Enabling heterogeneous network function chaining

Enabling Heterogeneous Network Function Chaining

Lin Cui, *Member, IEEE,* Fung Po Tso, *Member, IEEE,* Song Guo, *Senior Member, IEEE*, Weijia Jia, *Senior Member, IEEE*, Kaimin Wei, *Member, IEEE*, and Wei Zhao, *Fellow, IEEE*

Abstract—Today's data center operators deploy network policies in both physical (e.g., middleboxes, switches) and virtualized (e.g., virtual machines on general purpose servers) network function boxes (NFBs), which reside in different points of the network, to exploit their efficiency and agility respectively. Nevertheless, such heterogeneity has resulted in a great number of independent network nodes that can dynamically generate and implement inconsistent and conflicting network policies, making correct policy implementation a difficult problem to solve. Since these nodes have varying capabilities, services running atop are also faced with profound performance unpredictability. In this paper, we propose a Heterogeneous netwOrk Policy Enforcement (HOPE) scheme to overcome these challenges. HOPE guarantees that network functions (NFs) that implement a policy chain are optimally placed onto heterogeneous NFBs such that the network cost of the policy is minimized. We first experimentally demonstrate that the processing capacity of NFBs is the dominant performance factor. This observation is then used to formulate the Heterogeneous Network Policy Placement problem, which is shown to be *NP-Hard*. To solve the problem efficiently, an online algorithm is proposed. Our experimental results demonstrate that HOPE achieves the same optimality as *Branch-and-bound* optimization but is 3 orders of magnitude more efficient.

Index Terms—Network policy, Service chain, Middleboxes, Network functions, Heterogeneous, Datacenters.

1 Introduction

As cloud computing sees widespread adoption, data centers (DCs), the underpinning infrastructures of cloud, deploy a great variety of network functions (NFs) such as firewall (FWs), intrusion prevention/detection system (IPS/IDS), deep packet inspection (DPI), load balancer (LB), network address translation (NAT), etc., in the network to safeguard networks and improve application performance [1] [2]. In practice, various permutations of these NFs form an ordered composition (or chain) – as defined by a network policy [3] [4] – that must be applied to packets in uni-directional or bi-directional manner. This process is also known as network service chaining [5] [6]. Hence, network policy enforcement implies correct and efficient chaining of NFs. The chain of NFs will then be placed onto different Network Function Boxes (NFBs), e.g., middleboxes (MBs), programmable switches and NFV servers [7].

The rise of Software Defined Networking (SDN) and Network Function Virtualization (NFV) has enabled three dimensions of heterogeneity among NFBs: (1) network location – there are *in-network* NFBs such as middleboxes, SDN switches and *edge* NFBs such as commodity NFV servers; (2) processing capacity – hardware NFBs are generally more efficient than virtualized NFBs; and (3) supported NF types – each hardware middlebox usually can only support one specific type of NF whereas an SDN switch can implement a few simple NFs, e.g., stateless FW and LB built on top of OpenFlow [8]. NFV servers, albeit less efficient, can run any type of NFs.

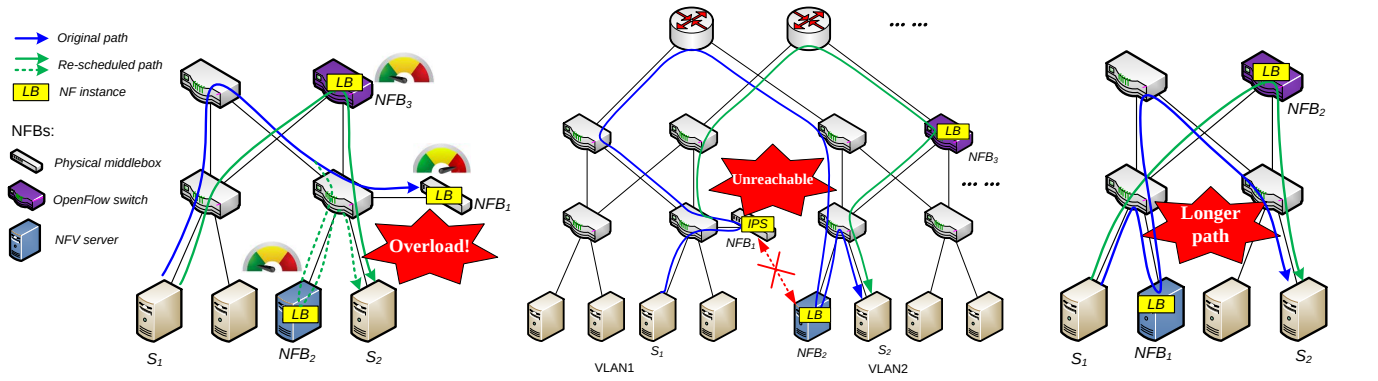
Clearly, the heterogeneity is a double edge sword. On one hand, it provides a vast combination of different NFBs – an opportunity for more innovative and sophisticated network policy implementation. On the other hand, it presents great challenges in correct and efficient chaining of NFs:

(1) Existing body of work for deployment of network policies mainly support policy enforcement on either legacy middleboxes or virtualized servers with NFV and SDN paradigms [3] [9] [10] [11];

(2) The performance of NFs is subject to the computing capability of commodity servers. Also, bottlenecked NFs can throttle the performance of other NFs in the chain. This will lead to unpredictability in application performance such as end-to-end latency [12].

To better visualize the challenges and opportunities, Fig. 1 depicts three example scenarios where heterogeneous NFBs are used. Assume the traffic from s_1 to s_2 is subject to the policy containing *LB* and/or *IPS*. (1) *Violation - MB capacity overloaded*: As shown in Fig. 1a, while the hardware middlebox NFB_1 is overloaded, rejecting incoming packets, we can place the rules to an NFV server NFB_2, which has enough capacity. Moreover, simple load balancer can also be implemented on an OpenFlow switch, e.g., NFB_3.

- *Lin Cui is with Department of Computer Science, Jinan University, Guangzhou, China.*
  E-mail: tcuilin@jnu.edu.cn.
- *Fung Po Tso is with Department of Computer Science, Loughborough University, LE11 3TU, UK.*
  E-mail: p.tso@lboro.ac.uk.
- *Song Guo is with Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong.*
  E-mail: song.guo@polyu.edu.hk.
- *Weijia Jia is with Department of Computer and Information Science, University of Macau, Macau SAR, China.*
  E-mail: jiawj@umac.mo.
- *Kaiming Wei is with Department of Computer Science, Jinan University, Guangzhou, China.*
  E-mail: cswei@jnu.edu.cn.
- *Wei Zhao is with American University of Sharjah, PO Box 26666, Sharjah, U.A.E.*
  E-mail: zhao8686@gmail.com.
  Corresponding author: Fung Po Tso



(a) Scenario 1: NFB capacity overloaded

(b) Scenario 2: Route unreachable

(c) Scenario 3: Suboptimal performance

Fig. 1: Examples scenarios in heterogeneous environment

This example also reflects the use of *all three dimensions of heterogeneity* to remedy potentially infeasible policy implementation. (2) *Violation - route unreachable*: Physical NFBs and virtualized NFBs are usually managed by different controllers, which may be misconfigured. As demonstrated in Fig. 1b, the traffic from s_1 to s_2 must be checked by IPS and LB. IPS is on a hardware middlebox NFB_1 that is behind VLAN1, while LB is on an NFV server NFB_2 behind VLAN2. Thus, packets would fail to be sent from IPS to LB if the policy requirement is enforced. However, by placing the load balancer on an OpenFlow switch (NFB_3) to exploit the *network location heterogeneity*, the policy can be successfully enforced. (3) *No violation - suboptimal performance*: As shown in Fig. 1c, placing LB in hardware middlebox or NFV server, i.e., NFB_1, would result in longer paths than placing LB on on an OpenFlow switch NFB_2. Hence, both of the *network location and supported NF type heterogeneity* should be used to ensure that the overall path is minimized.

In this paper, we study a novel network policy enforcement scheme, in terms of network function chaining, in heterogeneous NFBs environment. We first experimentally demonstrate that processing capacity of NFBs is the dominant performance factor. We then analytically show that some particular types of NFs can be re-ordered without compromising the correctness of the chain. Combining these two observations with the heterogeneity of NFBs, we formulate the Heterogeneous Network Policy Placement problem with an objective of minimizing network cost.

We then propose an online Heterogeneous netwOrk Policy Enforcement (HOPE) scheme that can always find an optimal service chain path for each policy. We favor online algorithms because today's policy generation and deployment are more dynamic due to proliferated adoption of SDN paradigm. Our experimental evaluation demonstrates that HOPE can achieve the same optimality as *Branch-and-bound* optimization but has three orders of magnitude smaller runtime.

In short, the contributions of this paper are three-fold:

1) The formulation of the Heterogeneous network policy enforcement (HOPE) problem which explores the performance heterogeneity for running same network functions on different NFBs.
2) Design of an efficient optimal online scheme to solve the HOPE problem. In addition, an greedy approach is also provided when efficiency is needed to trade-off optimality
3) Implementation and extensive evaluation of HOPE on a real testbed. The results demonstrate that HOPE is effective and practical.

The remainder of this paper is structured as follows. Section 2 presents our simple experiments on revealing performance heterogeneity across the same NF on different implementation of NFBs with various capacity. Section 3 describes the problem formulation with re-ordering of service chain. Efficient schemes for HOPE are proposed in Section 4, followed by testbed implementation and performance evaluation of HOPE in Section 5 and 6 respectively. Section 7 outlines related works. Finally, Section 9 concludes the paper and discusses future works.

2 NFBs Performance Heterogeneity

To understand the extent to which the existence of performance heterogeneity among different NFBs configurations, we have carried out a set of simple test-bed experiments using several commodity servers and one Pronto 3295 SDN switch. Each commodity server has an Intel's Xeon E5-1604 4 cores CPU, 16GB RAM and a dual port 1 Gbps NIC, and Ubuntu 14.04 operating system. One server has been used as a virtualized NFB, with KVM as the hypervisor. Another two servers have been used as client and server respectively, running software tools, e.g., *iPerf* [13]. We have also used a Pronto 3295 SDN switch to emulate a hardware NFB and used two popular open-sourced softwares – Firewall (pfSense v2.3.1 [14]) and IDS/IPS (Snort v2.9.8 [15]) – as our NFs.

2.1 Virtualization Impact on Performance

We first show how virtualization impact on the performance of NFBs. Virtualization usually introduces significant network delay due to the long queuing at the drivers domains

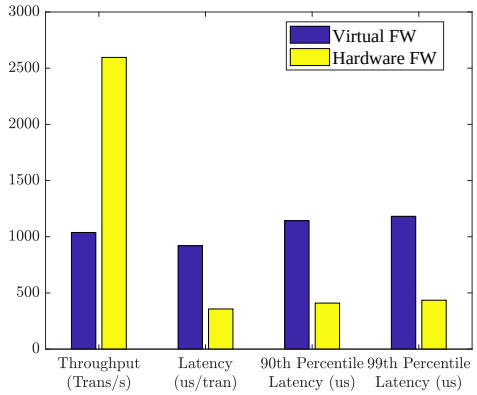


Fig. 2: Virtualization impact on firewall (FW)

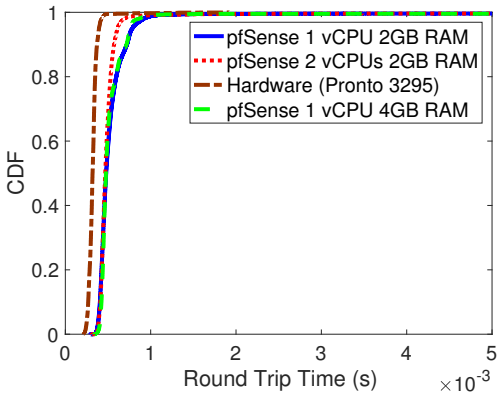
of the virtualized machines [16]. In this experiment, we have compared network performance on transaction processing at virtual Firewall (FW) deployed in a VM and another similar FW instance deployed on hardware NFB (a commodity server running Ubuntu 14.04 OS). We studied both their transaction processing capability and latency introduced. As shown in Fig. 2, the transaction processing throughput decreased from $2,500$ Transactions per second (Trans/s) to nearly $1,000$ Trans/s when using virtualized FW. Similarly, latency at both $90th$ and $99th$ percentiles increased similarly from $500\mu s$ to $1,300\mu s$. Also, the latency per transaction increased by nearly 250%. These results shows that different NFBs implementation have varied processing capabilities. Particularly, virtualized NFBs can have lower processing capability with longer latency compared to other implementations.

2.2 Correlation with CPU & Memory

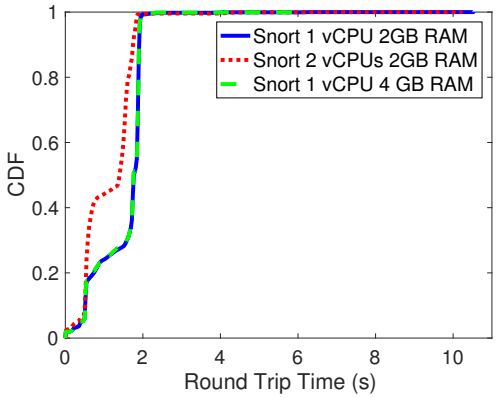
We then study the correlation of performance heterogeneity of NFBs with different number of allocated CPUs. In this set of experiments, we have first allocated only one vCPU (1 vCPUs, 2GB RAM) for both pfSense and Snort servers and then increased the number of vCPUs to two, while keeping the memory (2 vCPUs, 2GB RAM) and other configurations unchanged.

The computed RTT from the recorded traffic has been demonstrated in Fig. 3. Since no links in this setup are over-subscribed, the likelihood of traffic congestion is low. Thus, processing delay accounts for significant portion of end-to-end latency. Clearly, Fig. 3a shows that having twice as much hardware resource does not significantly improve RTT as there is only about 5% improvement at the region above 80 percentile. In comparison, the hardware switch implementation has much smaller and predictable RTT, even at the 99th percentile. Fig. 3b shows more diverse performance results among two configurations for Snort IDS/IPS in which 2 vCPUs could give significantly better performance up to as much as 100%.

In addition, we have also noticed that the magnitude of RTT for Snort is two orders higher than that of pfSense. This is because the pfSense's workload was mainly on examining the packet header for NAT translation, whereas for IDS/IPS the workload was mainly on deep packet inspection.



(a) NAT on hardware and virtual NFBs



(b) IDS/IPS on virtual NFB

Fig. 3: CDF of RTTs for pfSense and Snort NFBs with different allocated CPU and memory.

We then altered the configuration (1 vCPU, 2GB RAM) to increase the size of memory from 1GB to 4GB (1 vCPU, 4GB RAM). The results shown in Fig. 3 exhibit only small differences in performance across two configurations.

2.3 Observations

Clearly, thisds set of experiments has revealed that:

1) Programmable SDN switches can implement simple NFs.
2) Significant performance diversity exists among hardware and virtual NFs with different allocated system resources.
3) The performance of NF is largely limited by NFB's processing capacity rather than its amount of memory.

These observations demonstrate that alongside the heterogeneity that we have considered, it is of profound importance to take into account the capacity of an NF for processing packets (Section 3.3.2).

3 Problem Modeling

3.1 Heterogeneous NFBs

In Section 2, we have shown the heterogeneity on performance among different NFB implementations. Thus, in this paper, as opposed to existing works which mainly consider

TABLE 1: Notations

Symbols	Descriptions
$\mathbb{B}$	$\mathbb{B}$ is the set of all NFBs
b_i	b_i is an instance of $\mathbb{B}$, i.e., $b_i \in \mathbb{B}$
$b_i.cap$	maximum capability of b_i
$b_i.typeset$	supported NF types of b_i
$\mathbb{N}$	$\mathbb{N}$ is the set of all NFs
n_i	n_i is an instance of $\mathbb{N}$, i.e., $n_i \in \mathbb{N}$
$n_i.type$	function type of n_i
$n_i.cap$	processing requirement of n_i
$n_i.location$	the NFB that hosts n_i
$\mathbb{P}$	$\mathbb{P}$ is the set of all network policies
p_i	p_i is an instance of $\mathbb{P}$, i.e., $p_i \in \mathbb{P}$
$p_i.sep, p_i.dep$	source and destination End Points of p_i
$p_i.len$	number of NFs in p_i
$p_i.set$	all possible sequence of p_i with re-ordering
$p_i.chain$	sequence of NFs that all flows matching p_i should traverse in order
$Cost(p_i)$	cost introduced by policy p_i
$D(n_i, n_j)$	delay between n_i and n_j
$t_p(n_i)$	processing delay of n_i
$T(p_i)$	expected delay for the flow constrained by p_i
$B(n_j)$	NFBs that can host n_j
$A(b_j)$	the set of NFs hosted on b_j
T_{update}	timer of HOPE periodical operation
P_H	Policies that need to be re-scheduled

homogeneous NFBs deployment, we consider a heterogeneous environment where NFs can be implemented at various network locations, either in-network or at-edge, and on different kinds of NFBs. For example, these heterogeneous NFBs can be hardware middleboxes, commodity servers, and (SDN) switches/routers. These NFBs implementations are distinctively different in following ways:

- *Hardware middleboxes* are usually vendor specific, proprietary boxes for providing specific network functions [17]. Their designs are often optimized for performance but less extensible. They are implemented as real hardware boxes that are often located in exact point where flows should be operated [18].
- *NFV servers* are virtualized that can run multiple, and theoretically, any types of virtual network functions (VNFs). As they are built on virtualization, enable flexibility running on commodity hardware. So, better agility can be guaranteed [19].
- Some NFs can also be implemented on *switches* or *routers* such as NAT, simple firewalls and load balancers. They are amongst hardware middleboxes. However, SDN also allows us to exploit OpenFlow switches to increase the performance of service chain by installing some rules (i.e., NF) to their tables [20] [21].

Due to special properties of different implementations of NFBs, they are suitable for different scenarios and applications. Therefore, we anticipate that the heterogeneous implementation of NFBs will exist for the foreseeable future. Fig. 4 shows an example of implementing a policy using different types of NFBs.

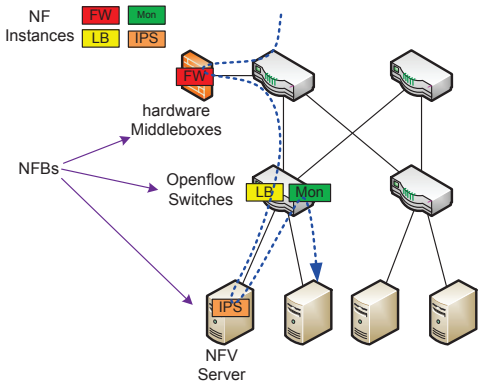


Fig. 4: Service chain example for north-south traffic in DC: $FW \rightarrow LB \rightarrow IPS \rightarrow Monitor$

3.2 Network Policy Model

A table of notations that will be used in the following of the paper is shown in Table 1.

Denote $\mathbb{B} = \{b_1, b_2, \ldots\}$ to be the set of all NFBs in a data center. For an NFB b_i, there are two fields $\{cap, typeset\}$. $b_i.cap$ denotes the maximum processing capability of b_i, measuring in number of packets per second (*pps*), e.g., 3800 *pps* [22]. And $b_i.typeset$ specifies the set of supported NF types on b_i. For example, NFV servers theoretically support all types of NFs, while hardware middleboxes and OpenFlow switches can only support one or few types of NFs. Without loss of generality, we assume that the memory space of NFBs are enough to accommodate states information of all NFs, i.e., bottleneck of NFBs is the processing capacity as shown by experiment results in Section 2.

Let $\mathbb{N} = \{n_1, n_2, \ldots\}$ be the set of all NF instances in data center. Each NF n_i has several important properties $\{type, cap, location\}$. The property $n_i.type$ defines the function of n_i, e.g., *IPS/IDS*, *LB*, or *FW*. The $n_i.cap$ is essentially the processing capacity requirement of n_i in *pps*. And, $n_i.location$ indicates the NFB that currently hosts n_i.

A centralized *Policy Controller* [11] is deployed in the network to manage and configure all NFs in $\mathbb{N}$, which may belong to different applications. The *Policy Controller* can monitor and control the liveness of NFs and NFBs, including addition, failure/removal or migration of NFs. Network administrators can specify and update policies through the *Policy Controller*.

A service chain defines an ordered or partially ordered set of abstract network functions and constraints, e.g., ordering, which need be applied to traffic. The set of enabled service chains reflects operator service offerings and is designed in conjunction with network policies [2].

The set of network policies, which can be determined by operators or administrators, is defined as $\mathbb{P}$. In practice, one policy can be applied to multiple flows. For a policy $p_i \in \mathbb{P}$, properties $p_i.sep$ and $p_i.dep$ define the Source End Point (SEP) and Destination End Point (DEP) [5], which refer to a device or an application that is the ultimate origination or destination entity of specific traffic. In addition, $p_i.chain$ defines the sequence of NFs that all flows matching policy p_i should traverse in order, e.g., $p_i.chain = \{n_1, n_2, n_3\}$, where, for example, $n_1.type =$

$FW, n_2.type = IPS, n_3.type = Proxy$. And, $p_i.len$ is the length of $p_i.chain$.

All NF instances in $p_i.chain$ must be assigned to appropriate NFBs beforehand, and we assume there are enough NFBs to accommodate all required NFs in data center. Since we consider heterogeneous NFBs, there are various possible locations for each NF in $p_i.chain$. For example, in the above example of p_i, $n_1.location$ could be a core router, $n_2.location$ could be a hardware middlebox, and $n_3.location$ could be an NFV server. An example of service chain for north-south traffic in DC is given in Fig. 4.

3.3 Cost with Network Functions

3.3.1 Cost Definition

Since NFs may be placed on NFBs with different locations and processing capability, the cost introduced by NFs placement (service function chaining) for a policy are varied. Many performance metrics can be used to measure such cost, for example, latency, bandwidth. For a policy p_i, its cost is defined as follows:

$$Cost(p_i) = \alpha \cdot T(p_i) + \beta \cdot BW(p_i) + \dots \quad (1)$$

where $T(p_i), BW(p_i), \dots$ are cost functions for each metric. For example, $T(p_i)$ is the cost on latency of the whole service chaining. $BW(p_i)$ is the cost calculated by bandwidth, which can be set inversely proportional to bandwidth of service chain path. Parameters $\alpha, \beta, \dots$ are weight factors for each metric. Both cost functions and corresponding weight values can be determined by administrators according to application scenarios and requirements. And the total cost of a policy $Cost(p_i)$ can be either calculated on single metric or combination of multiple metrics.

3.3.2 Example: Cost on Delays

We will show an example of how to calculate the cost of a policy based on latency, i.e., $Cost(p_i) = T(p_i)$. For simplicity, we define it as the total expected delay for a policy, including the transmission delay among adjacent NFs in the service chain and processing delay of all NFs in the chain (see Equation 3). These two components are explained below:

(1) Transmission delays

In order to steer traffic to the service chain, either Policy Based Routing (PBR) or VLAN stitching can be used in data centers [5] [23]. For either case, the intended solution in this paper should be unaware of these schemes and is general and applicable to these schemes. So, we do not consider the detailed routing schemes between two NFBs.

Since, in production data centers, the transmission delay of links in its path are relatively stable and can be easily obtained/estimated through large-scale measurement [24], we assume the transmission delay among NFBs are known and can be obtained through the controller. The controller will maintain a transmission delay matrix D. And, $D(a, b) = D(b, a)$ is the delay between node a and b. $D(a, b) = \infty$ if the delay is unknown or they are unreachable and, in either cases, these routes will not be considered for service chains.

(2) Processing delays of NFs

For an NF, say n_i, we define its service time $t_p(n_i)$ as the time that n_i takes to process a packet. Since that many NFs, such as firewalls and load balancers, only process packet headers of which sizes are fixed, ignoring variable length data payloads. For NFs that check packet payloads, e.g., firewalls and IDS/IPS, their service time can be estimated by assuming the largest packet size in the worst case, e.g., size MTU of the network or MSS of TCP segments. Thus, the service time $t_p(n_i)$ can be treated as a constant [25]. Specially, considering the processing capacity $n_i.cap$ of n_i, $t_p(n_i) = 1/n_i.cap$.

3.3.3 Remarks

The $Cost(p_i)$ is to evaluate the deployment cost of a policy. Many recent studies have consistently shown that latency is the most important revenue related performance metric in data centers [24] [26] [27] [28]. Thus, in the following sections of this paper, we will primarily focus on the latency (or delay) of a policy flow. However, the main idea in this paper can be easily applied to other metrics as shown above.

NFBs are considered over-provisioned in the network due to two reasons. First, the heterogeneity NFBs, as discussed in Section 1, can provide a large solution space for optimal chaining of NFs without oversubscribing any individual NFBs. Second, the computational resources and middleboxes are often overprovisioned rather than oversubscribed [29]. Moreover, when NFV is implemented, administrators can easily create new instances on commodity servers to handle large service requirements [30].

3.4 Re-ordering of Service Chain

To investigate more opportunity to optimize service chain schedule, we have surveyed a wide range of common NFs and service chains to understand their common behaviors and properties. Most of these NFs perform limited types of processing on packets, e.g., watching flows but making no modification, dropping packets, changing packet headers and/or payload. For example, in the simplest case, a flow monitor (FlowMon) obtains operational visibility into the network to characterize network and application performance, and it never modifies packets and flows [5]. Some types of NFs, e.g., IDS, will check packet headers and payload, and raise alerts to the system administrator. NFs, such as firewalls and IPS, usually do not change packet headers and payload, but they may use packet header information to make decision on whether to drop or forward the packet. Some NFs (e.g., NAT and LB) may check IP/port fields in packet headers and rewrite these fields for processing [9]. Others (e.g., traffic shaper) do not modify packet headers and payloads, but may perform traffic shaping tasks, such as active queue management or rate limiting [18].

Due to the nature of the functions applied, certain ordering of NFs naturally exists for a service chain. For instance, a service chain, which is applied to north-south traffic in datacenters, is composed by a VPN and a Web Optimization Control (WOC). Normally, the WOC is not effective on VPN traffic, requiring VPN termination prior to WOC [5]. For other service chain with IDS and FlowMon, since IDS never change the packet content, FlowMon can be applied to the traffic after IDS or placed prior to IDS.

In order to model these properties of NFs behaviours and leverage those properties to optimize service chain

TABLE 2: Examples of the dynamic actions performed by different NFs that are commonly used today [9]

Network Functions	Input	Actions	Type
FlowMon	Header	No change	Static
IDS	Header Payload	No change	Static
IP Firewall	Header	Drop?	Dropper
IPS	Header Payload	Drop?	Dropper
NAT	Header	Rewrite header	Modifier
Load balancer	Header	Rewrite header	Modifier
Redundancy eliminator	Payload	Rewrite payload	Modifier

schedule among heterogeneous NFBs, we can classify NFs into several classes according to their behaviours:

- Modifier: NFs that may modify the content of a packet (header or payload), e.g., NAT, Proxy.
- Dropper: NFs that may drop packets of flows, but never modify header of payload of packets, e.g., firewall.
- Static: NFs do not modify the packet or its forwarding path, and in general do not belong to any classes above, e.g., FlowMon, IDS.

Table 2 provides a summation of the dynamic actions performed by different types of NFs that are commonly used today. In addition to the three types above, some NFs may change the rate of flows, e.g., traffic shaper. These NFs only change the interval time among consecutive packets and do not modify content of packets. Since we mainly focus on NFs operations on each packet, they will be treat the same as static NFs.

To preserve the correctness of service chains, users can specify constraints on the order of NFs in service chains. For example, the order of consecutive static NFs can be switched. However, static NFs can not be moved across Modifiers, as this might lead to incorrect operations.

Considering such re-ordering of service chain, we define $p_i.set$ to be a set of all possible NFs sequences of the service chain, i.e., $p_i.set = \{l_1, l_2, \ldots\}$. For example, suppose the service chain of p_i is $FW_1 \rightarrow IDS_1 \rightarrow FlowMon_1$, and the position of IDS_1 and $FlowMon_1$ can be swapped. Then, $p_i.set = \{l_1 = (FW_1, IDS_1, FlowMon_1), l_2 = (FW_1, FlowMon_1, IDS_1)\}$. NFs in $p_i.chain$ can be organized according to any sequence defined in $p_i.set$. Moreover, p_i is called *satisfied* if and only if the following condition holds, i.e., the final assigned sequence of policy p_i must be equal to one accepted list in $p_i.set$ if re-ordering is allowed:

$$p_i.chain[j] == l[j], \forall j = 1, 2 \ldots, p_i.len, \exists l \in p_i.set \quad (2)$$

Specially, if $p_i.set$ contains only one list and equal to $p_i.chain$, it means that re-ordering is impossible or disabled by users.

3.5 Heterogeneous Network Policy Placement Problem

In this paper, since we use the latency as an example metric to measure the performance of service chaining. We only consider the delay of a policy p_i, denoted by $T(p_i)$, for the cost function $Cost(p_i)$. Thus, based on previous analysis, the expected delay for the flow constrained by policy p_i can be defined as:

$$\begin{aligned} Cost(p_i) = T(p_i) &= D(p_i.sep, p_i.chain[1]) \\ &+ \sum_{j=1}^{p_i.len-1} (D(p_i.chain[j], p_i.chain[j+1]) + t_p(p_i.chain[j])) \\ &+ D(p_i.chain[p_i.len], p_i.dep) \end{aligned} \quad (3)$$

We aim to reduce the total delay by efficiently placing NFs onto heterogeneous NFBs while strictly adhering to network policies. Denote A to be an allocation of NFs to NFBs and $A(b_j)$ is the set of NFs hosted on b_j.

The *Heterogeneous Network Policy Placement problem* is defined as follows:

Definition 1. *Given the set of policies $\mathbb{P}$, NFBs $\mathbb{B}$ and delay matrix D, we need to find an appropriate allocation of NFs A, which minimizes the total expected end-to-end delays of the network:*

$$\begin{aligned} \min \ & \sum_{p_k \in \mathbb{P}} Cost(p_k) \\ s.t. \ & p_k \text{ is satisfied}, \forall p_k \in \mathbb{P} \\ & n_i.location \neq \emptyset, \forall n_i \in p_k.chain, \forall p_k \in \mathbb{P} \\ & n_i.type \in n_i.location.typeset, \forall n_i \in p_k.chain, \forall p_k \in \mathbb{P} \\ & \sum\nolimits_{n_i \in A(b_j)} n_i.cap \leq b_j.cap, \forall b_j \in \mathbb{B} \end{aligned} \quad (4)$$

The first two constraints ensure that NFs of all service chains are appropriately accommodated by heterogeneous NFBs. The third constraint ensures that types of NFs and NFBs must be consistent. The forth constraint is the capacity constraint of all NFBs. The above problem can be easily proven to be *NP-Hard*:

Proof. To show that *Heterogeneous Network Policy Placement problem* is NP-Hard, we will show that the Multiple Knapsack Problem (MKP) [31], whose decision version has already been proven to be strongly NP complete, can be reduced to this problem in polynomial time.

Consider the following input for the MKP problem: there are $|M|$ items, and the item size of $m_i \in M$ is $m_i.req$. There are $|K|$ knapsacks, the $k_i \in K$ has limited capacity of $k_i.cap$. The profit of assigning m_i to k_j is f_{ij}. The objective of MKP is to maximizing the total profit.

If we take each item m_i to be an NF n_i, where $m_i.req = n_i.cap$. Each knapsack k_j is regarded as an NFB b_j that $k_j.cap = b_j.cap$. Consider a special case that each service chain of all policies contains only one NF. And the profit f_{ij} is corresponding to the negative of the cost of assigning n_i to b_j. Suppose that the cost between servers and NFBs are the same. And there are enough NFBs to hold all NFs, meaning that no NFBs are saturated. Thus, the MKP problem becomes a special cases of the *Heterogeneous Policy Placement problem*. And the objective is to minimizing the total cost, or maximizing the total profit.

Therefore, the MKP problem is reducible to the *Heterogeneous Policy Placement problem* in polynomial time, and hence the *Heterogeneous Policy Placement problem* is NP-hard.

□

4 HETEROGENEOUS POLICY ENFORCEMENT

In this section, we introduce *HOPE*, a Heterogeneous netwOrk Policy Enforcement scheme which utilize of the heterogeneity of NFBs.

4.1 Optimal Service Chain Path

We consider an online solution which processes one service chain at a time when a new policy requirement arrives, say p_i. The problem is to find appropriate NFBs to accommodate all NFs of p_i, ensuring that p_i is *satisfied*, with an objective to minimize its total cost $Cost(p_i)$. Let NFBs, which are able to host an NF n_j, be defined as:

$$B(n_j) = \{b_k | n_j.type \in b_k.typeset \text{ and } \sum_{n' \in A(b_k)} n'.cap + n_j.cap \leq b_k.cap, \forall b_k \in \mathbb{B}\} \quad (5)$$

Considering the (re-)ordering constraints and heterogeneous capability among different NFBs, the optimal service chain path with the smallest expected cost for a policy *does not* need to be the shortest path from source to destination end point.

Furthermore, two NFs in the chains of p_i may be assigned to the same NFB and share the same capacity, making the problem much more complex and challenging. For example, suppose both two NFs of p_i, say n_1 and n_2 ($n_1 \prec_l n_2, \exists l \in p_i.set$)[1], can be placed in NFB b, but the residual capacity of b can only accept one of them. If we assign n_1 to b, b would be unable to accept p_i later. In this case, n_1 and n_2 are called *conflict nodes*. A service chain path from the source will be *blocked* by the latter one of the *conflict nodes*.

Hence, we design the OSP (Optimal Service chain Path) algorithm to find the optimal allocation in such situation, as shown in Algorithm 1. Set S contains all NFBs whose final optimal distance from the source have already been determined and is initialized to be empty in line 1. Set Q refers to all NFBs that might be used host remaining NFs in the policy. Since some NFBs may host multiple types of NFs, it is possible that an NFB may appears in both S and Q. All NFBs that are able to host NFs in the policy, determined by Equation (5), are added to Q initially. The $d(v, j)$ is used to maintain the current length of the service chain path from source to jth NFs in p_i, which is hosted by NFB v. It is initialized to be infinite and will be relaxed during the course of the algorithm. All temporary solution are kept in $prev(v, j)$, which indicates the previous hop of the current optimal path from v to the source $p_i.sep$. During each while loop, at most one node u can find the optimal path to source and be added to set S (line 11). Then, other related nodes can relax their optimal distance to the source through u (line 15∼23). Conflict nodes are handled in line 18. The optimal service chain path is maintained in $prev$ and can be obtained through function $getPath()$. If re-ordering of service chain is enabled, some operations in Algorithm 1 must be aware of the index of candidate chain in $p_i.set$, making sure they are consistent, for example, $d(u, j)$, $prev(v, j)$ and $getPath()$.

1. $n_1 \prec_l n_2$ denotes that n_1 appears before n_2 in the ordered list l.

Algorithm 1 HOPE-OSP: Optimal Service Chain Path

Input: $p_i, \mathbb{B}, \mathbb{N}, D$
Output: Optimal service chain path for p_i

1: $S \leftarrow \emptyset$
2: $Q \leftarrow \bigcup_{l \in p_i.set, 1 \leq j \leq p_i.len} B(l[j])$
3: $d(v, j) \leftarrow \infty, \forall v \in Q, \forall j = 1, 2, \ldots, p_i.len$
4: $prev(v, j) \leftarrow$ undefined, $\forall v \in Q, \forall j = 1, 2, \ldots, p_i.len$
5: $d(p_i.sep, 0) \leftarrow 0$
6: **while** $Q \neq \emptyset$ **do**
7: $\quad (u, j) \leftarrow \arg\min_{u \in Q, 1 \leq j \leq p_i.len} d(u, j)$
8: $\quad$ **if** $u = p_i.dep$ **then**
9: $\quad\quad$ **break**
10: $\quad$ **end if**
11: $\quad S \leftarrow S \cup \{u\}$
12: $\quad$ **if** $u \notin B(l[k]), \forall l \in P_i.set, \forall k > j$ **then**
13: $\quad\quad Q = Q \setminus \{u\}$
14: $\quad$ **end if**
15: $\quad n_k \leftarrow$ NF in $p_i.set$ that will be placed in u
16: $\quad$ **for each** $v \in \bigcup_{l \in p_i.set} B(l[j+1])$ **do**
17: $\quad\quad$ **if** $d(v, j+1) > d(u, j) + D(u, v) + t_p(n_k)$ **then**
18: $\quad\quad\quad$ **if** $v \notin$ GETPATH(u, j) **or** $\sum_{n_x \in A(v)} n_x.cap + n_k.cap \leq v.cap$ **then**
19: $\quad\quad\quad\quad d(v, j+1) \leftarrow d(u, j) + D(u, v) + t_p(n_k)$
20: $\quad\quad\quad\quad prev(v, j) \leftarrow u$
21: $\quad\quad\quad$ **end if**
22: $\quad\quad$ **end if**
23: $\quad$ **end for**
24: **end while**
25: **return** GETPATH$(p_i.dep, p_i.len)$

26: **function** GETPATH$(node, index)$
27: $\quad path \leftarrow \emptyset$
28: $\quad u \leftarrow node$
29: $\quad j \leftarrow index$
30: $\quad$ **while** $prev(u, j)$ is defined **and** $j > 0$ **do**
31: $\quad\quad$ insert u at the beginning of $path$
32: $\quad\quad u \leftarrow prev(u, j)$
33: $\quad\quad j \leftarrow j - 1$
34: $\quad$ **end while**
35: $\quad$ insert u at the beginning of $path$
36: $\quad$ **return** $path$
37: **end function**

Lemma 1. *In Algorithm 1, when an NFB, say u, is added to S for the jth NFs in p_i, $d(u, j)$ is the shortest path length from $p_i.sep$ to u.*

Proof. If there is no *conflict nodes* in the current service chain path, Algorithm 1 becomes a variation of Dijkstra algorithm and Lemma 1 can be easily proved to be always hold.

Otherwise, denote $p_i.sep$ to be s and let $\delta_j(s, u)$ be the distance of the shortest path from s to u, which hosts the jth NF of p_i. Assume u is the first node for which $d(u, j) \geq \delta_j(s, u)$, when it is added to S. We must have $u \neq s$ because that s is the first node added to S and $d(s, j) = \delta_j(s, s) = 0$. Because $u \neq s$ and $S \neq \emptyset$ before u is added to S, there is at least one shortest path from s to u. Suppose y is the first node along this optimal path such that $y \in Q$, hosting the

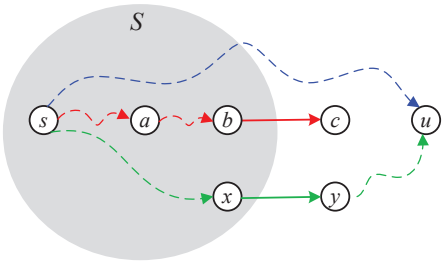


Fig. 5: Example for proof of Lemma 1

kth NF of p_i ($k < j$). Let $x \in S$ be y's predecessor. Because u is added to S before y, $d(u, j) \leq d(y, k)$.

So, the optimal path can be decomposed as $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$. See Fig. 5 for an example. Note that there should be no conflict nodes for y on this path, and this path may or may not go through edge $a \rightarrow b$.

Because y occurs before u on the optimal path from s to u and all edge weights and costs are non-negative, we have $\delta_k(s, y) \leq \delta_j(s, u)$.

As u is chosen as the first node for which $d(u, j) \neq \delta_j(s, u)$ when it is added to S and $x \in S$, we have $d(x, k-1) = \delta_k(s, x)$ when x was added to S. Edge $x \rightarrow y$ was relaxed at that time (line 16 $\sim$ 23). So, $d(y, k) = \delta_k(s, y) \leq \delta_j(s, u) \leq d(u, j)$.

Thus, $d(u, j) = d(y, k) = \delta_j(s, u)$, which contradicts our choice of u. □

According to Lemma 1, we can easily derived that

Theorem 1. *Algorithm 1 can always output a shortest service chain path.*

The complexity of the algorithm depends on the way of finding the (u, j) with the smallest cost $d(u, j)$. Let $m = |\mathbb{B}| \cdot |p_i.set| \cdot p_i.len$ to be all possible NFB elements of (u, j). Because paths with *conflict nodes* failed to reach the destination, not all elements are checked in Algorithm 1. Thus, each elements (u, j) is checked at most once (line 7 $\sim$ 14), and the neighbor of each element is examined in the for loop of lines 16 $\sim$ 23 at most once during the course of the algorithm.

Considering the operations of distance in Algorithm 1, a *priority queue* [32] to implement efficient operations. Priority queue is a data structure consisting of a set of item-key pairs, i.e., distance for each node in Algorithm 1: *insert*, e.g., implicit in line 3; *extract-min*, returning the element with the minimum distance in line 7, i.e., the $\arg\min$ operation; and *decrease-key*, decreasing the distance of a given element in line 19. Furthermore, *Fibonacci heaps* [33] implement *insert* and *decrease-key* in $O(1)$ amortized time, and *extract-min* in $O(\log n)$ amortized time, where n is the number of elements in the priority queue. So, by using *Fibonacci heaps*, the running time of Algorithm 1 is $O(m^2 + m \log m)$. Considering that $p_i.len \ll |\mathbb{B}|$ usually, the running time for optimal service chain path is acceptable.

4.2 Greedy Approach

Algorithm 1 ensures the optimality of the service chain path. However, it has one major drawback that its $O(m^2 + m \log m)$ time complexity. Thus, we also propose a greedy approach for HOPE, which trades off small accuracy for significantly faster speed.

Algorithm 2 HOPE-Greedy

Input: $p_i, \mathbb{B}, \mathbb{N}, D$
Output: Service chain path for p_i

1: $path \leftarrow \emptyset$
2: $B'_j \leftarrow B_j, \forall j = 1, 2 \ldots, p_i.len$
3: $j \leftarrow 1$
4: **while** $j \leq p_i.len$ **do**
5: **if** $B'_j = \emptyset$ **then**
6: **if** $j = 1$ **then**
7: $path \leftarrow \emptyset$ ▷ no available path
8: **break**
9: **end if**
10: remove last node in $path$
11: $B'_j \leftarrow B_j$
12: $j \leftarrow j - 1$
13: **continue**
14: **end if**
15: $(u, n_k) \leftarrow \arg\min_{u \in B'_j} D(path[end], u) + t_p(n_k)$ ▷ n_k is the jth NF of p_i that will be placed in u
16: $B'_j \leftarrow B'_j \setminus \{u\}$
17: **if** $u \notin path$ **or** $\sum_{n' \in A(u)} n'.cap + n_k.cap \leq u.cap$ **then**
18: append u at the end of $path$
19: $j \leftarrow j + 1$
20: **end if**
21: **end while**
22: **return** $path$

Algorithm 2 describes detailed steps of the greedy approach of HOPE. The main idea is that: starting from the first element in the policy, for each NF in the service chain, the algorithm will choose an NFBs with the smallest delay to the source end point or the previous NF in the service chain. If the current path is *blocked* by a *conflict node*, the algorithm will fall back to the previous NF and choose the NFB with the second smallest delay and so on. This process will continue until the destination end point is reached, or there is no available path. If service chain re-ordering is enabled and multiple candidate service chain are available in $p_i.set$, the B_j contains all acceptable NFBs defined in Equation 5. In this case, similar to Algorithm 1, the index of chain in $p_i.set$ should be consistent during operations of Algorithm 2. Specially, for any $l_1 \in p_i.set$ and $l_2 \in p_i.set$ ($l_1 \neq l_2$), if $l_1[j] = l_2[j]$, same NFBs obtained for l_1 and l_2 can be merged as a single node, otherwise, they will be treated as different nodes.

4.3 Stability and Adaptability

A single network flow may change dynamically. However, a network policy, such as "all traffic should be checked by application firewall", is applied to a group of traffic flows once it is defined. Hence, network policy is often stable and is not transient. As HOPE works on per network policy rather than per traffic flow level, we expect our HOPE scheme will be equally stable.

Algorithm 3 HOPE Operation

```
1: P_H ← ∅
2: Set timer T_update
3: loop
4:     event ←getEvent()
5:     switch event do
6:         case new_policy
7:             p_i ←new arrived policy
8:             Insert p_i at head of P_H
9:             Trigger event schedule_policy
10:        case timer_fires
11:            CollectInfo()        ▷ Pull network information
12:            P_temp ← all affected policies
13:            P_H ← P_H ∪ {P_temp}
14:            Update timer T_update
15:            Trigger event schedule_policy
16:        case schedule_policy
17:            while P_H ≠ ∅ do
18:                p_i ←Head(P_H)  ▷ Obtain head item of P_H
19:                path ←HOPE-OSP() Or HOPE-Greedy()
20:                Arrange p_i according to path
21:                P_H ← P_H \ {p_i}
22:            end while
23:    end switch
24: end loop
```

In the meantime, we also reckon the fact that traffic demand could change slowly over time and it is necessary to adapt to the changes to ensure optimality. In response to such dynamics, HOPE uses a set P_H to keep all policies that need to be processed. The *Policy Controller* of HOPE can periodically (controlled by a timer T_{update}) poll switches for traffic statistics to look for the changes in traffic demand in specific part of network topology, and then all policies that have been affected will be added to P_H. New arrived policies will be placed at head of the P_H so that they can be processed immediately with higher priority.

The detailed operations of HOPE are described in Algorithm 3. Initially, P_H is set to be empty and timer T_{update} is initialized. The function getEvent() will be blocked until an event occurs. Three *events* are defined: *new_policy* refers to arrival of policy configuration from users, *timer_fires* is triggered when T_{update} expires and *schedule_policy* let HOPE call either HOPE-OSP() or HOPE-Greedy() to process policies in P_H. Particularly, both *new_policy* and *timer_fires* will trigger *schedule_policy* event, i.e., line 9 and 15. New arrived policy request will be placed at the head of P_H (line 8), while others, e.g., existing policies affected by networking dynamics, are appended at the end of P_H (line 13). CollectInfo() is used to collect networking and user requirement information for detecting any dynamic changes.

5 HOPE Implementation

We have implemented some core components in Ryu controller and Mininet environment.

5.1 Policy Controller

The policy controller is needed to execute HOPE scheme, as well as managing NFBs and policies. In current implementation, the policy controller is implemented as an application module in Ryu. One reason to chosen Ryu is that it has a built in integration for Snort [15], which enables bidirectional communication using unix domain sockets. The policy controller interacts with NFBs that host firewalls using OpenFlow protocol. Although frameworks such as OpenNF [11] can also be added to enrich functionality of the controller, we note that the scope of this paper is to provide a proof-of-concept implementation rather than a full-blown testbed.

Besides managing NFBs and accepting policy configuration from network operators/administrators, the policy controller is also responsible for collecting link latency from Pingmesh Agent and maintaining an in-memory all-pair unidirectional end-to-end latency table which is essential to the HOPE scheme.

5.2 Link Latency

HOPE needs to obtain link latency to construct the transmission delay matrix D. So, we have implemented a reduced version of Pingmesh Agent [24] using C++. This Pingmesh Agent pings all servers using *TCPing*, and measures round-trip-time (RTT) from the TCP-SYN/SYN-ACK intervals. This module is light-weighted. Our primary measurement results shows that the average memory footprint is less than 2MB, and the average CPU usage is less than 1%. Ping traffic is very small and ping interval is configurable according to actual needs.

Each agent will upload collected results to the controller periodically for constructing all pairs end-to-end latency table, i.e., the delay matrix D, which can be queried by HOPE later. This is because we assume that most of deployed NFs will run in commodity servers. There are also some in-network hardware NFBs, as defined in Section 3.1, which are either SDN switches or attached directly to switches. In this case, the delay from/to these particular devices can be queried through OpenFlow's port statistics APIs or other techniques such as OpenNetMon [34].

As described in Section 3.3, the processing delay of NFs is obtained from $t_p(n_i)$, which is inverse proportional to NF's capacity. We do not consider queuing delay in our testbed implementation because HOPE ensures that NFBs are not overloaded. Moreover, as explained in Section 3.3.2, the computational resources and middleboxes are often overprovisioned rather than oversubscribed [29]. We also note that there are also some other techniques which are useful for monitoring processing capacities such as sFlow [35].

6 Evaluation

6.1 Evaluation Environment and Setup

In addition to testbed implementation, we have also extensively evaluated the performance of HOPE scheme at scale in *ns*-3.25 on a Ubuntu 14.04.5 server with Intel(R) Xeon(R) E5-2670 2.3GHz CPU and 8GB of memory. We simulated a fat-tree data center topology with factor k ranged from 4 to 20, meaning that there are at most 2000 servers and 500 switches in these setups. Each link transmission latency in the network is randomly distributed from tens to hundreds of microseconds [36].

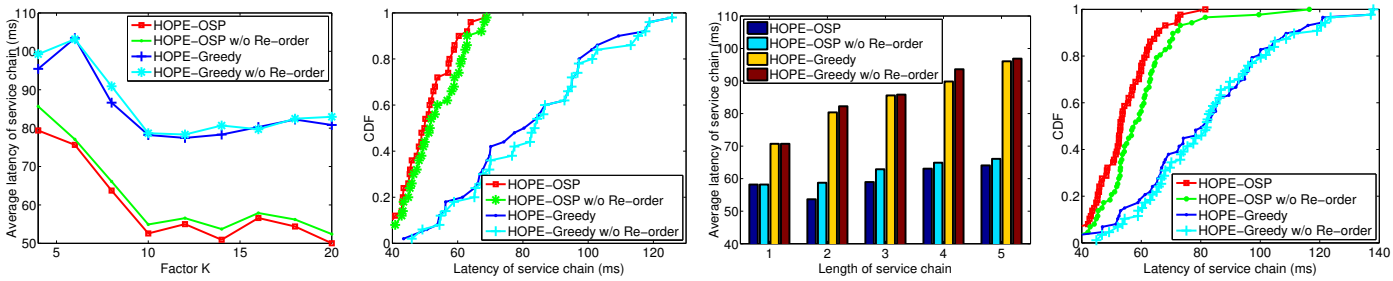


Fig. 6: Comparison of latency of service chain: (a) Average latency for various network scale; (b) Latency of service chain for $k = 20$; (c) Average latency for various length of service chain; (d) Latency of service chain for $length = 4$

In all experiments, traffic flows are randomly generated to transmit packets between two servers. Each flow is required to traverse a sequence of various NFs – the service chain – before being forwarded to their destinations as specified by policies. In our experiments, each service chain is comprised of 1∼5 NFs (normal distribution) including FW, IDS, RE, LB, NAT, Proxy and (traffic) Monitor [5]. Specially, IDS and Monitor belongs to static NFs while others belongs to Modifier or Dropper (see Table 2). The processing requirement of NFs are generated randomly in *pps* (see Section 3.2). Thus, the processing latency of NFs are varied accordingly.

Each NFB is modeled with random capability, e.g., residual capacity (number of packets it can process per second). Each NFB is also assigned a set of NF types that it supports. Therefore, an NFB can accept a NF as long as it has sufficient residual capacity and the NF's type is amongst its support list. NFBs are deployed in the network, including OpenFlow switches, hardware middleboxes and NFV servers. The set of NF types supported by NFBs are determined by their implementations. NFV servers are designed to support all types of NFs, while hardware middleboxes are set to support only one type of NFs and OpenFlow switches can support few simple NFs, e.g., NAT and LB.

The policy controller (as described in Section 5.1) is also used. For simplicity, the scheme using OSP (Algorithm 1) to achieve optimal schedule for a service chain is referred as HOPE-OSP, and the greedy approach (Algorithm 2) is referred as HOPE-Greedy.

As a comparison, we have also implemented *Branch-and-bound* [37] optimization algorithm to verify the optimality of HOPE-OSP. The *Branch-and-bound* algorithm explores branches of a tree of candidate solutions, and branches are checked against upper and lower estimated bounds on the optimal solution, which is finally obtained through recursive search.

6.2 Evaluation Results

Fig. 6 demonstrates the performance of HOPE with regard to the latency of service chain. By default, re-ordering is enabled for both schemes. We also evaluate the performance difference when re-ordering is disabled (indicated by "w/o Re-order" in the figures).

Fig. 6a shows the average latency of all service chains under different network scales with the factor k of fat-tree ranging from 4 to 20. We observed that *Branch-and-bound* curves completely overlap with that of HOPE-OSP, hence are omitted for readability in Fig. 6. However, overlapped curves mean that HOPE-OSP can always find a service chain path with the same latency as that of *Branch-and-bound*, which is optimal. This is in line with our theoretical proof in Theorem 1 that HOPE-OSP can always output the optimal service chain path for a policy. In comparison, the HOPE-Greedy approach could fall behind both HOPE-OSP and *Branch-and-bound* by up to 29.4%. Re-ordering can be optionally turned off, and our results show that this can slightly increase total latency by 4%. However, for HOPE-Greedy, there is no obvious effect whether the re-ordering of service chain is enabled or not. A detailed breakdown view is further shown in Fig. 6b for all policies for a large scale network when $k = 20$. Particularly, the HOPE-OSP can outperform HOPE-Greedy scheme by 83% at the 99th percentile.

Fig. 6c reveals that average latency increases linearly with the length of service chain when all NFBs have sufficient capacity for accommodating all NFs. Obviously, when there is only one NF in the service chain, re-ordering is not needed. The breakdown of CDF for latency of service chain whose length is comprised of four NFs shown in Fig. 6d. It unveils that amongst HOPE's two algorithms, HOPE-OSP outperforms HOPE-Greedy by 33.2%. And disabling re-ordering can degrade the performance of HOPE-OSP by 8.6%. We also notice that disabling re-ordering can give HOPE-OSP a long tail, meaning that large latency can occur in the high percentile region.

Next we study the performance of different schemes in terms of system runtime in the Ryu controller. This is essentially to test the performance of HOPE controller for its efficiency and scalability in cloud data center environment. Fig. 7 shows the average total running time to process a policy increases exponentially for all schemes. Nevertheless, as we can see from this figure that HOPE-Greedy without re-ordering is the most efficient methods, consuming only $0.74ms$ for $k = 20$ to process a policy. This is because HOPE-Greedy scheme has the smallest search space. On the contrary, HOPE-OSP can complete one policy placement cycle for at $2.8ms$ and *Branch-and-bound* needs nearly 1s. By disabling re-ordering, running time of HOPE-OSP can be improved by 37% on average. Among HOPE-OSP and HOPE-Greedy, the latter is almost 2.5 times more efficient

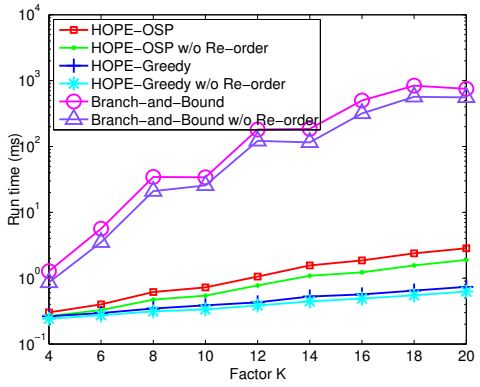


Fig. 7: Performance comparison on running time

that the former one. This means that processing 20K ACL policies for a large enterprise [3] would merely require 56s and 14.8s for HOPE-OSP and HOPE-Greedy respectively (comparing to approximately 5.5 hours required by *Branch-and-bound* optimization).

7 Related Works

Network configurations and management are complex tasks and usually governed by network policies. When deployed in the network, a policy is translated and implemented as one or more packet processing rules in a diverse range of "middleboxes" such as WAN optimizers, load balancers, IDS/IPS, application acceleration boxes, network- and application-level firewalls, and application-specific gateways [38] [39]. Middleboxes are critical part of today's networks and it is reasonable to expect that they will remain so for the foreseeable future [40].

SDN and NFV have enabled more flexible middlebox deployments over the network while still ensuring that specific subsets of traffic traverse the desired set of middleboxes in a predefined order [41]. For example, Chaithan et al. [3] tackled the problem of automatic, correct and fast composition of multiple independently specified network policies by developing a high-level Policy Graph Abstraction (PGA). Anat et al. [18] presented OpenBox, which decouples the control plane of middleboxes from their data plane, and unifies the data plane of multiple middlebox applications using entities called service instances. There are also many other works on correct policy compositions and enforcement [9] [10], consolidating policy rules to end hosts [42] and network switches [43], or providing a framework for migrating middleboxes states [11], or policy-aware application placement to incorporate policy requirements [38] [44] [45].

In the meantime, with network programmability enabled by SDN and NFV technologies, such policy rules can also be implemented outside of traditional "middleboxes" in network switches [46] as well as end-hosts [42]. Jackson et al. [8] described the design of SoftFlow, an extension of Open vSwitch designed to bring tightly integrated middleboxes to network virtualization platforms. One of the design requirements for today's cloud data centers is to support the insertion of new middleboxes [47].

However, given the large variety of NF entities in terms of both types (e.g., hardware middleboxes, NFV and OpenFlow switches) and network locations, inappropriate selections not only eliminate the advantage of SDN and NFV but could also cause severe consequences including data center outage.

Furthermore, most data center applications are sensitive to network latencies. These latencies can be introduced by network congestion as throughput-intensive applications cause queuing at switches that delays traffic from latency-sensitive applications. Existing techniques to combat queuing are to prioritize flows such that packets from latency-sensitive flows can "jump" the queue [48]; to centrally schedule all flows for every server so no flows will have to queue [49]; or to pace end host packets to achieve guaranteed bandwidth for guaranteed queuing [50].

8 Discussions

HOPE aims to enforce policy schedule in heterogeneous NFBs environment. In this paper, delay is used as an example to demonstrate how HOPE design would work. In practice, other metrics such as bandwidth and throughput can also be included. The detailed definition of Equation 1 need to be determined by administrators according to application scenarios and requirements.

As data center resource including computation power and network bandwidth is often overprivisioned [29], HOPE exploits this fact and assumes that NFBs resources are over-provisioned in the networks, such as OpenFlow switches and NFV servers, ensuring that NFBs are not overloaded. However, in case of burst traffic, temporary network congestion might happen, increasing delay of the service chain. Currently, such situation could be captured by the *Policy Controller* and reflected during the periodically cost calculation, as shown in Section 4.3. More other improvements could be considered in future, e.g., more elaborated optimizing scheme with traffic prediction and queue modeling.

9 Conclusions

In today's data centers, network functions can be deployed in different implementations of NFBs. For example, in addition to hardware middleboxes, network functions can also be implemented on OpenFlow switches and NFV servers. Such heterogeneous environment of NFBs for policy allocation remain unexplored in previous research works. Thus, in this paper, we study the Heterogeneous Policy Enforcement Problem with a focus on the latency. We first analyze and model the optimization problem, which is shown to be NP-Hard. And then, we simplified the problem and proposed the HOPE scheme, which is proved to be able to find the optimal service chain path for each policy. An efficient greedy approach of HOPE is also proposed and discussed. Extensive evaluation results and comparisons with Branch-and-bound approach have demonstrated high effectiveness and optimality of HOPE.

Acknowledgments

This work has been partially supportedby Chinese National Research Fund (NSFC) No. 61772235 and 61502202;

the Fundamental Research Funds for the Central Universities 21617409; the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1; FDCT 0007/2018/A1, DCT-MoST Joint-project No. 025/2015/AMJ of SAR Macau; University of Macau Funds No. CPG2018-00032-FST & SRG2018-00111-FST; NSFC Key Project No. 61532013; National China 973 Project No. 2015CB352401; 985 Project of Shanghai Jiao Tong University: WF220103001; Natural Science Foundation of Guangdong Province No. 2017A030313334; and American University of Sharjah.

REFERENCES

[1] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan *et al.*, "Steering: A software-defined networking for inline service chaining," in *21st IEEE International Conference on-Network Protocols (ICNP)*. IEEE, 2013, pp. 1–10.

[2] P. Quinn and T. Nadeau, "Problem Statement for Service Function Chaining," RFC 7498, Apr. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7498.txt

[3] 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.

[4] A. M. Medhat, T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci, and T. Magedanz, "Service function chaining in next generation networks: State of the art and research challenges," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 216–223, 2017.

[5] S. Kumar, M. Tufail, S. Majee, C. Captari, and S. Homma, "Service function chaining use cases in data centers," Internet Draft, IETF SFC WG, Tech. Rep. draft-ietf-sfc-dc-use-cases-06, February 2017.

[6] N. Huin, A. Tomassilli, F. Giroire, and B. Jaumard, "Energy-efficient service function chain provisioning," *Journal of Optical Communications and Networking*, vol. 10, no. 3, pp. 114–124, 2018.

[7] L. Cui, F. P. Tso, and W. Jia, "Heterogeneous network policy enforcement in data centers," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 552–555.

[8] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker, "Softflow: A middlebox architecture for open vswitch." in *USENIX Annual Technical Conference*, 2016, pp. 15–28.

[9] 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.

[10] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags," in *Proc. NSDI*, vol. 14, 2014, pp. 533–546.

[11] A. Gember-Jacobson, C. P. Raajay Viswanathan, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: enabling innovation in network function control," in *Proc. of ACM SIGCOMM*, 2014, pp. 163–174.

[12] T. Lukovszki, M. Rost, S. Schmid, and S. Schmid, "It's a Match! Near-Optimal and Incremental Middlebox Deployment," *ACM Sigcomm Computer Communication Review*, vol. 46, no. 1, pp. 30–36, 2016.

[13] iperf. [Online]. Available: https://iperf.fr

[14] Electric Sheep Fencing LLC. pfsense. [Online]. Available: https://blog.pfsense.org/

[15] Cisco. Snort. [Online]. Available: https://www.snort.org

[16] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *Proceedings of the 29th Conference on Information Communications*, ser. INFOCOM'10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 1163–1171. [Online]. Available: http://dl.acm.org/citation.cfm?id=1833515.1833691

[17] C. Wang, X. Yuan, Y. Cui, and K. Ren, "Toward secure outsourced middlebox services: Practices, challenges, and beyond," *IEEE Network*, 2017.

[18] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *ACM SIGCOMM Computer Communication Review*. ACM, 2016, pp. 511–524.

[19] W. Ma, J. Beltran, Z. Pan, D. Pan, and N. Pissinou, "SDN-based traffic aware placement of nfv middleboxes," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 528–542, 2017.

[20] H. Mekky, F. Hao, S. Mukherjee, T. Lakshman, and Z.-L. Zhang, "Network function virtualization enablement within SDN data plane," in *IEEE Conference on Computer Communications (INFOCOM 2017), IEEE*. IEEE, 2017, pp. 1–9.

[21] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman, "Application-aware data plane processing in SDN," in *HotSDN*. ACM, 2014, pp. 13–18.

[22] Z. Liu, X. Wang, W. Pan, B. Yang, X. Hu, and J. Li, "Towards efficient load distribution in big data cloud," in *IEEE ICNC*, 2015, pp. 117–122.

[23] H. Huang, P. Li, S. Guo, W. Liang, and K. Wang, "Near-optimal deployment of service chains by exploiting correlations between network functions," *IEEE Transactions on Cloud Computing*, 2017.

[24] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 139–152.

[25] P. Duan, Q. Li, Y. Jiang, and S.-T. Xia, "Toward latency-aware dynamic middlebox scheduling," in *24th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2015, pp. 1–8.

[26] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats *et al.*, "Timely: Rtt-based congestion control for the datacenter," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 537–550.

[27] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 29–42.

[28] S. Liu, H. Xu, L. Liu, W. Bai, K. Chen, and Z. Cai, "Repnet: Cutting latency with flow replication in data center networks," *IEEE Transactions on Services Computing*, 2018.

[29] S. Ayoubi, S. Sebbah, and C. Assi, "A cut-and-solve based approach for the VNF assignment problem," *IEEE Transactions on Cloud Computing*, 2017.

[30] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, 2018.

[31] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems*. Springer Verlag, 2004.

[32] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 11–20.

[33] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[34] N. L. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in openflow software-defined networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–8.

[35] P. Phaal, S. Panchen, and N. McKee, "Inmon corporations sFlow: A method for monitoring traffic in switched and routed networks," RFC 3176, Tech. Rep., 2001.

[36] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review*, vol. 40, no. 4. ACM, 2010, pp. 63–74.

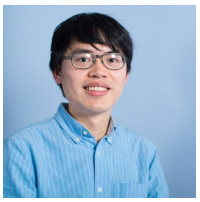
[37] D. Whitaker, "Branch and bound," *Wiley StatsRef: Statistics Reference Online*, 2006.

[38] L. E. Li, V. Liaghat, H. Zhao, M. Hajiaghayi, D. Li, G. Wilfong, Y. R. Yang, and C. Guo, "PACE: Policy-aware application cloud embedding," in *Proceedings of 32nd IEEE INFOCOM*. IEEE, 2013, pp. 638–646.

[39] L. Cui and F. P. Tso, "Joint virtual machine and network policy consolidation in cloud data centers," in *IEEE 4th International Conference on Cloud Networking (CloudNet)*. IEEE, 2015, pp. 153–158.

[40] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 2011, p. 21.
[41] H. Huang, S. Guo, J. Wu, and J. Li, "Service chaining for hybrid network function," *IEEE Transactions on Cloud Computing*, no. 1, pp. 1–1, 2017.
[42] L. Popa, M. Yu, S. Y. Ko, S. Ratnasamy, and I. Stoica, "CloudPolice: taking access control out of the network," in *ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.
[43] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan, "Scalable rule management for data centers." in *NSDI*, vol. 13, 2013, pp. 157–170.
[44] L. Cui, R. Cziva, F. P. Tso, and D. P. Pezaros, "Synergistic policy and virtual machine consolidation in cloud data centers," in *The 35th Annual IEEE International Conference on Computer Communications, IEEE INFOCOM 2016*. IEEE, 2016, pp. 1–9.
[45] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, and W. Zhao, "PLAN: Joint policy-and network-aware vm management for cloud data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1163–1175, 2017.
[46] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 7–12.
[47] L. Avramov and M. Portolani, *The Policy Driven Data Center with ACI: Architecture, Concepts, and Methodology*. Cisco Press, 2014.
[48] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *NSDI 2015*.
[49] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 307–318, 2015.
[50] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: predictable message latency in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 435–448, 2015.

Lin Cui is currently with the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focuses on the following topics: cloud data center resource management, data center networking, software defined networking (SDN), virtualization and so on.



Fun Po Tso received his BEng, MPhil and PhD degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently lecturer in the Department of Computer Science at the Loughborough University. Prior to joining Loughborough, he worked as SICSA Next Generation Internet Fellow at the School of Computing Science, University of Glasgow during 2011-2014 and lecturer in Liverpool John Moores University during 2014-2017. He has published more than 20 research articles in top venues and outlets. His research interests include: network policy management, network measurement and optimisation, cloud data centre resource management, data centre networking, software defined networking (SDN), distributed systems as well as mobile computing and system.

Song Guo is a Full Professor at Department of Computing, The Hong Kong Polytechnic University. He received his Ph.D. in computer science from University of Ottawa and was a professor with the University of Aizu from 2007 to 2016. His research interests are mainly in the areas of big data, cloud computing and networking, and distributed systems with over 400 papers published in major conferences and journals. His work was recognized by the 2016 Annual Best of Computing: Notable Books and Articles in Computing in ACM Computing Reviews. He is the recipient of the 2017 IEEE Systems Journal Annual Best Paper Award and other five Best Paper Awards from IEEE/ACM conferences.

Weijia Jia is currently a chair Professor at University of Macaua. He is leading currently several large projects on next-generation Internet of Things, environmental sensing, smart cities and cyberspace sensing and associations etc. He received BSc and MSc from Center South University, China in 82 and 84 and PhD from Polytechnic Faculty of Mons, Belgium in 1993 respectively. He worked in German National Research Center for Information Science (GMD) from 93 to 95 as a research fellow. From 95 to 13, he has worked in City University of Hong Kong as a full professor. From 14 to 17, he has worked as a Chair Professor in Shanghai Jiaotong University. He has published over 400 papers in various IEEE Transactions and prestige international conference proceedings.

Kaimin Wei received the PhD degree in computer science and technology from Beihang University, Beijing, China, in 2014. He is currently an associate research professor at the School of Information Technology, Jinan University. His research interests focus on Delay/Disruption Tolerant Networks, Mobile Social Networks, and Cloud Computing.

Wei Zhao is currently with American University of Sharjah. He served as the Rector (President) and Chair Professor at Macao University from 2008 to 2018. Before joining the University of Macau, he served as the dean of the School of Science, Rensselaer Polytechnic Institute. Between 2005 and 2007, he served as the director for the Division of Computer and Network Systems in the US National Science Foundation when he was on leave from Texas A&M University, where he served as senior associate vice president for research and professor of computer science. As an elected IEEE fellow, he has made significant contributions in distributed computing, real-time systems, computer networks, cyber security, and cyber-physical systems.