OpenFunction: An Extensible Data Plane Abstraction Protocol for Platform-Independent Software-Defined Middleboxes

Chen Tian, Ali Munir, Alex X. Liu, Jie Yang, and Yangming Zhao

***Abstract*— The data plane abstraction is central to software-defined networking (SDN). Currently, SDN data plane abstraction has only been realized for switches but not for middleboxes. A data plane abstraction for middleboxes is needed to realize the vision of software-defined middleboxes (SDMs). Such a data plane abstraction should be both platform independent and fully extensible. The match-action abstractions in OpenFlow/P4 have limited expression power to be applicable to middleboxes. Modular abstraction approaches have been proposed to implement middlebox data plane but are not fully extensible in a platform-independent manner. In this paper, we propose OpenFunction, an extensible data plane abstraction protocol for platform-independent software-defined middleboxes. The main challenge is how to abstract packet operations, flow states, and event generations with elements. The key decision of OpenFunction is: actions/states/events operations should be defined in a uniform pattern and independent from each other. We implemented a working SDM system including one OpenFunction controller and three OpenFunction boxes based on Netmap, DPDK, and FPGA, respectively, to verify OpenFunction abstraction.**

***Index Terms*— Software-defined networking, network middleboxes, network function virtualization.**

I. Introduction

THE data plane abstraction is central to Software-Defined Networking (SDN). Abstraction means that heterogenous network devices of different vendors/architectures have common data plane programming interfaces for implementing the functionalities needed by the control plane. Abstraction helps to break device vendor lock-in by allowing third-party software to run on all SDN compliant network devices. In contrast, traditional network devices are black boxes where both hardware and software are tightly coupled as they are typically from the same vendor, which leads to both network ossification and vendor lock-in.

Currently SDN data plane abstraction has only been realized for *switches* by OpenFlow [30], [35] and its advanced version P4 [11], but not for *middleboxes*. A data communication network has two types of devices, *switches* and *middleboxes*. Switches (including routers in the broader sense) provide packet forwarding. Middleboxes provide a wide variety of networking and security functionalities such as Network Address Translation (NAT), Load Balancers (LB), firewalls (FW), WAN optimizers, proxies, IPsec gateways (VPN), and network Intrusion Detection/Prevention Systems (IDS/IPS) [39].

A data plane abstraction for middleboxes is needed to realize the vision of software-defined middleboxes (SDM). SDM provides network operator the ability to dynamically load/unload various network functions without changing the network hardware, similar to what OpenFlow/P4 provides for the switches.

For software-defined middleboxes (SDM), a data plane abstraction should be both *platform-independent* and *fully extensible*. The platform-independent goal decouples the data plane function semantics and the underlying hardware that realizes the network function. This allows any third-party SDM program's data plane to execute at any SDM boxes (*i.e.*, SDM compliant middleboxes) with same semantics but different performance depending on hardware adequacy. Fully extensible goal means that any new middlebox functionality can be defined by an SDM program abstraction, which is critical to enable innovation for middleboxes. For example, this enables us to design a new packet encryption algorithm for VPN, define a new flow state in the data plane for firewall, or subscribe to an event when a specific condition is trigged for IPS.

Network Function Virtualization (NFV) attempts to address the issues of tight hardware/software coupling and hardware vendor lock-in for middleboxes by implementing middlebox functions purely in software running on commodity servers [4], [5]. Unfortunately, NFV middleboxes are software-based, not software-defined. The key weakness of NFV middleboxes is low performance because commodity servers are designed for general computing purpose.

The match-action abstractions defined by OpenFlow/P4 have limited expression power to be applicable to

Manuscript received December 2, 2016; revised June 24, 2017 and February 19, 2018; accepted April 15, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Chen. Date of publication May 2, 2018; date of current version June 14, 2018. This work was supported in part by the National Science and Technology Major Project of China under Grant 2017ZX03001013-003, in part by the Fundamental Research Funds for the Central Universities under Grant 0202-14380037, in part by the National Natural Science Foundation of China under Grant 61772265, Grant 61602194, Grant 61671130, Grant 61671124, Grant 61502229, Grant 61672276, and Grant 61321491, in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, and in part by the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program. The 2-page poster version of this paper titled "OpenFunction: An Extensible Data Plane Abstraction Protocol for Platform-independent Software-Defined Middleboxes" was published in the Proceedings of IEEE ICNP, Singapore, November 2016. *(Corresponding author: Chen Tian.)*

C. Tian and J. Yang are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China (e-mail: tianchen@nju.edu.cn; jieyang@smail.nju.edu.cn).

A. Munir and A. X. Liu are with the Department of Computer Science, Michigan State University, East Lansing, MI 48824 USA (e-mail: munirali@cse.msu.edu; alexliu@cse.msu.edu).

Y. Zhao is with the Department of Computer Science and Engineering, The State University of New York at Buffalo, Buffalo, NY 14260-2500 USA (e-mail: yangming@buffalo.edu).

Digital Object Identifier 10.1109/TNET.2018.2829882

middleboxes. First, for the match abstraction, OpenFlow/P4 limit themselves to match packet headers. OpenFlow can only specify matching conditions over a fixed number of predefined packet header fields. P4 improves over OpenFlow by allowing users to extract customized packet header fields. However, many middleboxes, such as application firewalls and IPsec, need to match packet payload against signatures [26]. Second, for the action abstraction, OpenFlow/P4 limit themselves to a fixed number of actions. However, middleboxes, such as IPsec VPN, often need to perform complex payload actions like encryption and decryption. To address this limitation, the OpenFlow standard has kept adding new actions. This is not a sustainable solution as packet processing actions can never be exhaustively standardized. Furthermore, frequent standardization of new action abstraction leads to frequent redevelopment of software/hardware, which results in vertically integrated hardware-software that must be replaced all together while performing network upgrades [7]. Third, the table abstraction in OpenFlow/P4 is fundamentally incapable of modeling the scheduling function. Scheduling is essential to implement Quality-of-Service (QoS), which is further essential for many middlebox functionalities. This inability is because the table abstraction performs splitting an aggregate flow into multiple flows whereas scheduling performs merging multiple flows into an aggregate flow. In some sense, a table and a scheduler perform opposite functionalities, and one cannot model the other.

Recently, modular abstraction approaches have been proposed to implement middlebox data plane [7], [12], [37], using insights from Click router [23] functionality. However, these approaches do not meet all the requirements of SDM data plane abstraction. Similar to Click router [23], E2 allows composing arbitrary packet processing functions, each configured as a data flow graph of processing elements [37]. Similarly, Slick [7], NetBricks [38], and OpenBox [12] allow implementing network functions as chains of low-level processing stages. Compared with match-action, modular abstraction is a more natural way of expressing SDM data plane. However, existing designs are not fully extensible in a platform-independent manner. They either rely on specific platforms, or do not consider how to extend data plane in a platform-independent manner at all.

In this paper, we propose OpenFunction, an extensible and platform-independent data plane abstraction protocol for software-defined middleboxes. This abstraction layer should be hardware optimizable, which means that the performance of an SDM box can be optimized by various hardware acceleration technologies. OpenFunction architecture consists of a logically centralized *OpenFunction controller* and a number of *OpenFunction boxes* distributed around a network. OpenFunction abstraction exposes an extensible set of elements to the control plane using the modular style. SDM programmers need not to be aware of the underlying hardware features of SDM boxes: just define the behaviour of data plane as a data flow graph of processing elements, and focus on the application logic of control plane. Under the hood, a OpenFunction defined data plane can be realized by a platform-dependent implementation that fully leverages the hardware features of the underlying SDM box. To support new operations, OpenFunction provides a platform-independent pseudo language for specifying customized elements beyond those predefined ones. Such a platform-independent pseudo program can be compiled to a platform-dependent element implementation by the underlying box.

The main challenge is how to abstract packet operations, flow states and event generations with elements. A data packet should be abstracted first before any access operation can be defined over it. There are three types of flow states: per-flow, multi-flow, or global [16], [42]. For example, an IDS's data plane might need to record packet counts of each flow, of flows to each destination, and of all the pass-through flows simultaneously. Some operations might depend on the flow states as input: for example, send an event to control plane when the number of packets to a destination exceeds a given threshold. Events can be used in various forms according to control plane's requirements. Apparently, explicitly defining different states/events for each application scenario separately would result in the exponential growth of the number of elements and unnecessary duplicate implementation.

The key design idea of OpenFunction is that actions/states/events should be defined in a uniform pattern and independent from each other. Accordingly, it separates three kinds of primitive data plane processing elements: *action*, *state* and *event*. An action element can modify packet data, but can not modify a state or issue an event to the control plane; a handful number of state elements are dedicated to updating the state records; several event elements are dedicated to issuing events. OpenFunction protocol treats every state as a record in a conceptually global key-value store. As a result, states can be accessed in a uniform way. OpenFunction treats the trigger condition of each event as a Boolean expression, and the event as a formatted string. As a result, event elements can be abstracted in the *evaluate-format* form.

OpenFunction data plane faces various other challenges. First, an SDM box may implement different kinds of network functions simultaneously. As a result, different elements may interact with each other to complete the service chain. Second, different elements may need to be loaded/unloaded dynamically to support different network functions' requirements. During load/unload operations, the OpenFunction box must not experience packet drops and hence degrade application performance. We propose a buffer-less in-box service chaining based solution to address this challenge. An OpenFunction box runs a parent program that creates the memory and defines the data structures for child processes. Each element is then started as a child process that can use the shared memory. Therefore, each element of a network function can be loaded/unloaded in a seamless fashion without hurting the application performance.

We implement an SDM system including one OpenFunction controller and three OpenFunction boxes based on Netmap [43], Data Plane Development Kit (DPDK) [21] and FPGA [27]. We also develop two stateless network functions (*i.e.*, NAT and IPsec), and two stateful network functions (stateful firewall and IPS). We draw the following

conclusions from our experimental results. First, middlebox functions implemented by using our OpenFunction abstraction can achieve high performance. For example, our FPGA platform can achieve near 10 Gbps throughput for NAT and IPsec middleboxes. Second, the performance of extended elements via OpenFunction pseudo programs are platform-dependent.

Rest of the paper is as follows: Related work is presented in § II. SDM architecture overview is presented in § III and the OpenFunction abstraction is in § IV. Control plane service chain scheduling algorithms is in Section V. Prototype system implementation is described in § VI. Evaluation results are demonstrated in § VII and we conclude in § VIII.

II. Background and Related Work

A. Network Function Virtualization

As an emerging direction of middleboxes, Network Function Virtualization (NFV) attempts to address the issues of tight hardware/software coupling and hardware vendor lock-in for middleboxes by implementing middlebox functions purely in software running on commodity servers [4]. Such a software based middlebox is called a Virtual Network Function (VNF). Its data plane can be realized by multiple VMs, each is called a VNF instance. Unfortunately, NFV middleboxes are software-based, not software-defined. NFV changes middleboxes from monolithic hardware/software blackboxes to monolithic software blackboxes. It does not provide a data plane abstraction, hence it is not a candidate architecture for the vision of software-defined middleboxes (SDM) [7], [37]. The key weakness of NFV middleboxes is low performance because commodity servers are designed for general computing purposes. The functions and requirements of packet processing are significantly different from those of general computing. Such differences warrant a wide variety of specialized hardware acceleration technologies such as ASIC chips and Network Processing Units (NPU), which typically have a 10-50 times performance improvement over commodity server based solutions [24]. This partially explains that market has been in favor of the closed-but-fast model, which uses proprietary hardware platform with fast packet processing capability, over the open-but-slow model, which uses commodity servers with slow packet processing capability.

OpenFunction advances the vision of NFV from two fronts. First, OpenFunction supports heterogenous hardware platforms (including hardware/software platforms such as network processors, multi-cores, ASICs, FPGAs, GPUs, and CPUs), whereas NFV only supports commodity servers. Commodity servers are designed for general computing purpose, not packet processing functions in middleboxes, which are quite unique in that they often have extremely high demand for time and memory efficiency. OpenFunction supports continuous innovation in both hardware and software of middleboxes. Second, OpenFunction abstraction significantly simplifies the programming of middlebox functions whereas in NFV every middlebox functionality has to be programmed from scratch as NFV does not have such an abstraction layer.

Some recent work focus on the smooth state transition among commodity servers for middlebox function scale-in and out. These techniques can be used by, and are complementary to, OpenFunction to support the extensible and manageable data plane. Split/Merge [42] groups middlebox states into *partitioned* and *coherent* categories and uses an application-level library to help middleboxes manage the state elastically. OpenNF [16] uses a combination of events and forwarding updates to address the race condition during the state migration phase. A software NFV framework CoMb [45] exploits consolidation opportunities in middlebox deployment to save cost. It also provides solutions to resource management, traffic redirection, and coordination among different data plane processing elements. CoMb limits the VNFs of a service chain in a single host. Another software NFV framework NetBricks builds a small set of customizable network processing elements and provides isolation [38]. Different from them, OpenFunction is fully platform-independent and can be implemented in both hardware (*e.g.*, FPGA) and software platforms.

With software-defined security (SDS) [6], [44], [46], information security is implemented, controlled and managed by security software. Being part of SDN, SDS is designed to be modular and scalable [44]. FRESCO is an application development framework designed to facilitate design, and modular composition of OpenFlow-enabled security detection and mitigation modules [46]. Based on the Mininet simulator, Darabseh *et al.* develop an experimental framework for SDS systems [50]. SDS could be seen as a kind of SDM. SDS could enjoy the benefits, such as platform independence, brought by OpenFunction.

B. Router Abstraction

Decomposing network protocols to components is important for performance and flexibility of implementation [34]. Some prior work focus on designing an abstraction layer to support portability across heterogeneous platforms. Handley *et al.* [19], [20] built extensible routers on top of commodity platforms called XORP, focusing on breaking the complex control plane of routing protocols. Mogul *et al.* [32] advocated an abstraction layer called Orphal to support portability of third-party software in a position paper. XORP and Orphal significantly differ from OpenFunction. First, XORP and Orphal are not designed for SDN whereas OpenFunction is. Second, XORP and Orphal are not platform-independent whereas OpenFunction is. Furthermore, XORP focuses on routing whereas OpenFunction focuses on middlebox functionalities. Orphal is based on router specific hardware resources (such as TCAM and DPI engines) whereas OpenFunction is based on middlebox functionalities and is hardware agnostic. Song [47] proposed a hardware abstraction called POF, which focuses on protocol-oblivious forwarding. It extends the OpenFlow instructions with protocol-oblivious operations such as *AddField* and *SetFieldFromValue*. Theoretically, a complete set of such low-level primitives can compose any data plane operation. However, using these primitives to construct middlebox functions is extremely difficult because they are too low-level. We implemented the POF abstraction and used these primitives to implement a NAT, and our experience well confirms the above insight. Anwer *et al.* [7]

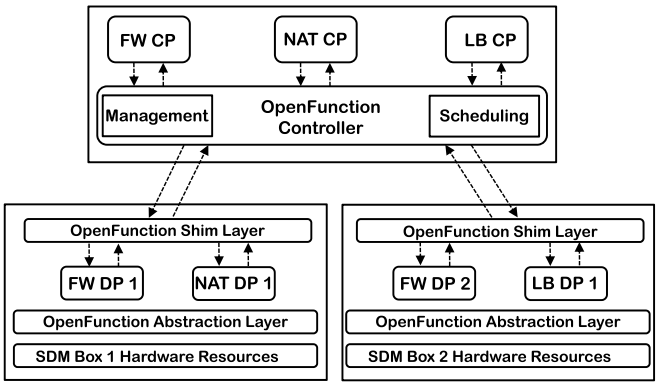


Fig. 1. Software-defined middlebox architecture.

proposed to decouple the processing at the controller and at middleboxes while providing an interface by which they can communicate. This vision is similar to ours, whereas the proposed architecture is not platform-independent [7].

C. Data Plane Programmability

Several systems focus on the data plane throughput of extensible software-based routers. These systems are orthogonal to OpenFunction and they are useful to build OpenFunction boxes. RouteBricks exploits parallelism both across multiple servers and across multiple cores inside each single server [13]. PacketShader exploits the massive parallelism of GPU for batch processing [18]. Programmable Protocol Processing Pipeline platform is based on FPGA [17]. GASPP focuses on stateful packet processing, and its GPU-accelerated framework can achieve multi-gigabit processing [48]. Some systems focus on data plane extensibility of new network forwarding protocols. The main direction is to pair the high-throughput ASIC processing path with a fully programmable co-processer to enhance the programmability. ServerSwitch uses x86 CPU as the co-processor [28], [29]. SSDP instead uses an NPU-driven subsystem [33]. SwitchBlade allows rapid prototyping in FPGA [8]. OpenDataPlane is an open-source cross-platform set of APIs for programming network processors [2]. OpenDataPlane abstraction is at a lower level than OpenFunction as OpenDataPlane specifies *how to do* by directly manipulating device resources whereas OpenFunction only specifies *what to do* and lets devices decide how to implement each action.

III. SDM Architecture Overview

OpenFunction-enabled SDM architecture consists of a logically centralized OpenFunction controller and a number of OpenFunction boxes distributed across the network. Every box implements the OpenFunction abstraction layer (Fig. 1).

Architecture: In OpenFunction, a network function (such as NAT, LB, or FW) is implemented by a Control Plane (CP) process and a set of Data Plane (DP) processes running on OpenFunction boxes. The CP and DP processes communicate with each other using the OpenFunction protocol. The CP process has a global view (via the controller) and its main role is to manage and deploy the DP processes according to required middlebox functionality. It receives events from DP processes, performs analysis, makes decisions, and sends commands to DP processes to enforce its decisions. A DP is modeled as a directed acyclic graph where each node is an element that implements middlebox functionality. A DP process has a local view of the network, it receives commands from CP and its main role is to process data plane packets accordingly. Based on the processed traffic, it may generate events for its corresponding CP process. We call a CP process together with its group of DP processes a *Software-Defined Middlebox* (SDM). With OpenFunction, implementing a middlebox functionality on the data plane becomes much simpler as it mostly involves composing a graph of pre-defined elements, possibly with a few user-defined elements. Thus, SDM developers mostly focus on designing CP programs, which are often the most innovative part of their implementation.

OpenFunction controller manages DP elements and makes scheduling decisions to install/remove elements and packet handling decisions. OpenFunction controller sends flow steering instructions to the OpenFunction shim layer running on SDM boxes, which maintains network state information and deploys network elements accordingly.

Data Plane: OpenFunction data plane abstraction is element oriented, similar to Click. An element is a self-contained and functionally independent packet processing unit, such as decreasing the TTL field, calculating the TCP checksum, or increasing a flow state counter. OpenFunction exposes an extensible set of elements to the control plane. The semantics of an element is to take a packet as its input, perform some operations, and either push the packet to the next element or wait for the next element to pull the packet. Each element object is an instance of an element class. There are three element classes: actions, states and events. For example, an element may need to update the flow counter state whenever a packet passes through this element. Another element may send an event to the control plane. There are two kinds of elements: pre-defined and user-defined. Pre-defined elements are those supported by OpenFunction compliant boxes and user-defined elements are those written by users using a platform-independent pseudo language. We allow multiple DP processes running on the same OpenFunction box, similar to router virtualization, where in the same box, different DP processes belongs to different CP processes for different middlebox functions (§IV).

Control Plane: OpenFunction, together with OpenFlow, makes service chain construction and scheduling much easier than the current NFV architecture. A service chain is a sequence of middlebox functions that the network operator wants certain traffic to traverse in order. To construct and schedule service chains, the OpenFunction controller performs resource allocation according to a given optimization policy (*e.g.*, load balancing) in collaboration with the forwarding plane authority (*e.g.*, an OpenFlow controller). For example, suppose the network operator wants to allocate 10 Gbps bandwidth between two networks and let all traffic between the two networks pass through an IDS. First, the OpenFlow controller finds an appropriate path between the two networks

where each switch on the path has at least 10 Gbps bandwidth and the path contains at least one OpenFunction box. Second, the OpenFunction controller starts a new IDS DP process on that OpenFunction box. In this SDM+SDN and OpenFunction+OpenFlow paradigm, one technical challenge is service chain scheduling: given a service chain that a flow needs to traverse, the controller chooses a sequence of middlebox data plane processes (running on one or more OpenFunction boxes) to form the service chain. The scheduling needs to satisfy three requirements: service chain constraints, device resource constraints, and flow conservation. Furthermore, it needs to be done in a load balancing way to minimize the maximum utilization of the resources in candidate devices. To address this challenge, in this paper, we formulate service chain scheduling as a NP-Hard problem, and propose both offline and online scheduling algorithms (§V).

Benefits: OpenFunction realizes the centralization vision of SDN for middleboxes by the logically centralized OpenFunction controller. As it is well understood that centralization in OpenFlow brings a long list of benefits to packet forwarding such as flexible path selection, better load balancing, finer-grained network control, less configuration errors, higher manageability, and increased reliability and security, centralization in OpenFunction brings many benefits to network functions. The global view that a CP process obtains from the events generated by its DP processes helps the CP to make better informed decisions. For example, an IDS CP process can perform correlation analysis on the events from its DP processes to better identify attack activities.

IV. OpenFunction Abstraction

In this section, we present OpenFunction abstraction. We first introduce how a data flow graph is implemented, followed by packet abstraction and pseudo language to implement elements in platform-independent manner. We then provide a deep down on the element abstraction at action, event and state level.

A key design aspect of OpenFunction abstraction is flexibility and extensibility. As mentioned above, OpenFlow is vertically integrated since it continuously adds new actions. A program calls new actions cannot run on a device supporting only older versions. OpenFunction does not enforce any standard element definition to prevent the exact fate. Instead, a set of elements are *recommended*; and for each such an element, a default implementation written in OpenFunction pseudo language (§IV-B) is provided. An OpenFunction box can choose elements to be optimized by exploiting their underlying hardware acceleration capabilities. For elements whose optimized implementation is not available at the box, the default pseudo code can be used, like user-defined elements.

Furthermore, we allow OpenFunction boxes to provide multiple implementations for the same element: for example, one version is CPU based and another version is GPU based. The controller can dynamically determine which version to load/unload based on the availability of network resources and the processing requirements of flows at run time.

```
1: @ FromDevice(1) fromdevice
2: @ ExactMatch(PROTO_IP,-) match
3: @ DecapHeader(14) decap
4: @ ESPEncap espencap
5: @ Aes(EBC) aes
6: @ IPsecEncap ipencap
7: @ ChangeSrcIP(IP_SRC) srcip
8: @ ChangeDstIP(IP_DST) dstip
9: @ SetIPChecksum checksum
10: @ EncapHeader(MAC_DEST, MAC_SRC, PROTO_IP) encap
11: @ ToDevice(2) todevice

12: fromdevice 0 0 match; match 0 0 decap
13: decap 0 0 espencap; espencap 0 0 aes
14: aes 0 0 ipencap; ipencap 0 0 srcip
15: srcip 0 0 dstip; dstip 0 0 checksum
16: checksum 0 0 encap; encap 0 0 todevice
17: match 1 0 discard;
```

Fig. 2. Pseudo code of IPsec Script (one way).

A. Data Flow Graph

OpenFunction datapath is modeled as a directed acyclic graph where each node is an element that implements one middlebox functionality. OpenFunction data plane specification can be implemented using a script, which contains the definition of each element and the definition of connection between elements. An example IPsec middlebox data plane is shown in Fig. 2. In OpenFunction script, each element instance definition line starts with an "@". After that, each line defines a connection between an egress port of one element and an ingress port of another element. Element definitions and connection definitions together construct the graph of a DP process.

As the target of element operations, a packet class consists of two types of data: fields, and properties. A property is a non-modifiable information about the packet, such as the packet size or the physical port from which the packet is received. Although some properties of a packet may change after certain actions, *e.g.*, the length of a packet changes after encapsulation, the properties of a packet cannot be written or modified by elements directly. Note that we deliberately do not include metadata in the packet abstraction. Metadata are short-lived attribute values attached with each packet, usually for the purpose of passing parameters among processing stages. Since metadata information are extracted from packet fields or properties, it can be regarded as implementation optimization, instead of a native packet abstraction.

B. Pseudo Language

We propose a platform-independent pseudo language that allows SDM developers to design arbitrary new middlebox data plane element, and should be supported by any OpenFunction enabled device. The vendor of an OpenFunction box needs to provide a source-to-source translator, which takes a pseudo C program as its input and outputs a native program that exploits the hardware acceleration strength of the box.

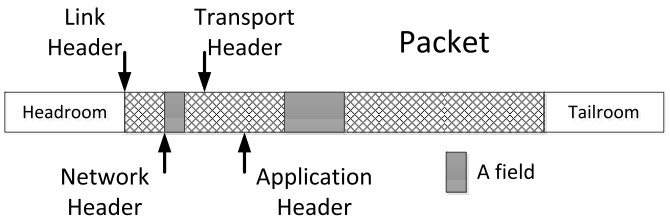


Fig. 3. Abstraction of packet field.

The output platform-dependent program is further compiled into executable components using the box's native compiler. As a proof-of-concept, shown in Section VI, in our prototype system an element class can be translated either to a platform-dependent C/C++ program such as a Click element, or a FPGA block (in Verilog).

We now introduce the packet field abstraction that the pseudo programs operate on. The whole packet is assumed to be stored in a continuous memory location as shown in Figure 3. There is large headroom/tailroom before/after the packet data. Such spaces are reserved for actions that change packet length (*i.e.*, Encap/Decap and Pad/Unpad). We also assume the existence of a mandatory preprocessing step, which marks the start offset of link/network/transport/app layers (if they exist). Each field is a continuous range of bytes in a packet memory range, defined by a tuple `(header, offset, type)` and can be attached with a name. For example, IP checksum can be defined as a field starts at the 10th bytes offset of the network header by the `FIELD` keyword. The length is automatically determined by its data type `UNIT16`. Note that a special type is `DATA`, which represents a raw array of bytes.

A property of a packet is accessed as a named object (with keyword `PROP`). In OpenFunction, a `(type, length)` tuple defines a specific object. The `length` configuration is required only when the `type` is `DATA`. Note that packet properties are limited to a small set of intrinsic packet characteristics and read-only to program. Element parameters (with keyword `PARA`) are also accessed as named objects. The additional feature of a parameter is that it can be read/written by the control plane.

A key design decision is to separate packet access code and normal code. More specifically, we use character "@" at the start of a line to denote a data access line. Following the "@" letter, there should either be a packet variable definition of `FIELD`, `PROP` or `PARA`, or `LOAD`/`STORE` to access the data. Currently, we do not support direct algorithmic operations over packet data variables. They are used to denote the physical locations in the packet. Instead, we first load them into local variables, perform calculation, and then store them back. The benefit is that all non-data-access codes can be written as normal C codes.

Fig. 4 shows an example pseudo program. This user-defined *SetIPChecksum* action accepts an incoming IP packet and calculates and sets its IP checksum field. Here the action assumes that the incoming packets are with MAC headers. As an example, the *iphlen* is defined first (Line 1), and loaded into a local variable *hlen* before access (Line 5).

```
1:  @ FIELD iphlen NETWORK 0 UINT8
2:  @ FIELD ipcsum NETWORK 10 UINT8
3:  @ FIELD ipheader NETWORK 0 DATA
4:  unsigned char hlen
5:  @ LOAD iphlen hlen
6:  hlen = hlen & 0x0f
7:  hlen = hlen ≪ 2
8:  @ STORE ipcsum 0
9:  unsigned int sum=0
10: unsigned short hoffset=0
11: unsigned short tempshort=0
12: while hoffset < hlen do {
13:     @ LOAD ipheader tempshort hoffset
    UINT16
14:     hoffset = hoffset + 2
15:     sum = sum + tempshort
16: }
17: unsigned int tempint = 0
18: tempint = sum & 0xffff0000
19: while tempint != 0 do {
20:     sum = sum & 0xffff
21:     tempint = tempint ≫ 16
22:     sum = sum + tempint
23:     tempint = sum & 0xffff0000
24: }
25: sum = ∼sum
26: @ STORE ipcsum sum
```

Fig. 4. SetIPChecksum (User Defined).

C. Action Elements

To carefully tradeoff composability and optimizability, we analyze a number of middleboxes such as NAT, IPsec, and IDS, and design a suite of primitive actions that we classify into the following five categories:

- *starting action:* These actions receive packets from either hardware/virtualized NICs, or specific memory locations (*e.g.*, FromDevice in the IPsec example).
- *one-to-many action:* This category consists of actions based on the following four matches: exact-match (*e.g.*, in the IPsec example), longest-prefix-match (*e.g.*, used in FIB), first-match (*e.g.*, used in ACL), and pattern-match (*e.g.*, regular expression for payload used in DPI).
- *many-to-one action:* A flow scheduling action takes multiple flows as input, and outputs one combined stream of traffic. Example flow scheduling operations include strict priority (SP) and weighted round robin (WRR). Traffic shaper can be regarded as a special case of this category.
- *one-to-one action:* An action performs a single packet processing such as changing the length of a packet, either at the head (*i.e.*, Encap/Decap) or the tail (*i.e.*, Pad/Unpad), modifying a field or the metadata of a packet (*i.e.*, decreasing the IP TTL value and setting the ECN bit are two actions), and decompressing/compressing packets (*i.e.*, stateful Web cache).

```
1: @ FIELD ect NETWORK 1 UINT8
2: @ STR str "ECN Enabled "
3: @ COND hlen & 0x02
4: @ FIRE str ect
```

Fig. 5. ECN Event Parameter.

- *ending action:* These actions send packets to either hardware/virtualized NICs, or specific memory locations (*e.g.*, ToDevice in the IPsec example).

The parameters of an action can be set either at the configuration phase, or at run-time. Considering the IPsec example again. For a given IPsec tunnel, the IP_SRC/IP_DST parameters of ChangeSrcIP/ChangeDstIP actions are set at the configuration phase as parameters. While for the ESPEncap action, its security index SPI and replay parameter RPL should be negotiated by the control plane at run-time. The IPsec DP process leaves the two parameters unset at first, and lets the CP process performs negotiation before setting the attribute value.

D. State Elements

Many middlebox states can only be maintained in the data plane. OpenFunction treats every data plane state as a record in a conceptually global key-value store. Currently, only flows' five tuples (*i.e.*, source/dest IP, source/dest port, and protocol) are supported as key in OpenFunction. Wildcards are used to represent the keys of multi-flow and global states. We believe it is easy to extend the design by using a data structure to support more data as key.

In OpenFunction, state elements are dedicated to updating the state records: *e.g.*, IncreaseStateNumerical and UpdateStateOctet. Action and event elements can read states by directly accessing the global key-value store. The reason is that state value can be updated in many ways. Sometimes the new value is not directly related to the old value: for example, a state of the TCP ack-window sequence number. Sometimes the new numerical value is related to the old value: for example, per-flow packet counter. Also, value can have many data types: *e.g.*, integer and octet.

E. Event Elements

The event abstraction uses a single *evaluate-format* element. As a configuration parameter, each event instance specifies the trigger condition by a matching rule, and an octet array that is the content of the event.

Fig. 5 shows an example parameter string passed to an event element instance. In this setting, if the transport protocol sets the ECN enable bit, an event is sent to the control plane. The information used are defined in the first part (line 1 and 2); the matching rule is in line 3; the event formation is defined in line 4.

F. State Management

We classify the states inside any middlebox appliance as internal and external [42]. An internal state is transient and can be created, modified and destroyed during the course of processing a packet. External states are further classified into *coherent* and *partitioned* states. An coherent state, consists of static configuration information, is maintained by the global controller. A partitioned state represents the flow-specific state maintained by each middlebox appliance. In our current design, partitioned states are maintained in the data plane, by each action element.

We manage the flow-specific states inside each action element *explicitly*. By explicit state management, we mean that each action element has to explicitly acquire and release any flow-specific state that it maintains, before and after processing a packet respectively. Explicit state management has been shown to enable the design of robust and scalable middlebox applications [40]–[42].

We create two generic interfaces, namely *read_flow* and *write_flow*, that each action needs to implement. *read_flow* needs to read the flow-specific state from the middlebox device and pass it on as input, along with the packet to the action element. Similarly, *write_flow* needs to write the flow-specific state to the middlebox device, for future retrieval, followed by passing the packet to the subsequent action in the processing graph. In DPDK-based appliances, read and write flow implementations translate into reading and writing to the action's internal flow table entries. In hardware-based middleboxes, these implementations may be optimized to lookup from internal TCAM tables or other storage media. Each action's *read_flow* and *write_flow* are represented as action elements themselves, in the processing graph that represents the middlebox application.

V. Service Chain Scheduling

One challenging task for OpenFunction controllers is to perform service chain scheduling. Given a flow and a sequence of SDM functions that the flow needs to pass through, the controller needs to choose a sequence of DP processes of that SDM so that the flow can go through this sequence of DP processes. Note that, if a specific action in the program has multiple implementations (*e.g.*, both CPU-based and GPU-based), then the compiled SDM data plane also has multiple implementations. The controller can load any desired implementation according to the resource scheduling and flow requirements. For example, if a middlebox's data plane graph has 3 actions and each action has 2 implementations. In total there could be $2^3 = 8$ implementations stored for this middlebox. Due to the cheap price of storage, we consider this exponential increase in the number of implementations a trivial concern.

Some existing schemes in service chain scheduling and enforcement can be used in OpenFunction to control the service chain among middlebox platforms [14], [39] with intelligent optimizations. CoMb requires that the middlebox processes of the whole service chain, pertaining to a given session flow, run on the same device [45]. However, this is not optimal in many OpenFunction scenarios. For example, imagine that one hardware platform has special optimization for IPsec, and another platform has special optimization for

TABLE I
SYMBOLS USED IN THIS PAPER

k, K	SDM k, $1 \leq k \leq K$.
n, N	OpenFunction box n, $1 \leq n \leq N$.
r, R	resource r, $1 \leq r \leq R$.
N_{kl}^{f}	Input binary constant of flow f's service chain to indicate if SDM l is the next SDM of k
$D_{k,i,r}$	The amount of resource r is required to support a unit of flow for SDM k with implementation i.
$\Gamma_{n,r}$	the existing utilization of resource r in box n.
$\Omega_{n,r}$	the capabilities of resource r in box n.
A_f	The amount of flow f
$Z_{k,i,n}^{l,j,m,f}$	Binary variable to indicate if flow f is traversed from the SDM k with implementation i on box n to SDM l with implementation j on box m.
$X_{k,i,n}$	The amount of flows traversing node n for SDM k with implementation i.

cache. Forcing the whole service chain in a single node compromises the benefits of global resource optimization. Some schemes have been proposed to solve the problem of steering traffic among physical middleboxes to follow the network policy (*i.e.*, service chain). As a pioneer work, the pswitches (*i.e.*, policy-aware switches) scheme adds a new layer-2 for data centers, which can steer the network traffic through unmodified middleboxes [22]. Using SDN, SIMPLE can enforce flow-level policy even if middleboxes modify the packets or change the session level semantics [39]. FlowTag takes a step further by attaching tags to flow packets to enable flow tracking [14].

The scheduling in this paper focuses on load balancing across different physical nodes. For isolation purpose, many researchers focus on the VM based VNF placement scheme. E2 [37] realizes each VNF as a VM and consolidates the VMs for a chain to a server, and hence reduces the inter-server traffic. Stratos [15] is another traffic-aware NFV placement which is an orchestration layer mainly to deal with the mangling NFs. VNP-OP [9] studies joint optimization of the VM placement and traffic routing to minimize the deployment cost and forwarding cost. PACE [25] also focuses on the VNF placement to accommodate as many requests as possible, but it assumes that all NF requirements in a chain are unordered.

Next, we present the service chain scheduling algorithm. Table I summarizes the symbols used in this section. For a given flow, the OpenFunction controller needs the following three types of inputs to perform service chain scheduling.

- *Chaining Requirement:* Each flow has a classification rule that uniquely identifies the flow and estimated bandwidth. For the given flow, we use $SDM_1, SDM_2, \cdots, SDM_K$ to denote the sequence of K SDMs that it needs to traverse. This flow specification is typically specified by network operators.
- *Implementation Availability:* For each SDM_k, let V_k denotes the total number of different implementations of SDM_k. Each implementation has its specific resource requirement, which is usually multi-dimensional because different actions in an implementation may require different resources.
- *Box Capability:* Let N denote the set of OpenFunction boxes. Let $\Omega_{n,r}$ denote the capabilities of resource r in each box n. Also, the existing utilization of resource r in OpenFunction box n is denoted as $\Gamma_{n,r}$ (in percentage). The OpenFunction controller obtains box capabilities by performing resource monitoring of each OpenFunction box in real time.

As a practical constraint, we assume, all packets in a given flow should traverse the same DP process for a given SDM in the chain.

Below, we formulate the resource allocation problem as an integer linear program and propose an offline and online heuristic to solve ILP problem.

ILP Formulation: The optimization objective in service chain scheduling is load balancing across all resources of all nodes.

$$\text{minimize } \max \frac{\sum_k \sum_i X_{k,i,n} D_{k,i,r}}{\Omega_{n,r}} + \Gamma_{n,r} \tag{1}$$

The main constraint is traffic conservation for each flow crossing the SDMs

$$\sum_{l,j,m \geq n} Z_{k,i,n}^{l,j,m,f} - \sum_{l,j,m \leq n} Z_{l,j,m}^{k,i,n,f} = \begin{cases} 1 & \text{if } k = 0, n = 0 \\ -1 & \text{if } k = K, n = N \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

In addition, a flow may traverse from SDM k to SDM l only if SDM k is followed by SDM l on flow f's service chain,

$$N_{kl}^{f} \geq Z_{k,i,n}^{l,j,m,f} \tag{3}$$

Based on this flow conservation constraint, the traffic can be calculated by

$$X_{k,i,n} = \sum_f \sum_{l,j,m} A_f Z_{l,j,m}^{k,i,n,f} \tag{4}$$

To derive an ILP formulation, we can introduce an auxiliary variable z to present the objective function and z should satisfy

$$z \geq \frac{\sum_k \sum_i X_{k,i,n} D_{k,i,r}}{\Omega_{n,r}} + \Gamma_{n,r} \tag{5}$$

and the objective becomes

$$\text{minimize } z \tag{6}$$

Therefore, the Integer Linear Programming (ILP) formulation of this problem can be summarized as the resource utilization optimization problem (RUOP). The ILP model formulated above is intractable in large networks. Specifically, due to the large dimension of binary variable $Z_{k,i,n}^{l,j,m,f}$, there will be more than 10^7 binary variables in the model even if each index has only 10 possible values. Accordingly, we need to design an efficient heuristic to solve this problem.

Offline Algorithm: We first consider the offline scheduling, where a number of flows are known and resource allocation can be optimized simultaneously. Since the most critical element which makes our problem intractable is the large number of binary variables, we consider relaxation and rounding method to solve this problem. For presentation clarity, we call the enforced service chain (*i.e.*, the ordered SDM) as the path of this flow. Following the relaxation and rounding idea,

Algorithm 1 Offline Resource Utilization Optimization

Require: Flow volume and service chain requirement, resource requirement for each SDM implementation
Ensure: The SDM path of each flow
1: Formulate RUOP model according to the input
2: **while** not all $Z_{k,i,n}^{l,j,m,f} = 0$or1 **do**
3: Solve RUOP model
4: Get a forbidden threshold ϵ
5: **for** all $Z_{k,i,n}^{l,j,m,f}$ **do**
6: **if** $Z_{k,i,n}^{l,j,m,f} < \epsilon$ **then**
7: Add one more constraint $Z_{k,i,n}^{l,j,m,f} = 0$ into RUOP model
8: **end if**
9: **end for**
10: **end while**
11: **return** $\{x_i^k\}$

we first relax the binary constraint of the variable $Z_{k,i,n}^{l,j,m,f}$, *i.e.*, treat it as a real variable in the range $[0, 1]$, and solve the derived Linear Programming model. With this solution, each flow may be split among multiple paths which are determined by the value of $Z_{k,i,n}^{l,j,m,f}$. To get an unsplittable path for each flow, a simple method is to directly round each flow to the path which carries the largest fraction of this flow. However, this method can not obtain a good result especially when there are a lot of boxes and many implementation versions for each SDM. In this case, each path may carry only a small fraction of the flow. Directly rounding each flow to the path carrying the maximum fraction may deviate far away from the ILP optimum. To solve this problem, we propose a progressive method for the rounding phase to get unsplittable paths.

The main idea of progressive rounding is to iteratively forbid some links that carrying very small fraction for each flow, till there is only one path for each flow or there is a path carrying most of the fraction of a flow. Based on above discussions, we design Algorithm 1 to solve the RUOP formulation.

Line 4, requires finding a threshold ϵ to determine if a $Z_{k,i,n}^{l,j,m,f}$ should be set to 0 for the rounding purpose. On the one hand, ϵ defines the tradeoff between the algorithm performance and the algorithm running time. With a large ϵ, more $Z_{k,i,n}^{l,j,m,f}$ will be set to 0 in each loop and hence there are fewer loops executed in the algorithm. With a small ϵ, we can derive a better solution. On the other hand, we should update ϵ in each loop since $Z_{k,i,n}^{l,j,m,f}$ should become larger in each loop as there are fewer paths can be used by the flows with the algorithm execution. Accordingly, without updating the rounding threshold, there will be endless loop in the algorithm.

From Offline to Online: We extend the algorithm to an online version of this problem. In real networks, service chain requests may arrive randomly, and not all at once. Our approach to solve this problem is to treat all new flows in a fixed interval (*i.e.*, 1 us) as a group. OpenFunction then triggers the offline algorithm to reserve SDM resources for this group of flows. This batch calculation is an optimization to save computation. This implementation-specific optimization can be customized and disabled. If the flow arrival rate is very small, holding flows to wait for later flows may be a large overhead for flows arrived earlier. In the worst case, we can treat each flow as a group to trigger Algorithm 1 to set up SDMs for each single flow.

VI. Prototype Implementation

A. Platform

We build a proof-of-concept system to verify the OpenFunction abstraction and estimate achievable performance.

Data Plane: We develop two x64 platform OpenFunction boxes based on Netmap [43] and DPDK [21], and one hardware box based on FPGA [27]. Netmap is a tool designed for high speed packet I/O in commodity servers. Implemented as either a modified NIC driver or a kernel module, Netmap also enables high-speed processing in user space. Supported by Intel, Data Plane Development Kit (DPDK) is a combination of data plane libraries and NIC drivers for fast packet processing in user space. Poll mode drivers (PMD) are designed to enable direct packets exchange between a user space process and a NIC. A low overhead *run-to-completion* model is used to achieve fast data plane performance.

In our testbed, there are 16 Dell R320 machines, each has a 10 Gbps Intel X520-SR1 ethernet NIC and is both DPDK and Netmap enabled. We use Redis distributed key-value store for SDM states [49]. We also develop a prototype box in a FPGA card via Xilinx Vivado High-Level Synthesis tool, which supports C/C++ to describe designs and translates them into hardware description languages (HDL) such as VHDL or Verilog HDL [31]. The platform is a programmable FPGA card with a Xilinx Kintex 7 chip, two 10 Gbps NICs and four 1 Gbps NICs.

For each prototype box, OpenFunction needs to provide three supports: OpenFunction script, OpenFunction language and *in-box chaining*. The third requirement means that the device needs to load/unload DP functions dynamically. For example, consider a scenario where initially only flow 1 passes through SDM 1 and SDM 2. Later the operator decides to add an additional action to be performed on that flow, which requires SDM 3 be inserted into the service chain. An example use case of this scenario is that a company may want to perform DPI to block instant message packets (such as GTalk and Skype) only during the office hours. Such temporary insertion/deletion of data plane instances should be seamless as packet losses during transition can affect application performance. A simple solution could be temporarily hold the packets of the flow, construct a new service chain, and switch to the new chain. However, this approach adds significant delay to packets processing during the transition. Instead, we prefer a buffer-less solution (§VI-D), where a function can be inserted/deleted without incurring a packet loss.

Control Plane: We use OpenDaylight (ODL) [36] as the OpenFlow controller that manages OpenFlow capable switches and routers. OpenFunction controller runs as an OpenDaylight application and sends flow steering instructions to the Openflow controller. It uses many base functions provided by ODL to learn about the network and control network elements.

```
1: in1 :: FromDevice(netmap:eth1, PROMISC true)
2: out2 :: Queue(100)→ToDevice(netmap:eth2)
3: elementclass ExactMatch Classifier
4: EM::ExactMatch(12/0800,-)
5: in1→EM→DecapHeader(14)→MyIPsecESPEncap
   →MyAes(1)→IPsecEncap→SetIPChecksum
   →MarkIPHeader(OFFSET 0)→ChangeSrcIP
   →ChangeDstIP→EncapHeader→out2
6: EM[1]->Discard
```

Fig. 6. Translated Click IPsec script.

In our testbed, we use OpenvSwitch [3] as a switching element to steer traffic. We use a quad core 3.0 GHz AMD Phenom(tm) II X4 945 Processor system to host the OpenFunction controller, the OpenFlow controller and all SDM CP processes.

B. Script Support

In this section, we discuss how OpenFunction uses script support for different platforms. For brevity, we only discuss Netmap and FPGA platforms.

Netmap tool focuses on high-speed IO and does not provide enough native packet libraries. For Netmap boxes, we use Netmap to perform IO and use Click as the packet processing pipeline library. We translate Netmap parameters to Click element style. The translated Click script of the IPsec data plane specification (Fig. 2) is shown in Fig. 6. This translation faces multiple challenges. First, how to handle the abstract parameters of actions in the specification? To address this challenge, we either replace them with real parameters in Click format, or implement them as action attributes, and let the IPsec CP process sets these parameters at runtime. Second, what other elements need to be defined? To address this challenge, we add additional elements. For example, we add additional *MarkMACHeader* and *MarkIPHeader* elements (Fig. 6), after the input device action, for Click specific requirements.

For FPGA box, both OpenFunction script and pseudo language are translated to the special C/C++ code: the specification is translated to the top level block (composed by a call graph among low level blocks), while each pseudo language based action definition is translated to a low level block. The translation of data plane IPsec specification (Fig. 2) as shown in Fig. 7, requires (1) defining the inter-link between functional actions (*i.e.*, Line 5-20); and (2) using direct function calls to represent actions (*e.g.*, Line 21-29). Note that in FPGA platform, we implement all parameters as action attributes, which are controlled by its corresponding CP process.

```
1: void ipsec(stream<axiWord> &inData,
   stream<axiWord> &outData) {
2: #pragma HLS dataflow interval=1
3: #pragma HLS INTERFACE port=inData axis
4: #pragma HLS INTERFACE port=outData axis
5: static stream<axiWord> ippacket("ippacket");
6: static stream<axiWord>
   decap2espencap("decap2espencap");
7: static stream<axiWord> espencap2aes("espencap2aes");
8: static stream<axiWord> aes2ipencap("aes2ipencap");
9: static stream<axiWord>
   ipencap2changeSrcIP("ipencap2changeSrcIP");
10: static stream<axiWord>
   changeSrcIP2changeDstIP("changeSrcIP2changeDstIP");
11: static stream<axiWord>
   changeDstIP2ipchecksum("changeDstIP2ipchecksum");
12: static stream<axiWord>
   ipchecksum2macencap("ipchecksum2macencap");
13: #pragma HLS STREAM variable=ippacket depth=16
14: #pragma HLS STREAM variable=decap2espencap
   depth=16
15: #pragma HLS STREAM variable=espencap2aes depth=16
16: #pragma HLS STREAM variable=aes2ipencap depth=16
17: #pragma HLS STREAM variable=ipencap2changeSrcIP
   depth=16
18: #pragma HLS STREAM
   variable=changeSrcIP2changeDstIP depth=16
19: #pragma HLS STREAM
   variable=changeDstIP2ipchecksum depth=16
20: #pragma HLS STREAM
   variable=ipchecksum2macencap depth=16
21: parser(inData, ippacket);
22: decap(ippacket, decap2espencap);
23: espencap(decap2espencap, espencap2aes);
24: aes(espencap2aes,aes2ipencap);
25: ipencap(aes2ipencap, ipencap2encap);
26: encap(decap2encap, encap2changeSrcIP);
27: changeSrcIP(encap2changeSrcIP,
   changeSrcIP2changeDstIP);
28: changeDstIP(changeSrcIP2changeDstIP,
   changeDstIP2ipchecksum);
29: ipchecksum(changeDstIP2ipchecksum, outData); }
```

Fig. 7. Translated Xilinx specification.

C. Language Support

In Netmap platform, we translate each action pseudo program to an element class definition in Click. Mostly, the generated codes are in the $simple\_action$ function. Each special line in the pseudo language file is processed. For example, Fig. 8 is the translated code of generating a pointer to IP

```
1: nh_data = p→network_header();
2: unsigned char* pch = NULL
3: pch = nh_data
4: pch = pch + 0
5: unsigned char* iphlen = pch
```

Fig. 8. Translated Click source.

header length field (Fig. 4 Line 1). Other codes are input from the pseudo program as it is.

OpenFunction language translation in Xilinx FPGA is similar to the operations of that in Netmap. The major difference

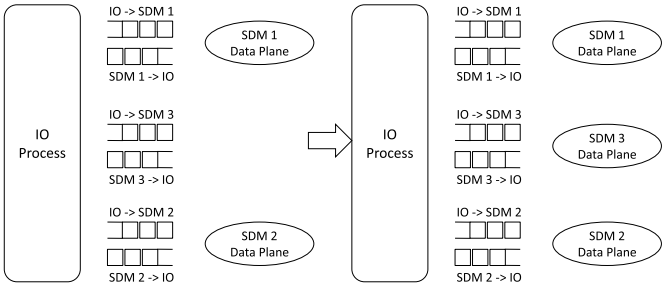


Fig. 9. IO based service chain.

is that: an additional C++ encapsulation class is required for each action class, to express the state transition semantics in FPGA hardware. Due to space limitation, we omit the details of both FPGA and DPDK platforms.

D. In-Box Service Chain

An OpenFunction box may implement more than one kind of network functions therefore, an SDM box may require different elements to interact with each other, within the same box, to implement a network functionality. One of the key requirement of in-box service chaining is that it should support loading/unloading elements dynamically. Below we discuss how we implement in-box service chaining for different platforms. For brevity, we only explain DPDK and FPGA platforms.

In DPDK equipped boxes, we use process group to realize the service chain and treat each SDM data plane instance as a separate process. In this model a central (parent) process sets up all memory structures for use by child processes (elements) and handles packet reception and transmission, as shown in Figure 9. A pair of rings is setup between the IO process and an SDM process for packets exchange. When a new SDM DP process starts, it discovers both the rings and packet memory locations via DPDK library support. The IO process sends a packet's descriptor through the ring. An SDM data plane processes the packet and returns it via another ring.

The pair of rings is the key to realize dynamic load/unload of the service chain elements. Each packet's metadata carries the service chain information. Metadata carries two entries: first, an ordered array records the indexes of ring pairs it needs to traverse, and second, an index pointer keeps the value of the next pair. Let's consider a scenario where initially flow 1 passes only through SDM 1 and SDM 2. Later the operator decides to add an additional action to be performed on that flow, which requires SDM 3 to be inserted into the service chain between SDM 1 and SDM 2. Initially only SDM 1 and SDM 2 are traversed by a flow. Later when the new SDM 3 data plane process starts, all subsequent packets' metadata change to the new service chain $SDM\ 1 \rightarrow SDM\ 3 \rightarrow SDM\ 2$; and as a result, packets traverse SDM 3 between SDM 1 and SDM 2. Similarly, to unload a service chain, when SDM 3 is removed from the service chain, the metadata of all subsequent packets change to the new service chain $SDM\ 1 \rightarrow SDM\ 2$ again; the controller waits for a while, before it cleans up the SDM 3 process from the system. For performance consideration, we assume all processes of a group is running on a single CPU socket (possibly with multiple cores). Note that multiple process group are supported, if we start different process groups in different sockets.

For FPGA equipped SDM boxes, in-box service chaining can be achieved using hardware support from FPGA. Unlike software, loading/unloading new data plane functions in programmable hardware needs direct support from hardware. Partial Reconfiguration is a feature of modern FPGAs, which allows a subset of the logic fabric of a FPGA to be dynamically reconfigured while the remaining logic continues to operate [10].

E. Developed SDMs

We implement NAT and IPsec gateway as two stateless SDMs on Netmap, DPDK and FPGA platforms. The NAT middlebox performs simple address translation. The mapping table is stored in its CP process and the mapping item is send to a DP process when the first packet of a flow comes in. The IPsec gateway uses ESP tunnel model with AES-EBC encryption. The control plane negotiates with the remote peer, and sends command to DP processes after security negotiation.

With Netmap, we also implement a simple stateful firewall and a stateful IPS. The stateful firewall monitors all pass-through TCP connections. It removes the `allow` rules in both directions whenever a RESET flag is detected. The stateful IPS monitors FTP connections: some FTP commands are allowed only after the user authentication; otherwise an alert signal is generated.

VII. Evaluation

In this section, we evaluate OpenFunction DP and CP performance. For DP, we are interested in the overall throughput achieved by different SDM implementations. For CP, we are interested in controller performance to see, how fast a OpenFunction CP adapts to dynamic network conditions and how quickly it can add/remove new network functions.

A. Data Plane Performance

First, we test the OpenFunction specification performance for the data plane abstraction of each SDM. For each SDM, we use the same set of OpenFunction specifications. We test with both small size packets (60 bytes) and large size packets (1400 bytes). The results of NetBricks on the same hardware platform are also added for comparison. Table II shows the throughput performance.

For non-stateful SDMs (NAT and IPsec), DPDK box achieves higher throughput than Netmap-based box in almost all the scenarios. This demonstrates that, even with the same OpenFunction abstraction implementation and same hardware, the performance actually relies on underlying systems. In turn, the FPGA box is always better than DPDK.

Key Takeaway 1: OpenFunction abstraction accommodates vendor differentiation with the same code across heterogeneous systems, which is beneficial for the NFV community.

TABLE II
THROUGHPUT OF OPENFUNCTION MIDDLEBOXES (G BIT-PER-SECOND)

	Netmap	DPDK	FPGA	FPGA-C	NetBricks
NAT Small	0.14	1.84	1.64	0.24	5.19
NAT Large	0.20	9.96	9.34	0.53	9.99
IPsec Small	0.02	1.2	2.61	0.04	N/A
IPsec Large	0.17	2.8	9.35	0.10	N/A
FW Small	0.03	N/A	N/A	N/A	0.32
FW Large	0.18	N/A	N/A	N/A	5.63
IPS Small	0.02	N/A	N/A	N/A	N/A
IPS Large	0.19	N/A	N/A	N/A	N/A

Next, we replace the *SetIPChecksum* and *SetTCPChecksum* actions in the specification with user defined versions (written in OpenFunction language and translated). The performance overhead caused by user defined versions is negligible: for these software-based boxes, without special acceleration treatment, the C compiler performs most of the optimization work. Also, for such an action, the efficiency of a translated version is similar to the implemented version.

For FPGA platform, the performance gap between optimized implementation and OpenFunction language generated version (*i.e.*, FPGA-C) is significant (10x in this case) in Table II. The reason is that FPGA enables a programmer to introduce several optimizations. For example, for NAT function, the programmer may choose to calculate the checksum at line rate, without the need to receive complete packet.

Key Takeaway 2: The performance of SDM data plane actions, if generated from OpenFunction pseudo language, is dominated by the nature of system.

Similarly, for two stateful SDMs (FW and IPS), the performance is comparable to non-stateful SDMs.

Key Takeaway 3: OpenFunction abstraction has the potential to support various stateful firewalls.

Lastly, we consider two scenarios to evaluate OpenFunction data plane overhead: (1) time it takes to start/stop a data plane process in the service chain; and (2) additional path delays due to our chain enforcement inside a box. In the first case, we are interested in understanding how much delay is incurred when a new service chain or SDM DP is added or removed from the network. To evaluate, we use a small topology with five DPDK middleboxes and generate TCP flows between source and destination. We have two DPDK boxes on path 1 and one DPDK box on path 2. We initiate flows on path 1 and path 2 which initiates flow rule installation in the switches along the datapath and calculate datapath setup times. Our evaluation shows that it takes less than 100 msec to add a new datapath or service chain in the network. In the second case, we are interested in understanding the additional delays caused by enforcing service chains in the MB's. Our evaluations shows that it adds less than 20 usec per SDM DP in the service chain.

We evaluate the responsiveness of the controller to adapt to dynamic network conditions and measure the time required to reconfigure the network in case of SDM DP failure or traffic overload in some of the service chains. We consider same topology as before, with DPDK middleboxes, and use Iperf [1] to generate TCP flows between source and destination. We first initiate flows on $path1$ and $path2$ by install rules in the switches along the datapath and distribute load evenly across the two paths. Next, we bring down the switch in $path2$ and observe how quickly network converges to the stable state and diverts traffic to $path1$. Our evaluation shows that network takes less than 200 msec to converge to a stable state and reconfigure flows to use other paths. We also dynamically change the datapaths, to replicate the SDM insertion and deletion scenarios, in our topology to steer network traffic through different paths and we observe that the network quickly adapts to these changes without any drop in the flow throughput.

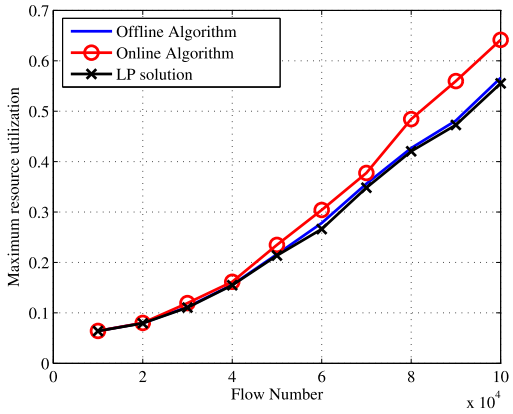


Fig. 10. Performance impacted by flow number.

Key Takeaway 4: OpenFunction abstraction is flexible and has light overhead.

B. Scheduling Performance

In this section, we evaluate our algorithm by leveraging the traffic data collected from a production datacenter hosted by IBM global services. To this end, we conduct performance evaluation from two perspectives. First, we evaluate the impact of varying number of flows in the system on algorithm performance, and then the impact of threshold selected at line 4 in Algorithm 1 on algorithm performance.

Performance Impact by Varying the Number of Flows: In this section, we fix the box number to be 10, there are 6 SDMs and the required SDM number of each flow is evenly distributed in the range [1,5]. Each SDM has 2 or 3 versions implemented in the system. In addition, we set the forbidden threshold to be 0.2, i.e. 20% of the $Z_{k,i,n}^{l,j,m,f}$ and the fraction value according to the LP solution will be set to 0 in each iteration. To see how the algorithm performance changes with the number of flows, we change the flow number from 10^4 to 10^5, and the simulation results are shown in Fig. 10. We evaluate the performance of proposed online and offline algorithm and compare it to the LP solution, which treats all the flows as splittable, and hence the LP solution is a lower bound of our problem. From Figure 10, we make two observations: *i) the online algorithm achieves performance very close to LP solution, and ii) its performance margin from offline approach improves with increasing number of flows in the system.*

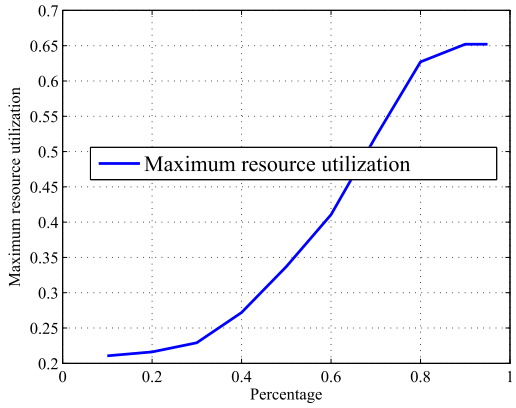


Fig. 11. Performance impacted by threshold ϵ.

A single flow only needs very little system resource compared to the available system resources, and therefore, we can treat all the flows as splittable in the system's point of view. As a result, the performance of offline algorithm is very close to the LP solution.

The performance of online algorithm deviates from the optimal solution as the number of flows increase in the system. This is for the reason that the online algorithm is a myopic algorithm and optimizes the flows one by one, whenever a flow enters the system, it optimizes the maximum resource utilization based on the current system utilization but does not consider future flow arrivals.

The Impact of Threshold ϵ:

In this section, we study how the forbidden parameter impacts the offline algorithm performance. For this purpose, we inject 400 flows into the system and change the value of forbidden threshold from 0.1 to 0.95.

The simulation results, Fig. 11, show that *for smaller thresholds, the offline algorithm performance degrades very slowly with increasing the value of forbidden threshold. However, when the threshold value exceeds a certain point, the algorithm performance degrades quickly.* The reason is that in the LP solution, there are many $Z_{k,i,n}^{l,j,m,f}$ with very little fraction, and they can be set to 0 at one time without impacting the performance significantly.

In addition, as the value of the forbidden threshold continues to increase, the performance degradation slows down. This happens because if we set too many $Z_{k,i,n}^{l,j,m,f}$ as zero, the optimization space for the flows reduces significantly. Therefore, forbidding more $Z_{k,i,n}^{l,j,m,f}$ cannot degrade the algorithm performance if the forbidden parameter is relatively large.

Key Takeaway 5: Load balancing is just one scheduling objective, and there could be more scheduling challenges for OpenFunction.

VIII. Conclusions

In this paper, we make three main contributions. First, we propose the concept of Software-Defined Middleboxes to realize abstraction for middleboxes, which complements existing SDN efforts. Second, we propose OpenFunction as an SDM data plane abstraction protocol. Third, we implemented a working OpenFunction system including one OpenFunction controller, three OpenFunction boxes, and four network functions including both stateful and stateless ones. Our experimental results show that the middlebox functions implemented by using our OpenFunction abstraction can achieve high performance and platform independence.

Acknowledgment

The authors would like to thank anonymous reviewers for their valuable comments.

References

[1] *iPerf: The TCP/UDP Bandwidth Measurement Tool*. Accessed: 2018. [Online]. Available: https://iperf.fr
[2] *OpenDataPlane*. Accessed: 2018. [Online]. Available: http://www.opendataplane.org/
[3] *Open vSwitch*. Accessed: 2018. [Online]. Available: http://openvswitch.org/
[4] M. Chiosi et al., "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action," in *Proc. SDN OpenFlow World Congr.*, 2012.
[5] C. Cui, "Network functions virtualisation: Network operator perspectives on industry progress. White Paper No. 3, Issue 1," in *Proc. SDN OpenFlow World Congr.*, Dusseldorf, Germany, 2014.
[6] M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, and A. Rindos, "Sdsecurity: A software defined security experimental framework," in *Proc. IEEE Int. Conf. Commun. Workshop (ICCW)*, Jun. 2015, pp. 1871–1876.
[7] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, p. 14.
[8] M. B. Anwer, M. Motiwala, M. B. Tariq, and N. Feamster, "Switchblade: A platform for rapid deployment of network protocols on programmable hardware," in *Proc. ACM SIGCOMM*, 2010, pp. 183–194.
[9] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. (2015). "On orchestrating virtual network functions in NFV." [Online]. Available: https://arxiv.org/abs/1503.06377
[10] B. Blodget, C. Bobda, M. Hübner, and A. Niyonkuru, "Partial and dynamically reconfiguration of Xilinx Virtex-II FPGAs," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2004, pp. 801–810.
[11] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.
[12] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: Enabling innovation in middlebox applications," in *Proc. ACM SIGCOMM Workshop Hot Topics Middleboxes Netw. Function Virtual.*, 2015, pp. 67–72.
[13] M. Dobrescu *et al.*, "RouteBricks: Exploiting parallelism to scale software routers," in *Proc. ACM SIGOPS*, 2009, pp. 15–28.
[14] 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. USENIX NSDI*, 2014, pp. 533–546.
[15] A. Gember *et al.* (2013). "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds." [Online]. Available: https://arxiv.org/abs/1305.0209
[16] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 163–174.
[17] I. Hadžić and J. M. Smith, "Balancing performance and flexibility with hardware support for network architectures," *ACM Trans. Comput. Syst.*, vol. 21, no. 4, pp. 375–411, 2003.
[18] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Proc. ACM Conf. SIGCOMM*, 2010, pp. 195–206.
[19] M. Handley, O. Hodson, and E. Kohler, "XORP: An open platform for network research," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 53–57, 2003.
[20] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible IP router software," in *Proc. USENIX NSDI*, 2005, pp. 189–202.
[21] *Data Plane Development Kit*, Intel, 2014.
[22] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proc. ACM Conf. SIGCOMM*, 2008, pp. 51–62.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[24] J. Larkins, "Recent trends in virtual network functions acceleration for carrier clouds," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, p. 3.

[25] L. E. Li *et al.*, "PACE: Policy-aware application cloud embedding," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 638–646.

[26] A. X. Liu, C. Meiners, E. Norige, and E. Torng, "High-speed application protocol parsing and extraction for deep flow inspection," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1864–1880, Oct. 2014.

[27] J. W. Lockwood *et al.*, "NETFPGA—An open platform for gigabit-rate network switching and routing," in *Proc. Microelectron. Syst. Educ.*, Jun. 2007, pp. 160–161.

[28] G. Lu *et al.*, "ServerSwitch: A programmable and high performance platform for data center networks," in *Proc. USENIX NSDI*, 2011, pp. 1–2.

[29] G. Lu, R. Miao, Y. Xiong, and C. Guo, "Using CPU as a traffic co-processing unit in commodity switches," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw.*, 2012, pp. 31–36.

[30] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Apr. 2008.

[31] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Des. Autom. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, 2012.

[32] J. C. Mogul, Y. Praveen, and T. Jean, "API design challenges for open router platforms on proprietary hardware," in *Proc. HotNets*, 2008, pp. 7–12.

[33] R. Narayanan *et al.*, "Macroflows and microflows: Enabling rapid network innovation through a split sdn data plane," in *Proc. Softw. Defined Netw. (EWSDN)*, Oct. 2012, pp. 79–84.

[34] S. W. O'Malley and L. L. Peterson, "A dynamic network architecture," *ACM Trans. Comput. Syst.*, vol. 10, no. 2, pp. 110–143, 1992.

[35] ONF. (2015). *OpenFlow Switch Specification Version 1.5.0*. [Online]. Available: https://goo.gl/oACYmp

[36] S. Ortiz, "Software-defined networking: On the verge of a breakthrough?" *Computer*, vol. 46, no. 7, pp. 10–12, 2013.

[37] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. Symp. Oper. Syst. Principles*, 2015, pp. 121–136.

[38] A. Panda *et al.*, "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, vol. 16. 2016, pp. 203–216.

[39] Z. A. Qazi *et al.*, "SIMPLE-fying middlebox policy enforcement using SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 27–38, 2013.

[40] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, p. 1.

[41] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Escape capsule: Explicit state is robust and scalable," presented at the 14th Workshop Hot Topics Oper. Syst., 2013.

[42] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. USENIX NSDI*, 2013, pp. 227–240.

[43] L. Rizzo, "NetMap: A novel framework for fast packet I/O," in *Proc. USENIX Annu. Conf.*, 2012, pp. 101–112.

[44] M. N. Sadiku, A. E. Shadare, S. Koay, and S. M. Musa, "Software-defined security," *Int. J. Eng. Res. Adv. Technol.*, vol. 2, no. 10, pp. 13–17, 2016.

[45] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. USENIX NSDI*, 2012, p. 24.

[46] S. Shin *et al.*, "FRESCO: Modular composable security services for software-defined networks," in *Proc. NDSS*, 2013, pp. 1–16.

[47] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, 2013, pp. 127–132.

[48] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A GPU-accelerated stateful packet processing framework," in *Proc. USENIX Annu. Conf. (USENIX ATC)*, 2014, pp. 321–332.

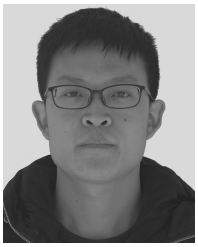
[49] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Mag.*, Aug. 2009.

[50] A. Darabseh *et al.*, "SDSecurity: A software defined security experimental framework," in *Proc. IEEE Int. Conf. Commun. Workshop (ICCW)*, 2015, pp. 1871–1876.

Chen Tian received the B.S., M.S., and Ph.D. degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. From 2012 to 2013, he was a Post-Doctoral Researcher with the Department of Computer Science, Yale University. He was an Associate Professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology. He is currently an Associate Professor with the State Key Laboratory for Novel Software Technology, Nanjing University, China. His research interests include data center networks, network function virtualization, distributed systems, Internet streaming, and urban computing.

Ali Munir received the B.S. degree in electronics engineering and the M.S. degree in electrical engineering from the National University of Sciences and Technology, Pakistan. He is currently pursuing the Ph.D. degree with the Computer Science and Engineering Department, Michigan State University. His research interests focus on networking and security.

Alex X. Liu received the Ph.D. degree in computer science from The University of Texas at Austin in 2006. His research interests focus on networking and security. He received the IEEE and IFIP William C. Carter Award in 2004, the National Science Foundation CAREER Award in 2009, and the Michigan State University Withrow Distinguished Scholar Award in 2011. He received best paper awards from ICNP-2012, SRDS-2012, and LISA-2010. He is currently an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING, an Editor of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, and an Area Editor of *Computer Communications*.



Jie Yang received the B.S. degree in computer science from Nanjing University, China. He is currently pursuing the master's degree with the Computer Science Department, Nanjing University. His research interests focus on networking and computer architecture.

Yangming Zhao received the B.S. degree in communication engineering and the Ph.D. degree in communication and information system from the University of Electronic Science and Technology of China in 2008 and 2015, respectively. He is currently a Post-Doctoral Researcher with SUNY Buffalo. His research interests include network optimization and data center networks.