OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions

Anat Bremler-Barr [*]
bremler@idc.ac.il

Yotam Harchol [†]
yotamhc@cs.huji.ac.il

David Hay[†]
dhay@cs.huji.ac.il

[*] School of Computer Science, The Interdisciplinary Center, Herzliya, Israel
[†] School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel

ABSTRACT

We present OpenBox — a software-defined framework for network-wide development, deployment, and management of *network functions* (NFs). OpenBox effectively decouples the control plane of NFs from their data plane, similarly to SDN solutions that only address the network's forwarding plane.

OpenBox consists of three logic components. First, user-defined *OpenBox applications* provide NF specifications through the OpenBox north-bound API. Second, a logically-centralized *OpenBox controller* is able to merge logic of multiple NFs, possibly from multiple tenants, and to use a network-wide view to efficiently deploy and scale NFs across the network data plane. Finally, *OpenBox instances* constitute OpenBox's data plane and are implemented either purely in software or contain specific hardware accelerators (e.g., a TCAM). In practice, different NFs carry out similar processing steps on the same packet, and our experiments indeed show a significant improvement of the network performance when using OpenBox. Moreover, OpenBox readily supports smart NF placement, NF scaling, and multi-tenancy through its controller.

CCS Concepts

•**Networks** → **Middle boxes / network appliances;** *Programming interfaces; Network control algorithms;*

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

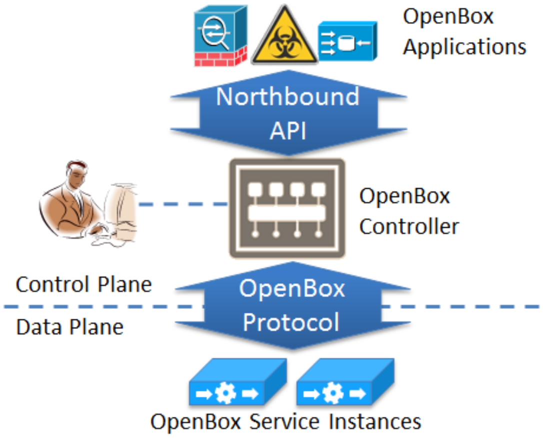


Figure 1: The general architecture of the OpenBox framework.

Keywords

Network functions; Middleboxes; Software-Defined Networks

1. INTRODUCTION

Software-defined networking (SDN) has been a tremendous game-changer, as it decouples the control plane of network forwarding appliances (e.g., switches and routers) from their data plane. SDN has succeeded in solving important problems in the forwarding plane, such as cost, management, multi-tenancy, and high entry barriers that limit innovation.

However, in current SDN solutions, such as OpenFlow [15], only the forwarding appliances are software-defined, while the other data plane appliances continue to suffer from all of the above problems. Moreover, these appliances, which are usually referred to as *network functions* (NFs) or middleboxes, often suffer from additional, more complex problems as well. Yet studies show that NFs constitute 40%-60% of the appliances deployed in large-scale networks [39]. In this paper we present OpenBox, a framework and a protocol that make network functions software-defined, using a logically centralized controller.

Traditionally, each middlebox was marketed as a single piece of hardware, with its proprietary software already installed on it, for a high price. This prevented, as noted above, on-demand scaling and provisioning. The call for *network function virtualization* (NFV) [11] aims to reduce the cost of ownership and management of NFs by making NFs virtual appliances, running on top of a hypervisor or in a container. While NFV improves on-demand scaling and provisioning, it does not solve other problems such as the limited and separate management of each NF.

Network traffic nowadays usually traverses a sequence of NFs (a.k.a. a *service chain*). For example, a packet may go through a firewall, then through an Intrusion Prevention System (IPS), and then through a load balancer, before reaching its destination. A closer look into these NFs shows that many of them process the packets using very similar processing steps. For example, most NFs parse packet headers and then classify the packets on the basis of these headers, while some NFs modify specific header fields, or also classify packets based on Layer 7 payload content. Nonetheless, each NF has its own logic for these common steps. Moreover, each NF has its own management interface: each might be managed by a different administrator, who does not know, or should not know, about the existence and the logic of the other NFs.

Our OpenBox framework addresses the challenges of efficient NF management by completely decoupling the control plane of a NF from its data plane using a newly defined communication protocol [35], whose highlights are presented in this paper. The observation that many NFs have similar data planes but different control logic is leveraged in OpenBox to define general-purpose (yet flexible and programmable) data plane entities called *OpenBox Instances* (OBIs), and a logically-centralized control plane, which we call the *OpenBox Controller* (OBC). NFs are now written as *OpenBox Applications* on top of the OBC, using a *northbound programming API*. The OBC is in charge of deploying application logic in the data plane and realizing the intended behavior of the applications in the data path. The OpenBox protocol defines the communication channel between the OBC and the OBIs.

To the best of our knowledge, we are the first to introduce such a general framework for software-defined NFs, which includes specifications for a logically-centralized controller and its northbound API for NF application development, for an extensible data plane instance (OBI), and for a communication protocol between the two. We also propose a novel algorithm for merging the core logic of multiple NF applications in the control plane, such that computationally-intensive procedures (e.g., packet classification or DPI) are only performed once for each packet. Since this implies that packets may traverse a smaller number of physical locations, latency is reduced and resources can be reallocated to provide higher throughput. We compare our work to previous works that apply SDN ideas to the middlebox domain [2, 17, 18, 33, 38] in Section 7.

We have implemented a prototype of the OpenBox framework and we provide a simple installation script that installs the entire system on a Mininet VM [28], so users can easily create a large-scale network environment and test their network functions. All our code is publicly available at [29].

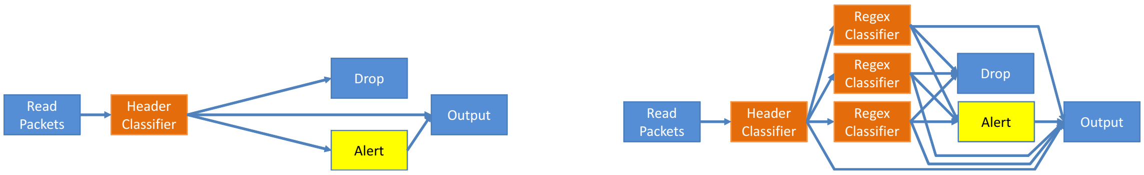
OpenBox promotes innovation in the NF domain. Developers can develop and deploy new NFs as OpenBox applications, using basic building blocks (e.g., header classification) provided by the framework. Furthermore, the capabilities of the OpenBox data plane can be extended beyond these basic building blocks: an application can provide an extension module code in the control plane. This module can then be injected into the corresponding OBIs in the data plane, without having to recompile or redeploy them.

OpenBox is designed to also support hardware-based OBIs, which use specific hardware to accelerate the data plane processing. For example, certain web optimizers may require specific video encoding hardware. However, the OBI with this hardware does not have to implement the entire set of processing logic defined by the OpenBox protocol. Instead, it can be chained with other, software- or hardware-based OBIs, which would provide the additional logic. This reduces the cost of the specialized OBI and the effort required to develop the NF application that uses it. Developers may also create a purely-software version of their optimizer OBI, which will be used by the OpenBox framework to scale up at peak load times.

2. ABSTRACTING PACKET PROCESSING

We surveyed a wide range of common network functions to understand the stages of packet processing performed by each. Most of these applications use a very similar set of processing steps. For example, most NFs do some sort of header-based classification. Then, some of them (e.g., translators, load balancers) do some packet modification. Others, such as intrusion prevention systems (IPSs) and data leakage prevention systems (DLP), further classify packets based on the content of the payload (a process usually referred to as *deep packet inspection* (DPI)). Some NFs use active queue management before transmitting packets. Others (such as firewalls and IPSs) drop some of the packets, or raise alerts to the system administrator.

In this section we discuss and present the abstraction of packet processing applications required to provide a framework for the development and deployment of a wide range of network functions.



(a) Firewall

(b) Intrusion Prevention System (IPS)

Figure 2: Sample processing graphs for firewall and intrusion prevention system NFs.

2.1 Processing Graph

Packet processing is abstracted as a *processing graph*, which is a directed acyclic graph of *processing blocks*. Each processing block represents a single, encapsulated logic unit to be performed on packets, such as header field classification, or header field modification. Each block has a single input port (except for a few special blocks) and zero or more output ports. When handling a packet, a block may push it forward to one or more of its output ports. Each output port is connected to an input port of another block using a *connector*.

The notion of processing blocks is similar to Click's notion of *elements* [23] and a processing graph is similar to Click's router configuration. However, the OpenBox protocol hides lower level aspects such as the Click push/pull mechanism, as these may be implementation-specific. A processing block can represent any operation on packets, or on data in general. A processing block can buffer packets and coalesce them before forwarding them to the next block, or split a packet.

In our implementation, described in Section 4, we use Click as our data plane execution engine. We map each OpenBox processing block to a compound set of Click elements, or to a new element we implemented, if no Click element was suitable.

Figure 2 shows sample processing graphs for a firewall network function (Fig. 2(a)) and an IPS network-function (Fig. 2(b)). The firewall, for example, reads packets, classifies them based on their header field values, and then either drops the packets, sends an alert to the system administrator and outputs them, or outputs them without any additional action. Each packet will traverse a single path of this graph.

Some processing blocks represent a very simple operation on packets, such as dropping all of them. Others may have complex logic, such as matching the packet's payload against a set of regular expressions and outputting the packet to the port that corresponds to the first matching regex, or decompressing gzip-compressed HTTP packets.

Our OpenBox protocol defines over 40 types of *abstract processing blocks* [35]. An abstract processing block may have several implementations in the data plane, depending on the underlying hardware and software in the OBI. For example, one block implementation might perform header classification using a trie in software while another might use a TCAM for this task [42]. As further explained in Section 3 and in the protocol specification [35], the OBI informs the controller about the implementations available for each supported abstract processing block. The controller can then specify the exact implementation it would like the OBI to use, or let the OBI use its default settings and choose the implementation itself. The OpenBox protocol also allows injecting new custom blocks from the controller to the OBI, as described in detail in Section 3.2.1.

Abstract Block Name	Role	Class
FromDevice	Read packets from interface	T
ToDevice	Write packets to interface	T
Discard	Drop packets	T
HeaderClassifier	Classify on header fields	C
RegexClassifier	Classify using regex match	C
HeaderPayload Classifier	Classify on header and payload	C
NetworkHeader FieldRewriter	Rewrite fields in header	M
Alert	Send an alert to controller	St
Log	Log a packet	St
ProtocolAnalyzer	Classify based on protocol	C
GzipDecompressor	Decompress HTTP packet/stream	M
HtmlNormalizer	Normalize HTML packet	M
BpsShaper	Limit data rate	Sh
FlowTracker	Mark flows	M
VlanEncapsulate	Push a VLAN tag	M
VlanDecapsulate	Pop a VLAN tag	M

Table 1: Partial list of abstract processing blocks. The *class* property is explained in Section 2.2.1.

Table 1 lists some of the fundamental abstract processing blocks defined by the OpenBox protocol. Each block has its own configuration parameters and additional information, as described in Section 3.2.

2.2 Merging Multiple Graphs

Our framework allows executing multiple network functions at a single data plane location. For example, pack-

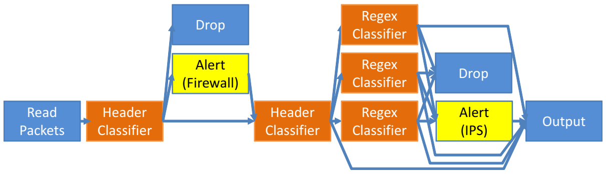


Figure 3: A naïve merge of the two processing graphs shown in Figure 2.

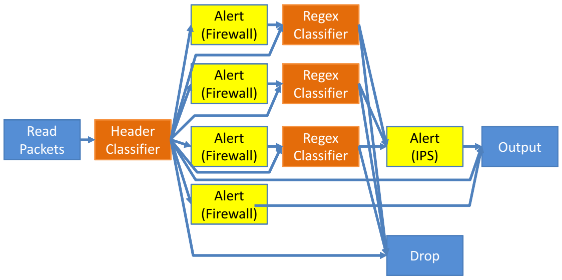


Figure 4: The result of our graph merge algorithm for the two processing graphs shown in Figure 2.

Algorithm 1 Path compression algorithm

```
function COMPRESSPATHS($G = (V, E)$, $root \in V$)
Require: $G$ is normalized
    $Q \leftarrow$ empty queue
    Add $(root, -1)$ to $Q$
    $start \leftarrow null$
    while $Q$ is not empty do
        $(current, port) \leftarrow Q.poll()$
        if $current$ is a classifier, modifier or shaper then
            if $start$ is null then                    ▷ Mark start of path
                $start \leftarrow current$
                for each outgoing connector $c$ from $current$ do
                    if $c.dst$ not in $Q$ then
                        Add $(c.dst, c.srcPort)$ to $Q$
                    end if
                end for
                continue
            else                          ▷ $start$ is not null - end of path
                $end \leftarrow current$
                if $start$ and $end$ are mergeable classifiers then
                    $merged \leftarrow merge(start, end)$
                    for each output port $p$ of $merged$ do
                        Clone the path from $start$'s correct
                        successor for port $p$ to $end$ (exclusive)
                        Mark clone of last block before $end$
                        Clone the sub-tree from $end$'s correct
                        successor for port $p$
                        Rewire connectors from $merged$ port $p$
                        to the clones and between clones
                    end for
                    $current \leftarrow merged$
                else                             ▷ Not mergeable classifiers
                    if $start$ and $end$ are classifiers then
                        Treat $start$ and $end$ as a single classifier
                        Find next classifier, modifier or shaper
                        and mark the last block before it.
                    end if
                end if
                for each outgoing connector $c$ from $current$ do
                    Find mergeable blocks from $c$ to a marked
                    block. Merge and rewire connectors
                end for
                if graph $G$ was changed then
                    Restart COMPRESSPATHS($G$, $current$)
                end if
            end if
        else                                    ▷ Skip statics, terminals
            for each outgoing connector $c$ from $current$ do
                if $c.dst$ not in $Q$ then
                    Add $(c.dst, c.srcPort)$ to $Q$
                end if
            end for
            continue
        end if
    end while
    return $G$
end function
```

ets may have to go through a firewall and then through an IPS. We could simply use multiple processing graphs at such locations, making packets traverse the graphs one by one, as shown in Figure 3. In this section we show how to merge multiple graphs while preserving the correct processing order and results.

Consider two network functions as shown in Figure 2, running at the same physical location in the data plane. We would like to merge the two graphs into one, such that the logic of the firewall is first executed on packets, followed by the execution of the IPS logic. Additionally, we would like to reduce the total delay incurred on packets by both NFs by reducing the number of blocks each packet traverses. The desired result of this process is shown in Figure 4: We would like packets to go through one header classification instead of two, and execute the logic that corresponds to the result of this classification.

2.2.1 Graph Merge Algorithm

Our graph merge algorithm must ensure that correctness is maintained: a packet must go through the same path of processing steps such that it will be classified, modified and queued the same way as if it went through the two distinct graphs. We also want to make sure that static operations such as alert or log will be executed on the same packet, at the same state, as they would without merging. Our goal in this process is to reduce the per-packet latency, so we would like to minimize the length of paths between input and output terminals in the graph.

In order to model the merge algorithm, we classify blocks into five classes:

- **Terminals** (T): blocks that start or terminate the processing of a packet.
- **Classifiers** (C): blocks that, given a packet, classify it according to certain criteria or rules and output it to a specific output port.
- **Modifiers** (M): blocks that modify packets.
- **Shapers** (Sh): blocks that perform traffic shaping tasks such as active queue management or rate limiting.

- **Statics** (St): blocks that do not modify the packet or its forwarding path, and in general do not belong to the classes above.

We use these classes in our graph merge algorithm in order to preserve the correctness of the merged graph: We can change the order of static blocks, or move classifiers before static blocks, but we cannot move classifiers across modifiers or shapers, as this might lead to incorrect classification. We can merge classifiers, as long as we pay special attention to the rule set combination and output paths. We can also merge statics and modifiers in some cases. The right column in Table 1 specifies the class of each block.

Our algorithm works in four stages. First, it normalizes each processing graph to a *processing tree*, so that paths do not converge.[1] Then, it concatenates the processing trees in the order in which the corresponding NFs are processed. Note that a single terminal in the original processing graph may correspond to several leaves in the processing tree. A copy of the subsequent processing tree will be concatenated to each of these leaves. Nevertheless, the length of any *path* in the tree (from root to leaf) is exactly the same as it was in the original processing graph, without normalization.[2]

While the number of blocks in the merged tree can increase multiplicatively,[3] in practice this rarely happens, and most importantly, the number of blocks in the graph has no effect on OBI performance. The significant parameter is the length of paths, as longer paths mean greater delay. Moreover, two graphs need not be merged if the overheads are too high. The controller is responsible for avoiding such a merger.

As the processing tree is in fact a collection of paths, the third stage in our algorithm is re-ordering and merging blocks along a path. This is shown in Algorithm 1. As mentioned before, the algorithm works by examining the class of the blocks and deciding whether blocks can be merged (Line 7).

Perhaps the most interesting case is merging two classifier blocks. Specifically, classifier blocks of the same type can support merging by having their own merge logic. The merge should resolve any conflicts according to the ordering and priorities of the two input applications (if applicable) and on the priority of the merged rules (Lines 18-29).

For example, in our implementation, the HeaderClassifier block is mergeable: it implements a specific Java interface and a `mergeWith(...)` method, which creates a cross-product of rules from both classifiers, orders them according to their priority, removes duplicate rules caused by the cross-product and empty rules caused by priority considerations, and outputs a new classifier that uses the merged rule set. After merging classifier blocks, our algorithm rewires the connectors and clones the egress paths from the classifiers such that packets will correctly go through the rest of the processing blocks. The merge algorithm is then applied recursively on each path, to compress these paths when possible. See the paths from the header classifier block in Figure 4 for the outcome of this process in our example.

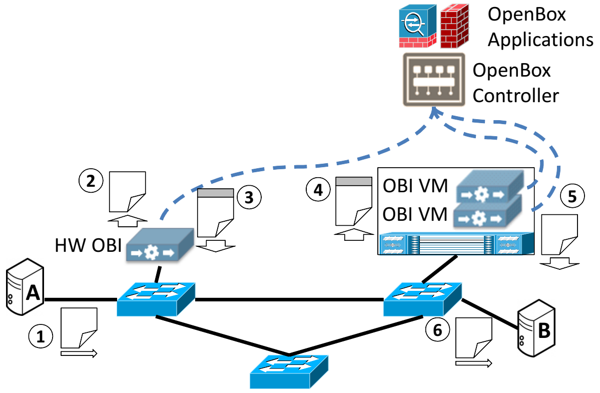


Figure 5: Sample OpenBox network with distributed data plane processing (as in Figure 6).

It is also possible to merge static and modifier blocks, if they are of the same class and type, and their parameters do not conflict. For example, two instances of a rewrite header block can be merged in constant time if they modify different fields, or the same field with the same value (lines 38-39 in Algorithm 1).

The last stage of our algorithm takes place after the merge process is completed. It eliminates copies of the same block and rewires the connectors to the remaining single copy, so that eventually the result is a graph as shown in Figure 4, and not necessarily a tree. Note that the diameter of the merged processing graph, as shown in Figure 4, is shorter (six blocks) than the diameter of the graph we would have obtained from a naïve merge (seven blocks, see Figure 3).

The correctness of the process stems from the following: First, any path a packet would take on the naïvely merged graph exists, and will be taken by the same packet, on the normalized and concatenated graph. Second, when merging classifiers we duplicate paths such that the previous property holds. Third, we only eliminate a copy of a block if the remaining copy is pointing to exactly the same path (or its exact copy).

3. OPENBOX FRAMEWORK ARCHITECTURE

In this section we describe the OpenBox framework in detail by dividing it into layers, as shown in Fig-

[1] The process of graph normalization may theoretically lead to an exponential number of blocks. This only happens with a certain graph structure, and it never happened in our experiments. However, if it does, our system rolls back to the naïve merge.

[2] The process of graph concatenation requires careful handling of special cases with regard to input and output terminals. We address these cases in our implementation. However, due to space considerations, we omit the technical details from the paper.

[3] For graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the number of blocks in the merged graph is up to $|V_1|^2(1 + |V_2|^2)$

ure 1: from OpenBox service instances (OBIs) in the data plane at the bottom, through the OpenBox protocol and the OpenBox controller (OBC), to the applications at the top.

3.1 Data Plane

The OpenBox data plane of consists *OpenBox service instances* (OBIs), which are low-level packet processors. An OBI receives a processing graph from the controller (described in Section 3.3). The OBI applies the graph it was assigned on packets that traverse it. It can also answer queries from the controller and report its load and system information.

OBIs can be implemented in software or hardware. Software implementations can run in a VM and be provisioned and scaled on demand. An OBI provides implementations for the abstract processing blocks it supports, and declares its implementation block types and their corresponding abstract block in the Hello message sent to the OBC. The controller may use a specific implementation in the processing graph it sends to the OBI, or use the abstract block name, leaving the choice of exact implementation to the OBI.

An OBI may be in charge of only part of a processing graph. In this case, one or more additional OBIs should be used to provide the remaining processing logic. A packet would go through a service chain of all corresponding OBIs, where each OBI attaches *metadata* (using some encapsulation technique [12, 19, 37] see also Section 3.4) to the packet before sending it to the next OBI. Upon receiving a packet from a previous OBI, the current OBI decodes the attached metadata and acts according to it.

For example, consider the merged processing graph shown in Figure 4 and suppose its header classification block can be implemented in hardware, e.g., using a TCAM. Thus, we can realize this processing graph using two OBIs. The first OBI, residing on a server or a dedicated middlebox equipped with the appropriate hardware, performs only packet classification. Only if the packet requires further processing does the first OBI store the classification result as metadata, attach this metadata to the packet, and send it to another, software-based OBI, to perform the rest of the processing. The split processing graphs are illustrated in Figure 6. Even an SDN switch that supports packet encapsulation could be used as the first OBI.

Figure 5 illustrates this scenario in a network-wide setting: packets from host A (Step 1 in the figure) to host B should go through the firewall and the IPS. This is realized using two OBIs as described above. The first performs header classification on hardware (Step 2), then sends the results as metadata attached to the packets (Step 3) to the next, software-based OBI. In this example, this OBI is scaled to two instances, multiplexed by the network for load balancing. It extracts metadata from the packets (Step 4), performs the rest of the processing graph, and sends the packets out without metadata (Step 5). Eventually the packets are forwarded to host B (Step 6).

In our implementation, we use NSH [37] to attach metadata to packets. Other methods such as VXLAN [24], Geneve [19], and FlowTags [12] can also be used but may require increasing the MTU in the network, which is a common practice in large scale networks [34]. Different OpenBox applications may require different size metadata. In most cases, we estimate the metadata to be a few bytes, as it should only tell the subsequent OBI which path in the processing graph it should follow. Nevertheless, it is important to note that attaching metadata to packets is required only when two blocks that have originated from the same OpenBox application are split between two OBIs.

Finally, an OBI can use external services for out-of-band operations such as logging and storage. The OpenBox protocol defines two such services, for packet logging and for packet storage (which can be used for caching or quarantine purposes). These services are provided by an external server, located either locally on the same machine as the OBI or remotely. The addresses and other parameters of these servers are set for the OBI by the OBC.

3.2 The OpenBox Protocol

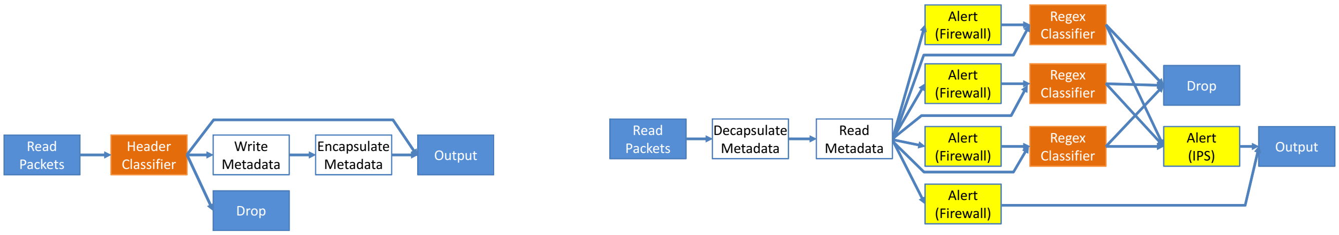
The OpenBox communication protocol [35] is used by OBIs and the controller (OBC) to communicate with each other. The protocol defines a set of messages for this communication and a broad set of processing blocks that can be used to build network function applications.

Abstract processing blocks are defined in the protocol specification. Each abstract block has its own configuration parameters. In addition, similarly to Click elements [23], blocks may have *read handles* and *write handles*. A read handle in our framework allows the controller, and applications that run on top of it, to request information from a specific processing block in the data plane. For example, it can ask a *Discard* block how many packets it has dropped. A write handle lets the control plane change a value of a block in the data plane. For example, it can be used to reset a counter, or to change a configuration parameter of a block.

3.2.1 Custom Module Injection

An important feature in the OpenBox protocol allows injecting custom software modules from the control plane to OBIs, if supported by the specific OBI implementation. Our implementation, described in Section 4, supports this capability. This allows application developers to extend existing OBIs in the data plane without having to change their code, or to compile and re-deploy them.

To add a custom module, an application developer creates a binary file of this module and then defines any new blocks implemented by this module, in the same way as existing blocks are defined in the protocol. The format of the binary file depends on the target OBI (in



(a) First OBI: Performs header classification on hardware TCAM and, if necessary, forwards the results as metadata along with the packet to next OBI.

(b) Second OBI: Receives header classification results and applies the corresponding processing path.

Figure 6: Distributed processing in data plane with the processing graph from Figure 4.

our implementation the file is a compiled Click module). When such a module is used, the controller sends an *AddCustomModuleRequest* message to the OBI, providing the module as a binary file along with metadata such as the module name. In addition, this message contains information required in order to translate the configuration from the OpenBox notation to the notation expected by the lower level code in the module.

A custom module should match the target OBI. A module developer may create multiple versions of a custom module and let the controller choose the one that is best suited to the actual target OBI.

3.3 Control Plane

The OpenBox controller (OBC) is a logically centralized software server. It is responsible for managing all aspects of the OBIs: setting processing logic, and controlling provisioning and scaling of instances. In an SDN network, the OBC can be attached to a traffic-steering application [36] to control chaining of instances and packet forwarding between them. OBC and OBIs communicate through a dual REST channel over HTTPS, and the protocol messages are encoded with JSON [10].

The OBC provides an abstraction layer that allows developers to create network-function applications by specifying their logic as processing graphs. We use the notion of *segments* to describe logical partitions in the data plane. Different segments can describe different departments, administrative domains, or tenants, and they can be configured with different policies and run different network function applications. Segments are hierarchical, so a segment can contain sub-segments. Each OBI belongs to a specific segment (which can, in turn, belong to a broader segment). Applications declare their logic by setting processing graphs to segments, or to specific OBIs. This approach allows for flexible policies in the network with regard to security, monitoring, and other NF tasks, and by definition, supports the trend of micro-segmentation [41]. Micro-segmentation reduces the size of network segments to allow highly customized network policies.

Upon connection of an OBI, the OBC determines the processing graphs that apply to this OBI in accordance with its location in the segment hierarchy. Then, for each OBI, the controller merges the corresponding graphs to a single graph and sends this merged processing graph to the instance, as discussed in Section 3.2. Our OBC implementation uses the algorithm presented in Section 2.2 to merge the processing graphs.

The controller can request system information, such as CPU load and memory usage, from OBIs. It can use this information to scale and provision additional service instances, or merge the tasks of multiple underutilized instances and take some of them down. Applications can also be aware of this information and, for example, reduce the complexity of their processing when the system is under heavy load (to avoid packet loss or to preserve SLAs).

The OBC, which knows the network topology and the OBI locations, is in charge of setting the forwarding policy chains. It does so on the basis of the actual deployment of processing graphs to OBIs. As OpenBox applications are defined per segment, the OBC is in charge of deciding which OBI(s) in a segment will be responsible for a certain task, and directing the corresponding traffic to this OBI.

3.4 OpenBox Applications

An *application* defines a single network function (NF) by *statement* declarations. Each statement consists of a *location specifier*, which specifies a network segment or a specific OBI, and a processing graph associated with this location.

Applications are event-driven, where upstream events arrive at the application through the OBC. Such events may cause applications to change their state and may trigger downstream reconfiguration messages to the data plane. For example, an IPS can detect an attack when alerts are sent to it from the data plane, and then change its policies in order to respond to the attack; these policy changes correspond to re-configuration messages in the data plane (e.g., block specific network segments, block other suspicious traffic, or block outgoing traffic to prevent data leakage). Another example is a request for load information from a specific OBI. This request is sent from the application through the OBC to the OBI as a downstream message, which will later trigger an event (sent upstream) with the data.

Although events may be frequent (it depends on the applications), graph changes are not frequent in general, as application logic does not change often. Applications that are expected to change their logic too frequently may be marked so that the merge algorithm will not be applied on them. The controller can also detect and mark such applications automatically.

3.4.1 Multi-Tenancy

The OpenBox architecture allows multiple network tenants to deploy their NFs through the same OBC. For example, an enterprise chief system administrator may deploy the OpenBox framework in the enterprise network and allow department system administrators to use it in order to deploy their desired network functions.

The OBC is responsible for the correct deployment in the data plane, including preserving application priority and ordering. Sharing the data plane among multiple tenants helps reduce cost of ownership and operating expenditure as OBIs in the data plane may have much higher utilization, as discussed in Section 5.

3.4.2 Application State Management

Network functions are, in many cases, stateful. That is, they store state information and use it when handling multiple packets of the same session. For example, Snort stores information about each flow, which includes, among other things, its protocol and other flags it may be marked with [40].

Since the state information is used in the data plane of NFs as part of their packet processing, it is important to store this information in the data plane, so it can be quickly fetched and updated. It cannot, for example, be stored in the control plane. Hence, the OpenBox protocol defines two data structures that are provided by the OBIs, in the data plane, for storing and retrieving state information.

The *metadata storage* is a short-lived key-value storage that can be used by an application developer to pass information along with a specific packet, as it traverses the processing graph. The information in the metadata storage persists over the OBI service chain of a single packet. It can be encapsulated and sent over from one OBI to another, along with the processed packet, as described in Section 3.1.

The other key-value storage available for applications in the data plane is the *session storage*. This storage is attached to a *flow* and is valid as long as the flow is alive. It allows applications to pass processing data between packets of the same flow. This is useful when programming stateful NF applications such as Snort, which stores flow-level metadata information such as flow tags, gzip window data, and search state.

Frameworks such as OpenNF [18] can be used as-is to allow replication and migration of OBIs along with their stored data, to ensure correct behavior of applications in such cases.

4. IMPLEMENTATION

We have implemented the OpenBox framework in two parts: a software-based OBI, and an OpenBox controller. We provide a simple installation script that installs both on a Mininet VM [28]. All our code is available at [29].

4.1 Controller Implementation

Our controller is implemented in Java in about 7500 lines of code. It runs a REST server for communication with OBIs and for the management API. The controller exposes two main packages. The first package provides the basic structures defined in the protocol, such as processing blocks, data types, etc. The other package lets the developer define applications on top of the controller, register them, and handle events. It also allows sending requests such as *read requests* and *write requests*, which in turn invoke read and write handles, accordingly, in the data plane (as described in Section 3.2).

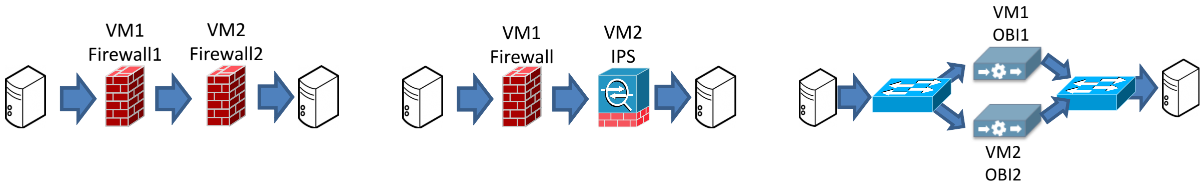
When an application sends a request, it provides the controller with callback functions that are called when a response arrives back at the controller. The controller handles multiplexing of requests and demultiplexing of responses.

Along with the controller implementation, we have implemented several sample applications such as a firewall/ACL, IPS, load balancer, and more. In addition, we implemented a traffic steering application as a plugin for the OpenDaylight OpenFlow controller [14]. We use it to steer the traffic between multiple OBIs.

4.2 Service Instance Implementation

Our OBI implementation is divided into a *generic wrapper* and an *execution engine*. The generic wrapper is written in Python in about 5500 lines of code. It handles communication with the controller (via a local REST server), storage and log servers, and translates protocol directives to the specific underlying execution engine.

The execution engine in our implementation is the Click modular router [23], along with an additional layer of communication with the wrapper and storage server, and several additional Click elements that are used to provide the processing blocks defined in the protocol (a single OpenBox block is usually implemented using multiple Click blocks). All our code for the execution engine is written as a Click user-level module, without any modification to the core code of Click. The code of this module is written in C++ and consists of about 2400 lines. Note that by changing the translation module in the wrapper, the underlying execution engine can be



(a) Two-firewall service chain (b) Firewall and IPS service chain (c) Test setup with OpenBox

Figure 7: Test setups under pipelined NF configuration.

replaced. This is necessary, for example, when using an execution engine implemented in hardware.

Finally, our OBI implementation supports custom module injection, as described in Section 3.2.1. An application developer who wishes to extend the OBI with new processing blocks should write a new Click module (in C++) that implements the underlying Click elements of these new blocks, and implement a translation object (in Python) that helps our wrapper translate new OpenBox block definitions to use the code provided in the new Click module.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Environment

Our experiments were performed on a testbed with the following machines: a traffic generator with an Intel Xeon E3-1270 v3 CPU and 32GB RAM, and a hypervisor with a Dual Intel Xeon E5-2690 v3 CPU and 384GB RAM. Both machines run Linux Ubuntu 14.04, with VMs running on the hypervisor using KVM. The machines are connected through a 10 Gbps network. All NFs and OBIs, as well as the OBC, run on top of the hypervisor. We play a packet trace captured from a campus wireless network on the traffic generator, at 10Gbps. All packets go through the hypervisor on their corresponding service chain as defined for each test.

5.2 Test Applications

For our tests we have implemented a set of sample OpenBox applications. For each of the following applications, we also created a reference stand-alone version, in Click.

Sample Firewall.

We use a ruleset of 4560 firewall rules from a large firewall vendor. Our OpenBox firewall application reads the rules from a file and generates a processing graph that raises an alert for packets that match any non-default rule. In order to correctly measure throughput, we have modified the rules so that packets are never dropped. Instead, all packets are transmitted untouched to the output interface.

Sample IPS.

We use Snort web rules to create a sample IPS that scans both headers and payloads of packets. If a packet matches a rule, an alert is sent to the controller. As in the firewall, we have modified the rules to avoid dropping packets.

Sample Web Cache.

Our web cache stores web pages of specific websites. If an HTTP request matches cached content, the web cache drops the request and returns the cached content to the sender. Otherwise, the packet continues untouched to the output interface. When measuring performance of service chains that include this NF, we only send packets that do not match cached content.

Sample Load Balancer.

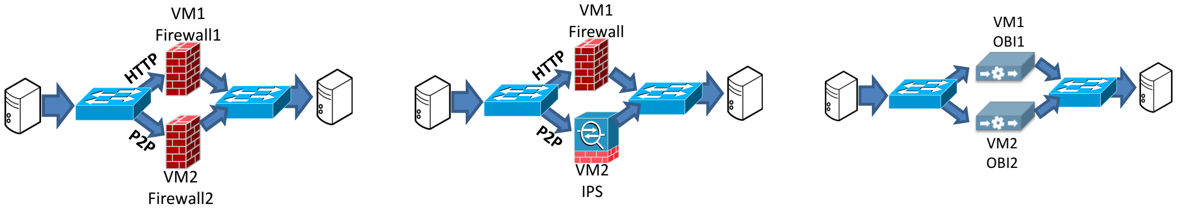
This NF uses Layer 3 classification rules to split traffic to multiple output interfaces.

5.3 Test Setup

We tested our implementation of the OpenBox framework using a single controller and several VMs. Each VM either runs Click with the reference standalone NF implementation, or an OBI that is controlled by our OBC.

We consider two different NF configurations in our tests. In the first configuration, packets from the traffic generator to the sink go through a pipeline of two NFs. The throughput in such a configuration is dominated by the throughput of the slowest NF in the pipeline. The latency is the total time spent while processing the two NFs in the pipeline.

Figures 7 illustrates two test setups under this configuration, without OpenBox: In the first test, packets go through two firewalls with distinct rule sets (Fig. 7(a)). In the second test, packets first go through a firewall and then through an IPS (Fig. 7(b)). With OpenBox (Fig. 7(c)), all corresponding NFs are executed on the same OBI, and the OBI is scaled to use the same two VMs used without OpenBox. In this case, traffic is multiplexed to the two OBIs by the network forwarding plane. We show that the OpenBox framework reduces the total latency (due to the merging of the two



(a) Two-firewall service chains

(b) A firewall service chain and an IPS service chain

(c) Test setup with OpenBox

Figure 8: Test setups under the distinct service chain configuration.

Network Functions	VMs Used	Throughput [Mbps]	Latency [μs]
Firewall	1	840	48
IPS	1	454	76
Regular FW+FW chain	2	840	96
OpenBox: FW+FW OBI	2	1600 (+90%)	48 (-50%)
Regular FW+IPS chain	2	454	124
OpenBox: FW+IPS OBI	2	846 (+86%)	80 (-35%)

Table 2: Performance results of the pipelined NFs configuration (Figure 7).

processing graphs) and increases the overall throughput (because of the OBI scaling).

Another NF configuration we consider is when packets of different flows go through different, distinct service chains, and thus visit different NFs. Under this configuration we test the following scenarios, as illustrated in Figure 8: in Figure 8(a) packets either go through Firewall 1 or through Firewall 2 while in Figure 8(b) packets either go through a firewall or through an IPS. We use the same rule sets as in the previous tests.

Merging the two NFs in this case provides dynamic load balancing by leveraging off-peak times of one NF to provide higher throughput to the other. We use the OBI setup as shown in Figure 8(c), this time only applying the processing graph of one NF on each packet, according to its type or flow information.

Note that in both configurations, each NF could come from a different tenant, or a different administrator. The different NFs are not aware of each other, but as discussed in Section 3, they may be executed in the same OBI.

5.4 Results

5.4.1 *Data Plane Performance*

Pipelined NFs.

Table 2 shows the results of the pipelined NF configuration. Without OpenBox, the throughput is bounded by the throughput of the slowest NF in the pipeline. Thus, in the two pipelined firewalls, the overall throughput is the throughput of a single firewall (both firewalls show the same performance as we split rules evenly). In the pipelined firewall and IPS service chain, the IPS dominates the overall throughput as it is much slower than the firewall, since it performs deep packet inspection. The overall latency is the sum of the latencies of both NFs in the chain, as packets should go through both VMs.

With OpenBox, the controller merges the two NFs into a single processing graph that is executed on OBIs on both VMs. Packets go through one of the VMs and are processed according to that processing graph. We use static forwarding rules to load-balance the two OBIs. The overall throughput is of the two OBIs combined. The overall latency is of a single OBI, as packets are only processed by one of the VMs.

OpenBox improves the throughput by 90% in the two-firewall setup and by 86% in the firewall and IPS setup. It reduces latency by 50% and 35% in these two setups, respectively.

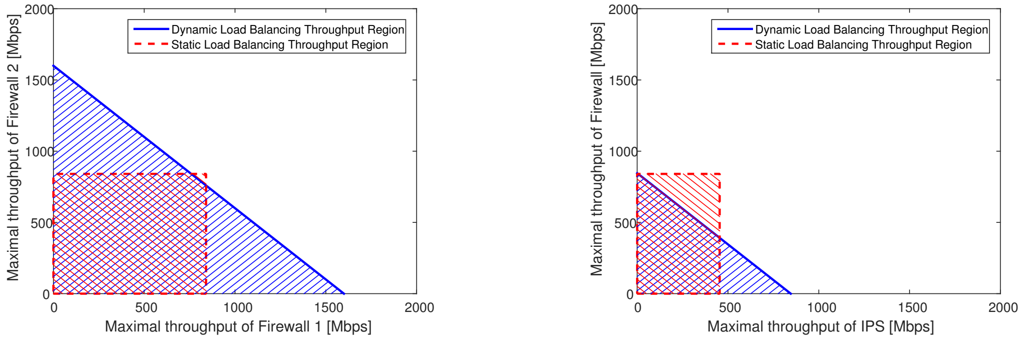
Distinct Service Chains.

Figure 9(a) shows the achievable throughput regions for the distinct service chain configuration with two firewalls, with and without OpenBox. Without OpenBox (see red, dashed lines), each firewall can utilize only the VM it runs on, and thus its throughput is limited to the maximal throughput it may have on a single VM. With OpenBox (see blue, solid line), each firewall can dynamically (and implicitly) scale, when the other NF is under-utilized. We note that if both NFs are likely to be fully utilized at the same time, merging them may not be worthwhile, but they can still be implemented with OpenBox and deployed in different OBIs.

Figure 9(b) shows the achievable throughput regions when merging a firewall with an IPS. In this case the IPS dominates OBI throughput and it might be less beneficial to merge the two service chains, unless the firewall is never fully utilized while the IPS is often over-utilized.

Discussion.

Two factors help OpenBox improve data plane performance. First, by merging the processing graphs and eliminating multiple classifiers, OpenBox reduces latency and the total computational load. Second, OpenBox allows more flexible NF deployment/replication than with monolithic NFs, so packets should traverse fewer VMs,



(a) Two Firewalls

(b) Firewall and IPS

Figure 9: Achievable throughput for the distinct service chain configuration (Figures 8(a) and 8(b)) compared to the achievable throughput of the two OBIs that merge both NFs (Figure 8(c)).

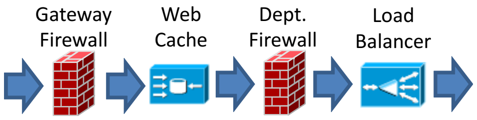


Figure 10: Service chain for the graph merge algorithm test.

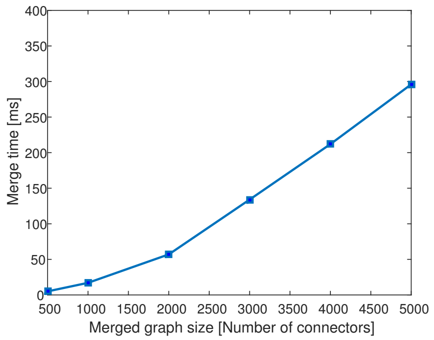


Figure 11: Scalability of the graph merge algorithm.

just as in the case when each NF is deployed separately. This flexible deployment also allows resource sharing.

5.4.2 Performance of the Graph Merge Algorithm

In order to evaluate the impact of our graph merge algorithm on the performance of the data plane, we considered a longer service chain, as illustrated in Figure 10. In this service chain packets go through a first firewall and then through a web cache. If not dropped, they continue to another firewall, and eventually go through an L3 load balancer.

We implemented this service chain in OpenBox by merging the four NFs into a single processing graph. When using a naïve merge, where graphs are simply concatenated to each other, we obtain 749 Mbps throughput (on a single VM, single core) for packets that do not match any rule that causes a drop or DPI. When using our graph merge algorithm, the throughput for the same packets is 890 Mbps (20% improvement).

Operation	Round Trip Time
SetProcessingGraph	1285 ms [4]
KeepAlive	20 ms
GlobalStats	25 ms
AddCustomModule	124 ms

Table 3: Average round-trip time for common messages between OBC and OBIs, running on the same physical machine.

Figure 11 evaluates the scalability of the graph merge algorithm. We tested the algorithm with growing sizes of input graphs on the Xeon E5-2690 CPU. The merge algorithm runs in orders of milliseconds, and the time grows nearly linearly with the size of graphs.

5.4.3 Control Plane Communication

In addition to data plane performance, we also evaluated the performance of the communication with the controller. Table 3 shows the round-trip time for three common protocol operations: SetProcessingGraph is the process of sending a SetProcessingGraphRequest from the OBC to an OBI with a new processing graph for the OBI, reconfiguring the execution engine and returning a SetProcessingGraphResponse from the OBI to the OBC. A KeepAlive message is a short message sent from an OBI to the OBC every interval, as defined by the OBC. GlobalStats is the process of sending a GlobalStatsRequest from the OBC to an OBI and returning a GlobalStatsResponse from the OBI to the OBC, with the OBI system load information (e.g., CPU and memory usage). AddCustomModule is the process of sending an AddCustomModuleRequest from the OBC to a supporting OBI with a custom binary module that ex-

[4] This operation involves (re-)configuration of Click elements, which requires polling the Click engine until all elements are updated. In Click, there is a hardcoded delay of 1000 ms in this polling. This can be easily reduced, albeit with a change in the core Click code.

tends the OBI behavior. In this test we used a module of size 22.3 KB, which adds support for a single processing block.

As these times were measured when the OBC and the OBI run on the same physical machine, they ignore network delays and mainly measure software delay.

6. DISCUSSION

This paper lays the foundations for a software-defined framework for NFs. In this section we discuss additional aspects of the proposed framework and possible future research directions.

Security.

Having multiple applications from possibly multiple tenants running on the same framework increases concerns about security and information leakage. Several previous works have addressed such concerns (e.g. [22, 25]) by creating privacy-preserving data structures for packet classifiers. Such works can be applied on the OpenBox framework as they are orthogonal to the OpenBox solution.

The declarative northbound API of our OBC lets applications specify their logic and listen to events. However, the API does not let applications query the logic of other applications, or the merged logic of specific OBIs. The OBC is responsible for safely demultiplexing events when these are signaled from the data plane. Each request from a specific application has its own ID and each response carries this ID to allow correct demultiplexing.

In the data plane, one application can reduce the performance of another application (possibly of a different tenant), especially if two applications are merged and executed on the same OBI. For example, a malicious application can cause an increased classification overhead by setting an overwhelming number of classification rules.

Custom module injection may also expose new threats to the data plane, especially in a multi-tenant environment. In such a case, we suggest that digital signatures be enforced on all injected modules (namely, verifying their security or origin before injecting them to the network).

Control Level Graph Optimization.

The OBC can provide optimization to user-defined processing graphs, in addition to that provided by the merge algorithm presented in Section 2.2.1. For example, it could reorder blocks or merge them, or even remove or replace blocks. The ability to split a processing graph between multiple OBIs can also be used to automatically optimize performance and for load balancing.

Debugging.

As with SDN applications, debugging of applications to be deployed on top of the control plane northbound API can be challenging. Works on SDN debugging such as [20] could serve as a basis for future research on OpenBox application debugging.

Application Programming Language and Verification.

Our current northbound API merely exposes the protocol primitives and Events API to the application developer, using a set of Java classes and interfaces. However, higher level abstractions for applications, such as dedicated programming languages or structures, could provide more convenient ways to program NFs on top of OpenBox. Such abstractions might simplify and enhance the merge process in the controller. Examples of such abstractions for SDN applications are Frenetic/Pyretic [13]. In addition, verification solutions such as [4] might be applied on OpenBox applications, with the required adaptations, to provide offline verification before deploying NFs.

7. RELATED WORK

In recent years, middleboxes and network functions have been major topics of interest. In this section we discuss and compare the state-of-the-art works that are directly related to this paper.

CoMb [38] focuses on consolidating multiple virtual middleboxes into a single physical data plane location, thus improving the performance of the network in the common case where not all the middleboxes have peak load at the same time. E2 [33] is a scheduling framework for composition of multiple virtual NFs. It targets a very specific hardware infrastructure, and manages both the servers on which NFs are running and the virtual network switches that interconnect them. Unlike OpenBox, CoMb and E2 only decompose NFs to provide I/O optimizations such as zero-copy and TCP reconstruction, but not to reuse core processing blocks such as classifiers and modifiers. Specifically, the CoMb paper has left for future research the exploration of the choice of an optimal set of reusable modules [38, Section 6.3]. We view our paper as another step forward in this direction.

xOMB [1] presents a specific software platform for running middleboxes on general purpose servers. However, it does not consolidate multiple applications to the same processing pipeline. ClickOS [26] is a runtime platform for virtual NFs based on the Click modular router [23] as the underlying packet processor. ClickOS provides I/O optimizations for NFs and reduced latency for packets that traverse multiple NFs in the same physical location. ClickOS does not have a network-wide centralized control, and it does not merge multiple NFs, but only chains them and optimizes their I/O.

Commercial solutions such as OpenStack [31], OpenMANO [30], OpNFV [32], and UNIFY [21] are focused on the orchestration problem. They all assume each NF is a monolithic VM, and try to improve scaling, placement, provisioning, and migration. Stratos [17] also provides a solution for NFV orchestration, including placement, scaling, provisioning, and traffic steering.

OpenNF [18] proposes a centralized control plane for sharing information between software NF applications, in cases of NF replication and migration. However, their work focuses only on the state sharing and on the forwarding problems that arise with replication and migration, so in a sense it is orthogonal to our work.

OpenState [5] and SNAP [3] are programming language for stateful SDN switches. OpenState makes it possible to apply finite automata rules to switches, rather than match-action rules only. SNAP takes a network-wide approach where programs are written for "one big switch" and the exact local policies are determined by the compiler. Both these works are focused on header-based processing, but such ideas could be useful to create programming languages on top of the OpenBox framework, as discussed in Section 6.

To the best of our knowledge, Slick [2] is the only work to identify the potential in core processing step reuse across multiple NFs. They present a framework with centralized control that lets NF applications be programmed on top of it, and use *Slick machines* in the data plane to realize the logic of these applications. The Slick framework is mostly focused on the placement problem, and the API it provides is much more limited than the OpenBox northbound API. Slick does not share its elements across multiple applications and the paper does not propose a general communication protocol between data plane units and their controller. Unlike our OBIs, Slick only support software data plane units; these units cannot be extended. This work complements ours as the solutions to the placement problems presented in [2] can be implemented in the OpenBox control plane.

Our preliminary workshop paper [7] on OpenBox described the proposed architecture but presented a very limited framework that uses a unified processing pipeline for merging multiple middleboxes. The proposed unified pipeline was very restrictive. In this paper we present a much more flexible NF programming model, including an algorithm to merge multiple applications given this flexible model.

Another work [8] suggested extracting the process of deep packet inspection (DPI) to an external network service. This work shows how performing DPI for multiple middleboxes at a single location could improve network performance. Still, middleboxes are assumed to remain monolithic units, with their DPI logic outsourced to an external service.

OpenBox allows easier adoption of hardware accelerators for packet processing. Very few works have addressed hardware acceleration in an NFV environment [27], and those that have focused on the hypervisor level [9,16]. Such ideas can be used in the OpenBox data plane by the OBIs, and thus provide additional hardware acceleration support.

The Click modular software router [23] is an extendable software package for programming network routers and packet processors. It has numerous modules for advanced routing and packet processing; additional modules can be added using the provided API. OpenBox generalizes the modular approach of Click to provide an network-wide framework for developing modular NFs. We use Click as the packet processing engine, as part of our software implementation for an OBI, described in Section 4.

Another related work in this context is the P4 programmable packet processor language [6]. 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). A P4 switch can be used as part of the OpenBox data plane, by translating the corresponding protocol directives to the P4 language.

8. CONCLUSIONS

This paper presents OpenBox, a software-defined framework for developing, deploying, and managing network-functions. OpenBox decouples the control plane of network-functions from their data plane, and allows reuse of data plane elements by multiple logical NFs. In addition to easier management, orchestration, provisioning and scale, it provides greater flexibility in terms of NF development and deployment, multi-tenancy support with complete tenant isolation, and improved data plane performance.

We have implemented OpenBox and shown that it is not only easy to deploy and to program but also improves network performance. We envision that frameworks such as OpenBox will pave the way for further advances in *network function virtualization* (NFV) with respect to NF programming, deployment, and easier management, while maintaining and improving performance. The flexible support for hardware accelerators for packet processing makes OpenBox even more appealing as today most NFV frameworks assume completely virtual environments and do not support any hardware accelerators [27].

Acknowledgments

We thank the reviewers of the SIGCOMM PC and our shepherd Vyas Sekar for their valuable comments on this paper. We also thank Pavel Lazar, Dan Shmidt, and Dana Klein, for their part in the implementation of the OpenBox framework. 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.

9. 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] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming Slick Network Functions. In *SOSR*, pages 14:1–14:13, 2015.

[3] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *SIGCOMM*, 2016.

[4] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: towards verifying controller programs in software-defined networks. In *PLDI*, page 31, 2014.

[5] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming platform-independent stateful OpenFlow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, Apr 2014.

[6] 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.

[7] A. Bremler-Barr, Y. Harchol, and D. Hay. OpenBox: Enabling Innovation in Middlebox Applications. In *HotMiddlebox*, pages 67–72, 2015.

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

[9] Z. Bronstein, E. Roch, J. Xia, and A. Molkho. Uniform handling and abstraction of NFV hardware accelerators. *IEEE Network*, 29(3):22–29, 2015.

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

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

[12] 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 *NSDI*, pages 533–546, 2014.

[13] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134, February 2013.

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

[15] 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.

[16] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu. OpenANFV: Accelerating network function virtualization with a consolidated framework in openstack. In *SIGCOMM*, pages 353–354, 2014.

[17] 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.

[18] 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.

[19] J. Gross, T. Sridhar, P. Garg, C. Wright, I. Ganga, P. Agarwal, K. Duda, D. Dutt, and J. Hudson. Geneve: Generic network virtualization encapsulation. IETF Internet-Draft, November 2015. https://tools.ietf.org/html/draft-ietf-nvo3-geneve-00.

[20] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, pages 71–85, 2014.

[21] W. John, C. Meirosu, B. Pechenot, P. Skoldstrom, P. Kreuger, and R. Steinert. Scalable Software Defined Monitoring for Service Provider DevOps. In *EWSDN*, pages 61–66, 2015.

[22] A. R. Khakpour and A. X. Liu. First step toward cloud-based firewalling. In *SRDS*, pages 41–50, 2012.

[23] 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.

[24] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network. IETF Internet-Draft, August 2014. https://tools.ietf.org/html/rfc7348.

[25] D. A. Maltz, J. Zhan, G. G. Xie, H. Zhang, G. Hjálmtýsson, A. G. Greenberg, and J. Rexford. Structure preserving anonymization of router configuration data. In *IMC*, pages 239–244, 2004.

[26] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *NSDI*, pages 459–473, 2014.

[27] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Comm. Surveys Tutorials*, 18(1):236–262, 2016.

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

[29] OpenBox Project Source Code. https://github.com/OpenBoxProject.

[30] OpenMANO. https://github.com/nfvlabs/openmano.

[31] OpenStack open source cloud computing software. https://www.openstack.org/.

[32] OpNFV. https://www.opnfv.org/.

[33] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: a framework for NFV applications. In *SOSP*, pages 121–136, 2015.

[34] P. Prakash, M. Lee, Y. C. Hu, R. R. Kompella, J. Wang, and S. Dassarma. Jumbo frames or not: That is the question! Technical Report 13-006, Purdue University, Twitter, 2013.

[35] Openbox framework specification, January 2016. http://www.deepness-lab.org/pubs/OpenBoxSpecification1.1.0.pdf.

[36] 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.

[37] 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.

[38] 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.

[39] J. Sherry and S. Ratnasamy. A survey of enterprise middlebox deployments. Technical Report UCB/EECS-2012-24, UC Berkeley, 2012.

[40] Snort users manual 2.9.7. http://manual.snort.org/.

[41] R. Stuhlmuller. Micro-Segmentation: VMware NSX's Killer Use Case, June 2014. https://blogs.vmware.com/networkvirtualization/2014/06/micro-segmentation-vmware-nsx.html.

[42] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, Sept. 2005.