OpenBox: Enabling Innovation in Middlebox Applications

Anat Bremler-Barr
School of Computer Science
The Interdisciplinary Center
Herzliya, Israel
bremler@idc.ac.il

Yotam Harchol
School of Computer Science and Engineering
The Hebrew University
Jerusalem, Israel
yotamhc@cs.huji.ac.il

David Hay
School of Computer Science and Engineering
The Hebrew University
Jerusalem, Israel
dhay@cs.huji.ac.il

ABSTRACT

Contemporary networks contain many different kind of middleboxes that perform variety of advanced network functions. Currently, a special box is tailored to provide each such function. These special boxes are usually proprietary, and operators control over them is limited to the set of capabilities defined by the provider of each box. Nonetheless, many middleboxes perform very similar tasks.

In this paper we present *OpenBox*: a logically-centralized framework that makes advanced packet processing and monitoring easier, faster, more scalable, flexible, and innovative. OpenBox decouples the control plane of middleboxes from their data plane, and unifies the data plane of multiple middlebox applications using entities called *service instances*. On top of the centralized control plane everyone can develop *OpenBox applications*. An *OpenBox application*, formerly implemented as a separate middlebox, instructs the data plane how to process packets in order to achieve its intended function. *OpenBox service instances* reside in data plane and process packets according to policies defined by the control plane. They can be implemented in software or use specialized hardware.

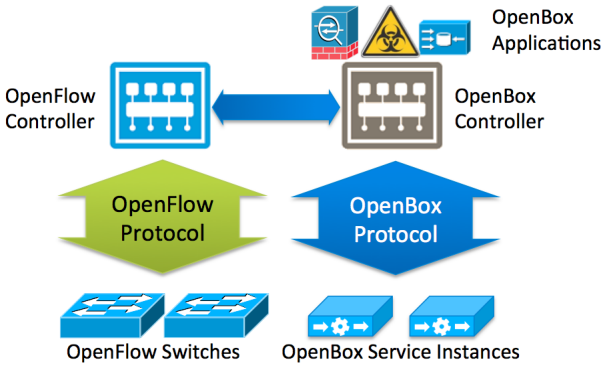

Figure 1: The general architecture of the OpenBox framework. OpenBox applications are programmed over the OpenBox controller, which sets the actual classification and monitoring rules in data plane's service instances. OpenBox controller communicates with an OpenFlow controller, if such exists, to control traffic flow to and from OpenBox service instances.

1. INTRODUCTION

Middleboxes play a major role in current large-scale networks such as datacenters, operators, and enterprise networks. Middleboxes are responsible for a wide range of performance and monitoring tasks, from simple address translation such as network address translator (NAT), or load balancing, to sophisticated security tools such as intrusion detection and prevention systems (NIDS/NIPS).

Currently, most middleboxes are implemented as real hardware boxes that are located in the exact point where the traffic they should operate on flows. Recent works suggest more flexible middlebox placement using software-defined networking (SDN) [16]. The call for network function virtualization (NFV) [5] propose implementing such middleboxes as virtual functions (i.e., in software), to reduce costs of purchasing, deployment, and management of such functions. In any case, middleboxes are usually closed-source and they expose a limited provider-specific control interface.

A closer look on the variety of existing middleboxes shows that most of them have many shared functionalities. For example, most middleboxes perform some sort of header fields analysis. Many middleboxes examine the application layer payload (a process usually referred as *deep packet inspection*, or DPI). Such middleboxes usually *reconstructs the TCP session* and *decompresses* the inspected traffic before performing the actual DPI process.

Network traffic in large-scale networks flow between several middleboxes before reaching its destination. For example, it is common that a packet would go through a firewall, NIPS and then a load balancer before reaching the desired application server it was destined at. Thus, the packet in this example will go through three very similar processing paths, for each of them there is a separate control interface, they may be managed by different administrators, and they may redundantly repeat each other's work (e.g., NIPS filters ports that were already filtered by firewall). Moreover, it is not rare to find multiple middleboxes with the same purpose

HotMiddlebox '15, August 21, 2015, London, United Kingdom
 ISBN 978-1-4503-3540-9/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785989.2785992

processing the same traffic, such as multiple NIPSs, each one with different rules, managed by a different administrator.

In this work we present *OpenBox*: a framework that decouples control from middleboxes data plane whose general architecture is shown in Figure 1. A high-level control plane defines monitoring and performance goals. Instead of legacy middleboxes, *OpenBox applications*, are developed on top on that control plane using a simple Java API. A low-level data plane unifies the processing stages of multiple middleboxes as instructed by the control. The data plane is composed of either separate or consolidated *OpenBox service instances* that provide the necessary functionality of the different processing stages.

Similarly to software-defined networks (SDN), having a centralized control with a simple programming API, and a unified interface to data plane, makes it easier to innovate both high-level applications and low-level data plane features. Several works have shown how simple middleboxes such as Layer-3 load balancers [21] can be implemented on SDN switches. However, for more sophisticated processing, a switch would naturally not suffice. OpenBox interplays with SDN environments such as OpenFlow [7], although the existence of such an environment is not a precondition.

Other benefits of the OpenBox framework are performance improvement when processing stages can be consolidated, better scalability and flexibility in resource provisioning, reduced cost of ownership and management, and easier multi-tenancy.

2. RELATED WORK

In recent years, network middleboxes have been a major topic of interest. Perhaps the most relevant related works are those calling to consolidate middleboxes, for the reasons listed before. In ComB [18], a new architecture is proposed to consolidate multiple middleboxes into a single location. A centralized control allocates middlebox applications throughout several consolidated locations in the network. Each consolidated location is built as a hypervisor that lets multiple middlebox applications to run. This increases hardware utilization and sometimes common procedures such as session reconstruction can be done once per flow for all middlebox applications. Though, most logic is not shared among applications, and each middlebox performs its own code on network traffic. The allocation mechanism presented in this work can be used for service instance allocation in our data plane.

xOMB [1] is a platform to create middleboxes using a general purpose server that provides a programmable general packet processing pipeline. It allows to program simple middlebox such as load balancing switch, and NAT. However, it does not consolidate multiple applications to the same processing pipeline, and does not have any network-wide view.

Kekely et al. [12] presents a hardware architecture for unified flow measurement and collection of application layer protocols information. It uses a centralized control software to aggregate demands from multiple middlebox applications and achieve them using their suggested hardware. OpenBox can use such hardware implementations in its data plane, while providing richer control logic for wider range of applications.

OpenBox uses network services to provide the necessary processing for packets. An example for such a service that

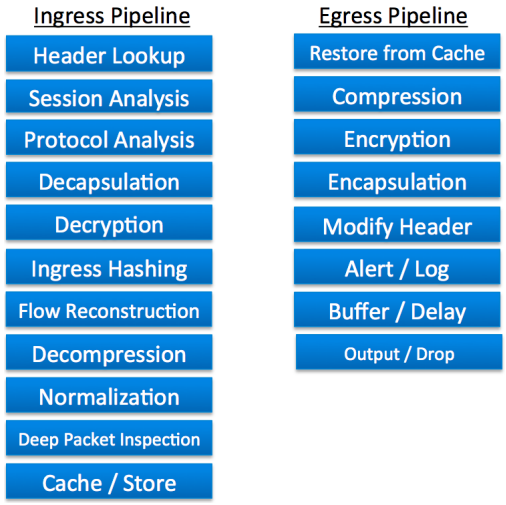

Figure 2: An example for general packet processing stages for multiple Box applications.

is centrally controlled is DPI, which was presented in [3]. In some sense, OpenBox extends this idea and generalizes it.

Another problem with middleboxes is that they should be placed in locations where the traffic they should handle flows. Several works tried to provide higher flexibility, either by modifying the link layer [11] or using software-defined networking [8, 16]. In addition to the placement problem, [8] also deals with the problem of provisioning new of instances of middlebox applications. Gember et al. [9] proposes a centralized control plane for sharing information between software middlebox applications (network functions). However, this work focuses only on the state maintained by each middlebox and the forwarding problem, so in a sense it is orthogonal to OpenBox. Sherry et al. [20] propose outsourcing the middleboxes to cloud services. This is completely orthogonal to our work as OpenBox can be used in the cloud in order to provide the outsourced middlebox functionality, or locally, instead of outsourcing at all.

Two interesting works can be used as building blocks for the OpenBox data plane service instance: The Click modular software router [13] is an extendable software package for programming network routers and packet processors. It has numerous modules for advanced routing and packet processing and additional modules can be added using the provided API. By adding a communication layer that communicates with the OpenBox controller, the Click package can be used as a basic service instance.

The other related work in this context is the P4 programmable packet processor language [2]. The P4 language aims to define the match-action table of a general purpose packet processor, such that it is not coupled with a specific protocol or specification (e.g., OpenFlow of a specific version).

3. UNIFIED PROCESSING

We surveyed a wide range of middlebox applications to understand the stages of processing performed by each one of them. Each application is implemented as an ordered list of processing steps (similarly to a pipeline, though stages are not always completely separate and parallel, as in a

Application	Header Lookup	Session Analysis	Protocol Analysis	Flow Reconstruction	Decompression	DPI	Modify Header	Alert / Log	Delay / Buffer	Output / Drop
NAT	V	V	X	X	X	X	V	X	X	V
L7 Load Balancer	V	V	X	X	X	V	V	X	X	V
L7 Traffic Shaper	V	V	V	V	V	V	V	X	V	V
Web Application Firewall	V	X	X	X	X	V	X	X	X	V
NIDS	V	V	V	V	V	V	X	V	X	X
NIPS	V	V	V	V	V	V	X	V	X	V
Network Anti Virus	V	V	V	V	V	V	X	V	X	V
Data Leakage Prevention	V	V	V	V	V	V	X	V	X	V
IPv6 Translator	V	X	X	X	X	X	V	X	X	V

Table 1: Breakdown of processing stages for variety of middlebox applications.

Box	Priority	Header Match	Payload Match	Actions
NIPS	HIGH	ETH_TYPE=IPv4 IP_PROTO=TCP TCP_SRC=2040	Exact string `"attack"` byte 0 to 250	Alert, Drop
Layer 7 Load Balancer	NORMAL	ETH_TYPE=IPv4 IP_PROTO=TCP IPV4_DST=10.0.0.10 TCP_DST=80	Exact string `"GET /images/"` byte 0 to 0	Output to port 1 Rewrite header: IPV4_DST=10.0.0.20
Layer 7 Load Balancer	NORMAL	ETH_TYPE=IPv4 IP_PROTO=TCP IPV4_DST=10.0.0.10 TCP_DST=80	Exact string `"GET /database/"` byte 0 to 0	Output to port 2 Rewrite header: IPV4_DST=10.0.0.30
Layer 7 Load Balancer	NORMAL	ETH_TYPE=IPv4 IP_PROTO=TCP IPV4_DST=10.0.0.10 TCP_DST=80	Regular expression `"GET /[^(images|database)]/"` byte 0 to 0	Output to port 3 Rewrite header: IPV4_DST=10.0.0.40

Table 2: Sample rules for NIPS and Layer 7 load balancer middlebox applications. By assigning priority for rules we resolve conflicting policies of multiple applications. In this example, since the priority of the NIPS rule is higher, when a HTTP request packet comes from port 2040 to IP 10.0.0.10 port 80, and contains the string "attack" (thus matching two rules), it will be dropped, and not forwarded to the destination server.

pipeline). These stages can be categorized as most of them are fairly similar in most middlebox applications. The result of each stage is the input of the next stages, and usually it is either a modified packet or part of it, or some metadata, along with a possible state update. For example, most middleboxes perform some sort of header fields lookup before processing a packet. The result of this lookup is some metadata used by later stages. Many middleboxes look into packet's payload using DPI. The result of this stage is metadata that tells which patterns were found in the payload.

A sample of our survey can be viewed in Table 1. A complete list of the processing stages suggested in OpenBox is presented in Figure 2. In this section we show how multiple middlebox applications can be implemented by using a subset of the processing stages, providing the requested functionality.

3.1 Unifying Processing Stages

In general, middleboxes use some *rule* mechanism. Examples for such rules are shown in Table 2 for a NIPS and a Layer 7 load balancer. Rules define *matches* on input traffic, and some corresponding *actions* to be taken. More specifically, matches in middleboxes' ingress processing stages impose actions to be taken in their egress processing stages. Thus, matches define the behavior of ingress processing stages, while actions define the behavior of egress processing stages. We would like to merge multiple rulesets to define the logic of a single logical middlebox application that fulfills the union of all.

In our framework, multiple rules may match a single packet. In such a case, multiple actions should be taken based on all rules that matched. In some cases two matching rules may contradict each other. An example is shown in Table 2: while load balancer rules instructs to output packets to a specified port, NIPS rules may dictate dropping the same packets. For such cases, each rule is assigned a priority, and the highest priority rule that matches is the one to eventually dictate the final behavior. An OpenBox application may set the default priority for its rules such that by default, all of its rules will have a priority different than normal (this is useful, for example, in security applications that would like to override other applications in case of a conflict).

3.2 Enhanced Performance

The decision on whether to unify the processing of multiple applications mainly depends on the potential perfor-

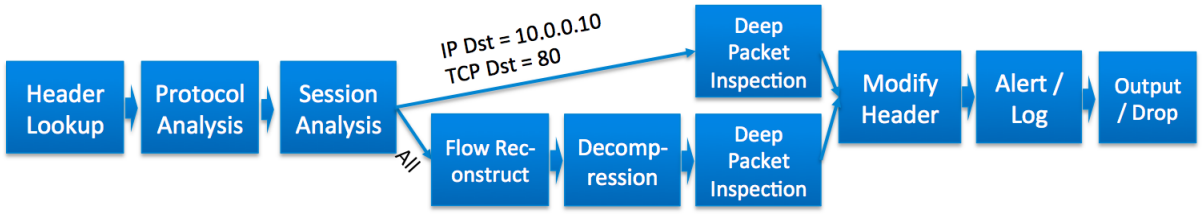

Figure 3: Processing graph for a data plane that unifies a NIPS and a Layer 7 load balancer. While all traffic should go through NIPS, which requires session analysis, decompression, text normalization, and deep packet inspection, HTTP traffic should also go through deep packet inspection to identify Layer 7 signatures used by load balancer. Load balancer is assumed here to search patterns on raw payload (without decompression or normalization), thus the fork in the graph. Actions are taken based on rules from both NIPS and load balancer (see Table 2).

mance gain (or loss). Naturally, we would like to unify the processing when a potential performance gain exists. Usually, packet processing functions are fairly simple with regard to computational complexity. Let n be the number of rules of some middlebox, and $f(n)$ be the processing function for a packet. The run-time complexity of $f(n)$ is usually sublinear with n (that is, $f(n) = o(n)$). Thus, given two rulesets with sizes n and m rules each, $f(n+m) < f(n) + f(m)$. In general, it makes sense to unify the processing for tasks that are sublinear with the number of rules. This criteria is true for most packet processing tasks, from simple IP lookup to complex deep packet inspection algorithms [3].

3.3 Dependencies and Precedence

In today's networks middleboxes are ordered such that packets go through them one after each other, according to the policies of the network [8, 16]. The order in which a packet should go through multiple services or middlebox is usually called a *policy chain* or a *service chain*.

When unifying the processing stages of multiple middleboxes applications we can make packets go through multiple applications at once. However, this may produce incorrect results in the cases when the result of one application (such as NAT) should be the input of a second application (such as a firewall). In this case, a packet will be processed twice: once with the rules from the first application, then, the output will be redirected to make another pass with the rules from the second application.

3.4 Room for Innovation

Of course, the OpenBox framework and the list of processing stages suggested in Figure 2 may not suffice to program *all* types of middleboxes. An example for one such middlebox is a WAN optimizer, which decodes, compresses or transforms network traffic according to specific policies. The specific transformation executed by each optimizer may not be common or may be proprietary. However, a vendor may provide WAN optimizer OpenBox application by providing, along with it, a modified OpenBox service instance implementation. By placing this version of service instance in the locations in network where WAN optimization is required, the provided OpenBox WAN Optimizer application can operate on network traffic as intended.

4. OPENBOX ARCHITECTURE

In this section we present the architecture of the OpenBox framework. The general framework consists of a data plane where OpenBox service instances are located, and a control plane, which sets the rules and actions for data plane processing.

4.1 Data Plane

The data plane of OpenBox consists of *OpenBox service instances* (OBIs), which are low-level processing entities that perform one or more stages of the unified processing described in Section 3. Each such OBI receives a *processing graph* (PG) and a set of *processing rules* from the OpenBox controller (described in Section 4.4). The PG defines the processing stages of packets that go through an OBI. An example for a processing graph is shown in Figure 3. In this example the data plane unifies the processing for both a NIPS and a Layer 7 load balancer, based on the rules in Table 2. The rules specify information to be looked at in packet headers or payload, and corresponding actions to be taken.

Hardware or Software Implementation.

Each OBI may be implemented either purely in software, running on a general-purpose CPU or as a virtual machine in the network (denoted as a *virtual OBI*), or using some specialized hardware as used in many middleboxes today. However, even if implemented using specialized hardware, the OBI should still communicate with the OpenBox controller using the OpenBox protocol and thus is still programmable just like a software-based OBI.

Multiple OBIs.

Each OBI may implement all or part of the required processing steps. An OBI that only implements part of the steps attaches the results of its processing as metadata to the original packet. There are numerous ways to do that but the cleanest is using *network service header* [17] which is designed for such cases[1]. In this case, another OBI that implements the next required processing steps should receive the packet with the metadata and continue processing it, or use the metadata to take the necessary actions.

A network may contain multiple OBIs, either to separate the processing steps as described above or also to load-balance network resources such as links and servers. Virtual OBIs can be brought up and down as network load changes, and can be migrated if traffic sources are migrated within a datacenter, for example using OpenNF [9]. Packets should flow from ingress points or source hosts through a chain of

[1]While NSH is not yet available on every network, Cisco's vPath [19] implements a similar idea and attempts to use it in SDN are in progress [14].

OBIs, denoted as a *service chain*, before reaching their destination or egress point. We elaborate on this in Section 4.4.

4.2 The OpenBox Protocol

The OpenBox protocol is the communication protocol between OBIs and the control plane. It defines the various messages and data structures to be sent between an OpenBox controller and OBIs. For example, it allows adding and removing rules, specifying processing graphs, sending alerts and log messages, and inquire OBIs for statistics.

Messages in the OpenBox protocol are sent in JSON format [4], which makes it easy to develop OBIs and controllers. The definition of the OpenBox protocol is still in progress.

4.3 Rules

Each rule comprises of a *header match* structure, a *payload match* structure, and one or more *instructions*. A header match specifies values for specific header fields to be matched in order for the rule to be applied. A payload match contains one or more patterns to be searched in packet's payload. These can either be exact strings or regular expressions. If both header and payload were matched, instructions are executed. Currently we only define instructions for applying a list of *actions*, but more instruction types can be added to the protocol.

Each rule may also be assigned a priority (as explained in Section 3.1), and a *cookie* - a 64-bits word assigned by the application for that rule, which makes it possible to later query the data plane on specific rules or make changes.

4.4 Control Plane

The *OpenBox controller* (OBC) is a logically centralized software server that communicated with the various OBIs in the network. Using the *Box abstraction layer* it exposes, OpenBox applications can be written on top of it to define the desired traffic monitoring and classification tasks in the network.

The Box abstraction layer (BAL) is an abstract API for writing OpenBox applications. In general, an OpenBox application specifies a processing path (which consists of multiple processing steps in some specified order) and a set of rules, which define actions to be taken based on information from traffic. The BAL provides a set of classes and interfaces to define such OpenBox applications.

The OBC has a global view of the network: it knows topology from the OpenFlow controller (OFC) and locations and properties of OBIs as they report this information directly to OBC. Thus, the OBC is in charge of splitting the processing load over multiple OBIs, defining service chains between them, and communicating with the OFC to enforce these service chains (using a *traffic steering SDN application* such as [8, 16]).

Given the knowledge about OBIs in the network and multiple processing paths from BAL, the OBC constructs a PG for each OBI by merging the paths (or partial paths) selected for execution on a specific OBI. The OBC decides which processing stages should be unified, and is responsible for precedence validation and for verifying the correctness of the resulting processing steps.

The control plane may also be in charge of OBI provisioning in data plane, as OBIs can report their status, such as their current load and throughput, to the controller.

4.5 OpenBox Applications

An OpenBox application defines a single traffic monitoring or classification application, such as a NIDS or a NAT, or new kinds of applications yet to be invented. An OpenBox application essentially defines which processing steps it requires and how each such step should work. In addition, it defines a set of rules.

4.5.1 Metadata

Each processing step produces metadata that may be used by later steps. For example, when implementing NAT, the source IP address and transport port are later used to determine the NAT port number. An OpenBox application can specify for each processing module what data it needs for next steps, and this data will be stored in a metadata map in the OBI. Later processing steps can access this map and retrieve the stored information.

4.5.2 Proactive Approach

To preserve high performance, Box applications are mostly proactive, as they proactively define the behavior of data plane without waiting for packets to first arrive at the network (as sometimes performed in OpenFlow applications). To allow such a proactive approach, OpenBox defines the *spawn-rule* action, which dynamically creates rules in data plane based on incoming traffic.

An example for a simple application that requires such an action is NAT: Upon the arrival of the first packet of a session, translation rules for outbound and inbound traffic of this session are created automatically in data plane. In OpenFlow, where dynamic rule spawning is not available, a reactive approach must be taken, and the first packet of every new session must go to controller, which plants the new rules back in data plane.

All rules spawned from the same parent rule by the same spawn-rule action have the same cookie value, which helps grouping them together in further operations such as statistics queries and ruleset modification.

Due to the proactive approach of OpenBox, applications use event handlers in code only to handle events such as alerts and logging.

4.5.3 State Management

Many middleboxes keep some sort of a state across multiple packets of the flow, or a collection of flows (usually denoted as a *session*). This state helps the different modules of a middlebox to remember information across different packets of a session and consequently take actions based not only the current packet but also its preceding (and sometimes succeeding) packets.

The *session analyzer* module of OpenBox is responsible, among other tasks, to generate a unique session identifier for packets of the same session. Then, each OBI holds a map for each session ID, which plays as a key-value store for the corresponding state information. For example, for NAT implementation the random port number and original port and address will be stored. Alternatively, an OBI that performs payload analysis such as decompression and deep packet inspection would store the compression window required to decompress next packets (e.g., in gzip, last 32KB of uncompressed data), and the state where the DPI automaton stopped scanning the last packet.

4.5.4 Content Storage

Some applications require storing content such as payload or whole packets for future use. Simple examples are web cache, which stores HTTP responses, and spam filter, which quarantines messages for future inspection. The OpenBox Storage Server (OBS) stores content received from OBIs, making it available for OpenBox applications and for OBIs. Multiple instances of OBS can exist in a network, usually one per each OBI or a set of OBIs. Data is stored separately for different applications for security purposes, and can be retrieved by the OpenBox Application that stored it or by an OBI on behalf of that application, using a key (e.g., a URL). OBS instances can be placed on the same physical machine that runs their corresponding OBI or in a remote location, thus data is sent to and from it over the network.

5. INITIAL IMPLEMENTATION

We have began implementation of an OpenBox framework that consists of an OpenBox controller named *Moonlight*, implemented in Java (sources: https://github.com/DeepnessLab/moonlight). OBIs are implemented in Python and C (sources: https://github.com/DeepnessLab/obsi). In addition, we implemented an OpenFlow 1.3 based traffic steering application (TSA) that manages and enforces service chains, as an OpenDaylight [6] bundle. We emulate a network in Mininet [15] where OBIs and OBC run as hosts.

The Moonlight controller defines the following processing modules: *header lookup*, *session analyzer*, *protocol analyzer*, *ingress hasher*, *payload handler*, and *decision maker*. Each module has some settings that can be applied to it. For example, an OpenBox application can tell the header lookup module which fields it needs for future processing and thus should be saved in metadata map. The payload handler can be set to use decompression and normalization modules.

On top of the Moonlight controller we have implemented three sample OpenBox applications: a NAT, a NIPS, and a L7 load balancer. Their code is available under the Moonlight github repository. The NAT implementation, which provides fully-functional NAT capabilities, takes about 100 lines of code.

6. CONCLUSIONS AND FUTURE WORK

In this paper we present OpenBox - an open framework for development of advanced network control applications on top of a network that contains general purpose service instances. The OpenBox framework can replace legacy middleboxes and provide more flexible and scalable development and deployment of applications with the same roles, and allow innovative solutions to be easily created. The framework can be augmented and extended by any vendor by adding more sophisticated functionalities to its data plane.

Our research now focuses on finishing the definition of the OpenBox protocol, and the implementation of the OpenBox service instances and the Moonlight OpenBox controller. Future research may focus on interesting problems of the suggested framework, such as smart allocation of OBIs, smart allocation of tasks for OBIs to avoid overloading, and questions related to traffic engineering in a network with OpenBox. Of course, the framework invites innovative development of applications for enhanced network security and performance.

Acknowledgments

This research was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n° 259085, the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), and the Neptune Consortium, administered by the Office of the Chief Scientist of the Israeli ministry of Industry, Trade, and Labor.

7. REFERENCES

[1] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: extensible open middleboxes with commodity servers. In *ANCS*, pages 49–60, 2012.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, Jul 2014.

[3] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Deep packet inspection as a service. In *CoNEXT*, pages 271–282, 2014.

[4] ECMA. The JSON data interchange format, October 2013. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[5] ETSI. Network functions virtualisation - introductory white paper, 2012. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[6] L. Foundation. Opendaylight. http://www.opendaylight.org/.

[7] O. N. Foundation. Openflow switch specification version 1.4.0, October 2013. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[8] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, abs/1305.0209, 2013.

[9] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: enabling innovation in network function control. In *SIGCOMM*, pages 163–174, 2014.

[10] V. Heorhiadi, M. K. Reiter, and V. Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In *CoNEXT*, pages 361–372, 2012.

[11] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, pages 51–62, 2008.

[12] L. Kekely, V. Pus, and J. Korenek. Software defined monitoring of application protocols. In *INFOCOM*, pages 1725–1733, 2014.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug 2000.

[14] P. Kothari. Network Service Header support for OVS. OVS Code Patch, September 2013. http://openvswitch.org/pipermail/dev/2013-September/032036.html.

[15] Mininet. http://mininet.org/.

[16] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM*, pages 27–38, 2013.

[17] P. Quinn, P. Agarwal, R. Manur, R. Fernando, J. Guichard, S. Kumar, A. Chauhan, M. Smith, N. Yadav, and B. McConnell. Network service header. IETF Internet-Draft, February 2014. https://datatracker.ietf.org/doc/draft-quinn-sfc-nsh.

[18] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *NSDI*, pages 323–336, 2012.

[19] N. Shah. Cisco vPath technology enabling best-in-class cloud network services, August 2013. http://bit.ly/1F6bbMH.

[20] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *SIGCOMM*, pages 13–24, 2012.

[21] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE'11*, pages 12–12, 2011.