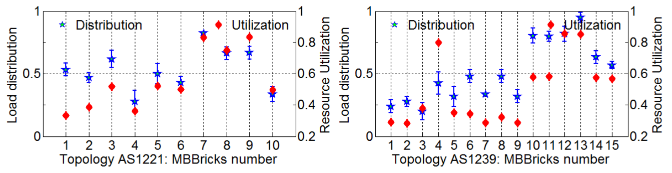

Fig. 10. Load Distribution and Resource Utilization

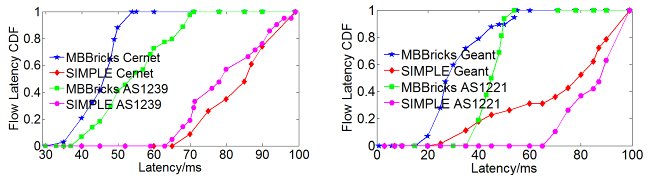

Fig. 11. Latency Distribution for different topologies

VI. Conclusion

MBBrick is a unified framework and MG language describes the process clearly. We compare MBBrick with OpenBox, and the experiment results show that MBBrick reduces the processing latency greatly. We propose a MBBrick deploy scheme and compare the latency with the SIMPLE algorithm. MBBrick reduces the transmission latency, improves the throughput and increase the resource utilization.

VII. Acknowledgement

This work is supported by the National Natural Science Foundation of China under grant No. 61402255 and No. 61625203, the National Key R&D Program of China under Grant 2016YFC0901605.

References

[1] B. B. Carpenter and S. Brim, “Middleboxes: Taxonomy and issues”, rfc 3234,” 2014.

[2] Z. A. Qazi, C. C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, “Simple-fying middlebox policy enforcement using sdn,” *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 27–38, 2013.

[3] Q. Yan, F. R. Yu, Q. Gong, and J. Li, “Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 602–622, 2016.

[4] Q. Yan and F. R. Yu, “Distributed denial of service attacks in software-defined networking with cloud computing,” *IEEE Communications Magazine*, vol. 53, no. 4, pp. 52–59, 2015.

[5] L. Cui, F. R. Yu, and Q. Yan, “When big data meets software-defined networking: Sdn for big data and big data for sdn,” *IEEE Network*, vol. 30, no. 1, pp. 58–65, 2016.

[6] R. Mijumbi, J. Serrat, J. L. Gorricho, and N. Bouten, “Network function virtualization: State-of-the-art and research challenges,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[7] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, “Opennf: enabling innovation in network function control,” *Acm Sigcomm Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2015.

[8] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, “Flowtags: enforcing network-wide policies in the presence of dynamic middlebox actions,” in *ACM SIGCOMM Workshop on Hot Topics in Software Defined NETWORKING*, 2013, pp. 19–24.

[9] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, “The middlebox manifesto: enabling innovation in middlebox deployment,” in *ACM Workshop on Hot Topics in Networks*, 2011, pp. 1–6.

[10] A. Bremler Barr, Y. Harchol, and D. Hay, “Openbox: A software-defined framework for developing, deploying, and managing network functions,” in *Conference on ACM SIGCOMM 2016 Conference*, 2016, pp. 511–524.

[11] S. Rajagopalan, W. Dan, H. Jamjoom, and A. Warfield, “Split/merge: system support for elastic execution in virtual middleboxes,” in *Usenix Conference on Networked Systems Design and Implementation*, 2013, pp. 227–240.

[12] E. T. Nadeau, “Problem statement for service function chaining,” 2015.

[13] S. Knight, H. X. Nguyen, N. Falkner, and R. Bowden, “The internet topology zoo,” *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[14] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, “Measuring isp topologies with rocketfuel.” *Acm Sigcomm Computer Communication Review*, vol. 32, no. 4, pp. 133–145, 2002.

[15] D. Joseph and I. Stoica, “Modeling middleboxes,” *IEEE Network*, vol. 22, no. 5, pp. 20–25, 2008.

[16] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, “Design and implementation of a consolidated middlebox architecture,” in *Usenix Conference on Networked Systems Design and Implementation*, 2012, pp. 24–24.

[17] H. Moens and F. D. Turck, “Vnf-p: A model for efficient placement of virtualized network functions,” in *International Conference on Network and Service Management*, 2014, pp. 418–423.

[18] X. Meng, V. Pappas, and L. Zhang, “Improving the scalability of data center networks with traffic-aware virtual machine placement,” in *Conference on Information Communications*, 2010, pp. 1154–1162.

[19] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, “Stratos: A network-aware orchestration layer for middleboxes in the cloud,” 2013.

[20] B. Anwer, T. Benson, N. Feamster, and D. Levin, “Programming slick network functions,” in *The ACM SIGCOMM Symposium*, 2015, pp. 1–13.

[21] P. Duan, Q. Li, Y. Jiang, and S. T. Xia, “Toward latency-aware dynamic middlebox scheduling,” pp. 1–8, 2015.

[22] B. Heller, R. Sherwood, and N. Mckeown, “The controller placement problem,” *Acm Sigcomm Computer Communication Review*, vol. 42, no. 4, pp. 7–12, 2012.

[23] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, “The click modular router,” in *ACM TOCS*, 1999, pp. 263–297.