Dynamic Service Chaining with (MB)

Xuan Kelvin Zou, Amy Tai, Ronaldo Ferreira, Jennifer Rexford, Pamela Zave

1. INTRODUCTION

Middleboxes are ubiquitous and elastic: Network administrators use middleboxes to apply services on the traffic exchanged between pairs of communicating endpoints. Today, they are routinely used to improve security (firewall, packet scrubber), protect user privacy (encryption, anonymization), share a set of IP addresses (network address translator), spread traffic over multiple backend servers (load balancing), reduce bandwidth consumption (compression, video transcoding, caching), monitor traffic and perform application-specific tasks. Recent trends of network function virtualization (NFV) implement middleboxes as virtual machines (VMs) or user-space applications separate from the physical host. Virtualization enables running network functions (NFs) at many different locations in the network or even in public clouds, and the NF instance can spin up (or down) or even migrate as load demands.

Endpoints move: The growing cellular network and WiFi hotspot coverage has pushed the term "mobile" to a new era. Mobility support has become a key consideration in terms of network infrastructure design. More and more network applications and services are run in virtualized environments; the server VMs can run anywhere and the VM migration can happen at any moment due to load balancing, infrastructure maintenance, etc.

To support a new Internet era where client mobility, and server and middlebox migration can take place anywhere and anytime, we propose a (MB) (definition goes here).

The existence of middleboxes already breaks the end-to-end principle, so we capitalize on this violation to create a new abstraction for a connection. We identify each connection by a supersession and a list of subsessions, where a supersession represents the original end-to-end communication. NF traffic steering simply stitches the subsessions together, and mobility and migration apply the same building block operation: replacing an old subsession with a new one. Hence, the subsession abstraction exists to isolate middlebox operations in a supersession to individual subsessions.

Although previous works consider some of the problems we address, none provides a comprehensive solution for endpoint mobility, server migration and middlebox service chain and migrations. TCP Migrate [40], Mobile IP [3], Serval [31] and ROAM [44] support endpoint mobility, but none of them explicitly deals with the existence of middleboxes. DoA [42] proposes a delegation architecture for middlebox service chain, however it neither supports flow migration across NF nor server migration. SIMPLE [32] and OpenNF [16] take advantage of modern switches that offer fine-grained control over routing (e.g., OpenFlow enabled switch), to steer traffic selectively through one or more middleboxes. However, the routing solutions fail to support endpoint mobility and suffer from scalability and flexibility of fine-grained policy control due to the limited size of ternary content-addressable memory (TCAM) on switches. More fundamentally, we argue middleboxes should be addressed explicitly instead of treated as second-class citizens.

(MB) achieves:

- **Unified solution:** It supports client mobility, server migration, middlebox service chain and flow migration during middlebox migration.
- **Incrementally deployable:** The system can be deployed between client and server, client and middleboxes, or even just between middleboxes.
- **Same socket abstraction:** It does not change transport layer, and thus does not require any change to the higher-level application if the application is using socket abstraction.
- **No routing change:** Since we address the subsessions explicitly by the network layer, simple solutions like shortest path routing still work.
- **High performance:** An in-kernel prototype shows that a shim translate layer can sustain up to 14.2 Gbps per CPU core.

The paper is organized as follows: Section 2 gives motivating examples of dynamic service chaining and

endhost mobility and how current solutions fail to meet the requirement. The architecture of (MB) is described in Section 3, and the protocol of supersession – subsession establishment and migration is described in Section 4. Section 5 describes crucial data plane properties provided by (MB) . Implementation is described in Section 7 and evaluation in Section 7. Section 8 presents a discussion about deployment, security and fault tolerance. We cover related work in Section 9 and conclude in Section 10.

2. MOTIVATING EXAMPLES

In this section, we first present a few scenarios where a system may require network function (NF) insertion, removal, or migration, and host mobility. We also discuss how the current solutions fail to address our requirements.

2.1 Dynamic NF Policy

Enterprise networks deploy various network functions for better performance and security. Being able to modify dynamically NF types/instances in a service chain improves the efficiency and flexibility of the network system. NF insertion is necessary if a flow is marked as suspicious by a coarse-grained intrusion detection and prevention system (IDPSs [37]) and requires a fine-grained deep packet inspection (DPI). NF removal is preferred if a connection goes through a cache proxy, but the proxy has a cache-miss and the content is uncacheable; the cache proxy may remove itself from the chain. NF load balancing is necessary if the network operators want to distribute a flow evenly among multiple instances of the same NF. Moreover, NF instance migration can happen in a network function virtualization (NFV) setting, and the flows that use the NF instance also need to migrate along with the instance.

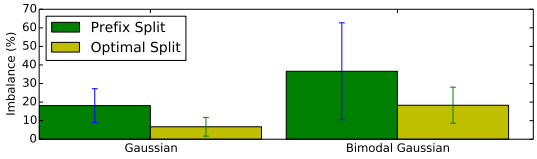

Figure 1: The distribution of flow size for different prefix can be measured using a weight[4]. We assume two different weight distributions for IP prefix: Gaussian, Bimodal Gaussian with the parameters from[4]. We show that in the case of distributing flow over two NF instances, a routing solution, while doubling rules, fails to balance the load across middleboxes in the scenario above. (Prefix split uses the most significant bit, and optimal split uses arbitrary one bit wildcard matching).

Efficient, dynamic NF policies cannot be implemented on conventional switches due to coarse-grained routing, e.g., the switch may simply tunnel all the traffic to IDS for further inspection and remove the IDS once it finds the proverbial "needle in the haystack". Some recent techniques leverage fine-grained routing switches (e.g., OpenFlow [26] based) for dynamically changing NF policies. However, these solutions are neither scalable due to TCAM rule size, nor flexible due to a complex dependency between routing rules. The SDN controller plays an excessive role in data plane, while still fails to fulfill certain network order-preserving properties [35, 16]. Figure 1 gives an example of the imbalance caused by routing solution when trying to distribute flows across two instances of the same NF type.

2.2 Mobility and NF Policy

A cellular network can be divided into user equipment (UE), local access network (LAN) and core network. LANs communicate to UEs through its base station and communicate to the Internet through the core network. Cellular networks rely on a wide range of NFs (e.g., firewall [23], load balancer [1], cache proxy [7]) to improve performance and enhance security. As mobility and network function virtualization become ubiquitous, we may ask for (i) seamless mobility and (ii) dynamic service chaining.

Many mobility solutions have been proposed over the past years [3, 40, 44, 31], yet none of them consider the existence of network functions. Worse, NFs can even be a hindrance for these protocols since they may interfere with the protocol control logic (e.g., TCP split [11]). Routing-based solutions have been proposed for service chaining in cellular networks(e.g., SoftCell [18]). However, in order to support host mobility, it places NFs only in the core network and does not support dynamic service chaining.

3. ARCHITECTURE

We now give an overview of (MB) architecture. We discuss the design goals that motivate a new architecture, which separates functionality into three different planes, architecture components, and the mechanisms to achieve the design goals.

3.1 Design Goals

Scalable policy management. One of the goals of our architecture is to ensure that traffic is correctly forwarded through a middlebox chain according to network policies. Relying on routing for traffic steering is inefficient for several reasons: distributed routing protocols are slow to converge during topology reconfiguration and provide only coarse-grained control over flows. SDN-based solutions avoid this problem by installing per-flow rules on network switches, but these do not scale well since the SDN controller must be involved on per-flow decisions, and switches have limited memory to store flow rules; in the worst case, each switch must install one rule per TCP flow.

In (MB) , we avoid both of these scalability imped-

iments. The protocol is designed such that the destination address of each packet identifies the next middlebox or endpoint in the path and the source address identifies the sending middlebox or endpoint. Doing so avoids the need of routing tweaks when network topology changes or endpoints move. Policy management in (MB) is controlled by a logically centralized controller, but packets on the data plane are never redirected to the policy server. This is a key difference from SDN solutions that queue packets on the controller during flow migration between middleboxes [16]. Moreover, the policy server can delegate per-flow decisions to the middleboxes. For instance, a middlebox may autonomously decide to remove itself from a session path. Delegation further relaxes any dependence on the policy server.

Low performance overhead. Recall that we identify each connection with a supersession and its associated subsessions. A simple approach for decomposing a supersession into subsessions can be implemented by establishing tunnels between middeboxes. However, in adding a new header to each packet, this approach introduces significant overhead from MTU increase and consequently packet fragmentation. Packet fragmentation can be solved by increasing the MTU of switches and routers inside a single network administration, but this solution is not viable when packets have to cross network domains, which has become a common case in virtualized network functions that are outsourced to public clouds. Moreover, solutions that rely on tunnels generally add extra overheads such as encryption and compression [39], features that might be redundant or unnecessary to all flows. (MB) relies on network address translation for explicitly addressing subsession endpoints, hence the only overhead is incurred by port remapping. Also, supersession IDs in (MB) are not carried on each packet, as in [42]. They are stored on the middlebox agents during supersession setup or supersession reconfiguration during middlebox migration.

Endpoint mobility and NF migration. The growing trend in network function virtualization can cause dynamic migration of flows between middleboxes. Along with frequent end host location changes due to device mobility or VM migration, these new possibilities complicate traffic-steering solutions relying on administrative control over network routing. In (MB) architecture, a separate policy server decides (i) the sequence of middleboxes traffic should traverse and (ii) informs the hosts–whether endpoints or middleboxes–of these decisions by pushing policies ahead of time to avoid increasing connection set-up delay. Then, the policy server steps back and allowing these hosts to establish sessions and handle migrations as needed.

3.2 Components

An overview of the (MB) architecture with its main components and the interactions between the modules is depicted in Figure 2. The architecture presents a clear separation of management, control, and data planes functionalities. The management plane is composed of a policy server that is responsible for coordinating the agents on the control plane based on high-level network policies provided by the network administrator. The control plane implements the protocol in §4 for session initiation, flow migration, and network function insertion and removal. The data plane is responsible for translating supersession IDs into local IP addresses and ports, delivering packets to the network functions, and forwarding packets between middleboxes. To simplify the presentation, we first assume a homogeneous scenario in which endpoints and all middleboxes run (MB) agents, and defer to §8 a discussion of a more realistic scenario where (MB) can be deployed incrementally. Also, we assume that network functions do not change the packet 5-tuple (i.e., source and destination IP addresses and port numbers, and protocol type), the case where packets are changed will be discussed in §6.2.

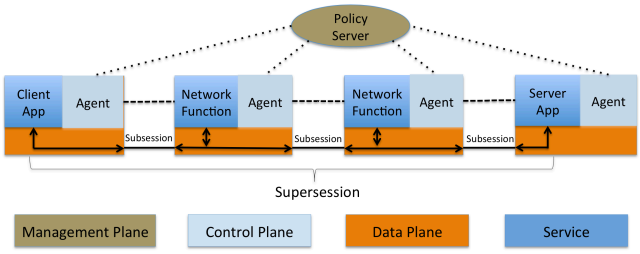

Figure 2: (MB) architecture

Policy server: The (MB) policy server is a logically centralized server that receives high-level policy specification from the network administrator and reliably delivers subpolicies to the agents, i.e., it makes sure that an agent receives a policy consistent with its location. For instance, a client agent should receive only policies related to the network it is connected.

The policy server makes global decisions based on: (i) configuration changes or (ii) network state changes. Configuration changes are triggered by the network administrator and network state changes are triggered by monitoring information. The policy server receives monitoring information from the (MB) agents and may decide to migrate flows from an overloaded network function to a different instance, insert new network functions in a supersession (e.g., a DPI may be dynamically introduced after a light IDS detects that packets from a supersession need deeper inspection), or remove a network function out of a supersession path because it is no longer needed. However, the policy server does not need to be involved in all fine-grained decisions, it can offload decisions to the (MB) agents runing on the middleboxes as determined by the network administrator.

A common decision that can be easily offloaded to the agents is the selection of a network function instance when the network administrator configures multiple instances of the same type.

In (MB) , policies define the service chains to be traversed by different packets or sessions (e.g., packets from subnet 10.0.1.0/24 should be forwarded through the chain {Load Balancer→IDS→Firewal→Proxy}). A policy specifies a predicate, which is used to restrict a set of packets, and a sequence of network function types. The general form of a policy is:

$$\texttt{match(predicate)} \texttt{ >> } \{\texttt{NFT}_1 \rightarrow \cdots \rightarrow \texttt{NFT}_n\}$$

A predicate is specified with source and destination IP addresses and ports, and protocol types. Complex predicates can be specified using conjunction (&), disjunction (|), and negation (~) operators like in Pyretic [27]. A packet that satisfies the predicate in the match statement is forwarded through the chain of network function types specified on the right-hand side of the >> operator. A network function type ($\texttt{NFT}_i$) specifies a set of network function instances of the same type. The network administrator can specify, for example, a set with the IP addresses of proxy servers and the policy server can choose an instance for sessions initiated in a specific client or it can delegate this decision to the (MB) agents.

(MB) Agents: (MB) agents are key components to enforce network policies. They must ensure that packets are correctly forwarded to the next middlebox on the service chain or dropped if they do not comply with the network policy. Whenever a client application starts a connection, the (MB) agent running on the same machine intercepts the first packet of the connection and matches it with the client's network policies to determine the sequence of network function types the packet must be forwarded. If a match is found, the agent starts the session setup protocol in §4.1 that will create the subsessions between middleboxes and the corresponding mappings between incoming and outgoing packets in each middlebox on the session path. The 5-tuple of the client connection is used to identify the supersession packets when they are processed by the network functions. Once the supersession is setup, all subsequent packets from the client's connection are forwarded to the first middlebox on the service chain.

In each middlebox, a (MB) agent must create two mappings for each session. One, which we call horizontal NAT, translates the 5-tuple of an incoming packet to the corresponding 5-tuple that is used in the next subsession. The other one, which we call vertical NAT, translates the 5-tuple of each incoming packet to the supersession 5-tuple before the packet is delivered to the network function or to the applications on the end-hosts. Keeping the supersession 5-tuple invariant on the middleboxes has a few advantages. First, it simplifies policy specification, because the network administrator can design policies based only on the endpoint addresses without worrying about the subsessions. Second, it simplifies network function state migration, because network functions always receive packets with the supersession 5-tuple regardless of flow migrations or middlebox insertions or removals. Third, network functions that do not change the packet 5-tuple do not need to be changed to work with (MB) .

(MB) agents are also heavily involved on service chain maintenance and supersession reconfigurations. The agents report management information, such as resource utilization in the middleboxes, to the policy server, so it can take global management decisions about service chains. Reconfigurations can also be triggered by a network function, if allowed by the network policy. A common case of a reconfiguration triggered by a network function is when a cache proxy detects that the content of a session is not cacheable and signals the (MB) agent to remove the proxy from the service chain. Once a reconfiguration is initiated, either by the policy server or by the network function, (MB) agents execute the protocols in §4.2.

4. PROTOCOL

Considering middleboxes as explicit components of the end-to-end between two endpoints is the crux of our protocol. Only by doing so can we achieve the desired scalability and flexibility for both endpoints and middleboxes. We discuss session setup in §4.1 and flow migration control in §4.2.

4.1 Session Setup

In (MB) protocol, each endpoint or middlebox sends packets whose destination is the next middlebox or endpoint in the session path. This obviates the need for special support in the switch or router to direct packets through the chosen chain of network functions (service chain), despite changes in network topology or host movement.

The list of middleboxes, L, that a flow has to traverse is provided by the policy server and can be pulled from the server or pushed to the client. When the client initiates the connection, the control plane uses a three-way handshake to establish the supersession and its associated subsessions. More specifically, the client's control plane sends a SYN message to the first middlebox that includes the supersession header and L. The middlebox strips itself from the head of L, gets the address of the next middlebox from L, and relays the rest of the message to the next middlebox. The SYN message is thus passed recursively through the elements of L before reaching the server. Upon receiving the SYN, the server sends a SYNACK back to the client using $reverse(L)$. Upon receiving the SYNACK, the client immediately

sends an ACK to the server via the same mechanism. Once these three control messages are exchanged, the supersession and subsessions are established, and data packets are explicitly addressed to the subsession IPs.

If we simply rewrite the source and destination IPs, we lose supersession information and introduce ambiguity. Consider the case where flow a and b have the same source port and destination IP and port, but different source IPs. If a and b share the same first hop middlebox, the two flows may become indistinguishable upon arrival at the first hop middlebox. To address this issue, we modify the port numbers to identify the flow, a standard technique in NAT [13]. We integrate this port allocation into the three-way handshake: port mappings are assigned in a middlebox when it receives a SYN, and initiates a new subsession with the rewritten port numbers. If we rewrite both source and destination ports, (MB) can support four billion unique flows per middlebox pair. See Figure 3 for the complete session setup.

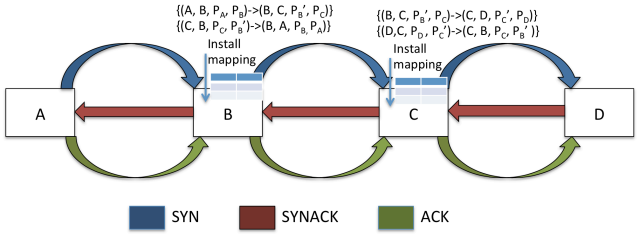

Figure 3: Session setup

4.2 Migration and Mobility Control

Supporting dynamic middlebox modification for a flow improves the efficiency of the network and NF use, e.g. — removing a cache proxy if the content is not cacheable, inserting an IDS upon detecting suspicious flows, or switching from a heavily loaded transcoder to a lightly loaded one. As a natural extension of general middlebox migration, we also include endhost mobility in the design of (MB) ; supporting mobility is also critical in cellular networks.

4.2.1 *"Make before break"*

To find the right mechanism to support flow migration, we investigated existing mobility protocols [40, 44, 3, 31, 29, 30] under the session-location mobility framework [43]. However, the key distinction between host mobility and NF flow migration is that a move in host mobility is unexpected, whereas flow migration among middleboxes is planned (or controlled?).

In fact, this type of reconfiguration is quite common in circuit design and addressed via the "make before break" philosophy. Namely, we stitch together subsessions on the new path before closing the subsessions on the old path. To achieve this, we treat the two neighbors of the moving middlebox as two *signaling endpoints* during a flow migration. We stress that the resulting three nodes that are involved in the migration are *consecutive*. Because we offload some policy decisions to the middleboxes, this property ensures that a middlebox cannot decide the fate of other on-path middleboxes that are not directly affected by the migration.

For a middlebox insertion, we first identify and notify one of the two *signaling endpoints* as an initiating point in a deterministic way. We start from the initiating point, set the two *signaling endpoints* in a suspend state for data transfer for the new path, and complete a three-way handshake (UPDATE-SYN, UPDATE-SYNACK and UPDATE-ACK) on a new path that includes the two *signaling endpoints* and the new middlebox to insert, to ensure the path establishment. Once the new path is established, we start the data transfer on the new path and remove the old path. We extend the same mechanism to handle middlebox removal and replacement.

4.2.2 *Handling Concurrent Migration*

When we offload dynamic network function policy to a distributed control plane, we expect cases where two middleboxes initiate simultaneous move operations. In particular, if two neighbor middleboxes both initiate removal, we may lose the supersession connection. Strawman solutions include: (1) using a two phase commit to allow the one with the highest ID to move first; (2) a central controller that assigns token to the middleboxes. However, strong serialization via a two phase commit is not necessary, and a central controller suffers from scalability, thereby deviating from the initial design goal. We should furthermore allow concurrent moves at different points of the service chain.

To address concurrent migration, we rely on the following properties: (i) migration is per flow, and (ii) at most three nodes on the current service chain are involved in each flow migration. The two assumptions help us design an efficient algorithm: migration can happen simultaneously for different flows, and concurrent migrations can happen at different points of the service chain if they involve disjoint sets of nodes; each node has a `pending` counter to ensure that it participates in at most one concurrent update.

Middleboxes for a flow have a strict ordering, which is simply the order in which the middleboxes are traversed by the path from the client to the server. We can define two comparators on this ordering, which we term "left" and "right". M_1 is left of M_2 iff the path from the client to M_2 passes through M_1; M_1 is right of M_2 iff the path from the client to M_1 passes through M_2. The proposed algorithm only allows one concurrent operation for all the nodes involved. When a migration is initiated at a certain node, the node postpones the

update if its `pending` counter is not zero, otherwise it increments the `pending` counter and sends the request to the node immediately to its left if the migration is removal or replacement, or connects to the new middlebox if it is an insertion. If the left side responds with a reject, the node backs off, otherwise it receives an approval and proceeds with the update. A node always accepts UPDATE-SYN requests (establishing a new path). Algorithm 1 describes the algorithm details and Figure 4 depicts the steps for migrations.

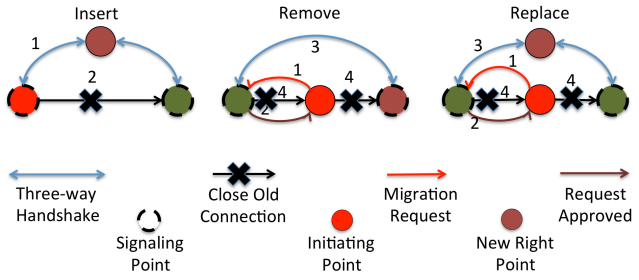

Figure 4: Flow migration during insert, remove and replace

4.2.3 *"Break before make"*

When a client moves, it may drop the old subsession before establishing a new subsession. Consider when a UE moves across a cell boundary, upon which the UE may suffer from transient connection loss, since it is out of the old cell's range.

After losing the old subsession, the client needs to rebind to the first hop middlebox. If we use the client's physical IP as part of the supersession identification, the first hop middlebox will fail to identify the supersession if the client changes its IP during mobility. To solve this problem, we can either put the old connection's information in the rebinding message sent by the client or, in a single domain case, administrators can assign a non-routeable IP to each device as a unique ID and use this ID to help identify the supersession.

5. DATA PLANE PROPERTIES

In this section, we discuss three different data plane properties and mechanisms to support them atop the control logic.

5.1 Loss-Free Update

At the data plane, there is a translation table stored in each middlebox. The (MB) agent accepts flows for which it has established state, rewrites the header, and forwards the flow based on the supersession-subsession mapping. Moving a flow from one path to another is equivalent to updating the translation tables at the data plane to accept a flow from the new subsession(s) and reject the same flow from the old subsessions. Since there are multiple middleboxes involved in the update

Algorithm 1: Concurrent Flow Migration

```
Function Trigger_Migration()
    if recv(migrate) then
        if pending == 0 then
            pending++;
            if migration == insert then
                sendto(New right, UPDATE-SYN);
            else
                sendto(Left, request);
        else
            Exponentially backoff and retry;
Function Msg_Handler ()
    if recv(request) then
        if pending> 0 then
            sendback(reject);
        else
            pending++ ;
            sendto(New right, UPDATE-SYN);
            sendback(approve);
    if recv(reject) then
        Exponentially backoff and retry sendto(Left, request);
    if recv(approve) then
        //do nothing, to avoid request re-transmission
    if recv(UPDATE-SYN) then
        pending++;
        if (migration==insert or replace) and (current ==
        not signaling point) then
            forward(UPDATE-SYN);
        else
            sendback(UPDATE-SYNACK);
    if recv(close) then
        //clean old flow state;
        pending−−;
    if recv(UPDATE-SYNACK) then
        pending−−;
        if (migration==insert or replace) and (current ==
        not signaling point) then
            forward(UPDATE-SYNACK);
        else
            sendto(OldMBox, close);
            sendback(UPDATE-ACK);
            //clean old flow state
    if recv(UPDATE-ACK) then
        pending−−;
        if (migration==insert or replace) and (current ==
        not signaling point) then
            forward(UPDATE-ACK);
        else
            //clean old flow state
```

procedure, the update order matters. Otherwise, the translation layer may drop packets if the update happens in the wrong order, for example if the old subsession is removed before the new subsession is fully established.

Finding the right sequence of updates for a general network is proven to be NP-complete [17, 21], but for a special topology, linear in our setting (we only abstract the topology between middleboxes), we can apply the concept from network consistent update [36, 22]. We first ensure that the new translation rules are all pushed before the egress applies the new rule, and the old rules are not removed until all the new rules are installed via a complete handshake. In particular, when a migration is initialized from the middlebox (a signaling point), the middlebox notifies its neighbor through the control plane, which inserts a new rule for incoming traffic. Then this neighbor notifies the other side of the connection with an UPDATE-SYN control mes-

sage. Every hop that receives UPDATE-SYN updates its own translation table. Hence once the other signaling endpoint receives the notification, a new rule for the flow has been installed at every hop for one direction of traffic; it is thus safe to apply the new rule for the egress. The opposite direction is set up via the same way with UPDATE-SYNACK messages. Once the new bidirectional path is built, we tear down the old path by removing the old rules. This loss-free update mechanism mirrors the control plane "make before break" philosophy and is in fact facilitated by the control plane design.

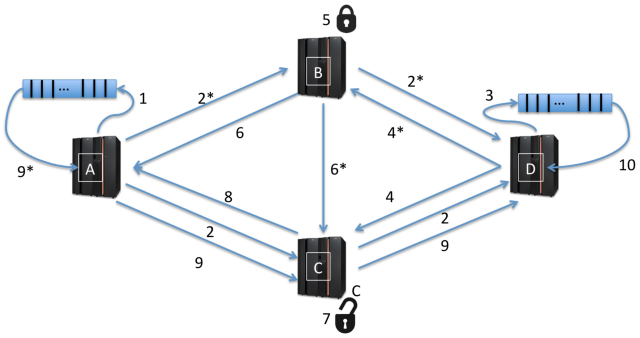

Figure 5: Flow Migration

5.2 Packet Order Preserving

The previously described loss-free update is insufficient if the NF instance (state) also needs to migrate. More specifically, the NF state cannot be migrated since it is being continuously updated as packets are coming in via the old path. To lock and migrate the middlebox state, (MB) must stop sending traffic during the new path setup. Since changing the protocols' (e.g., TCP) flow control is undesirable, we choose to buffer the traffic and not to release it until the network function state on the old path's middlebox has been locked and replicated to the new path's middlebox. Since NF state replication and migration is a well solved problem [16, 35, 34], we do not address this problem here.

The migration takes 10 steps[1] as depicted in Figure 5: 1. lock outgoing traffic; 2. send UPDATE-SYN via new path; 2*. send UPDATE-FIN via old path; 3. lock the reverse direction traffic; 4. send UPDATE-SYNACK via new path; 4*. send UPDATE-FINACK via old path; 5. lock middlebox states; 6. send UPDATE-FINACK via old path; 6*. migrate states; 7. unlock middlebox states; 8. send UPDATE-SYNACK via new path; 9*. release buffered traffic; 9. send ACK packets; 10. release buffered traffic for the other direction.

The update mechanism above is also order-preserving under the same assumption in OpenNF: the path to NF

[1] x and $x*$ can happen in parallel where x is a number.

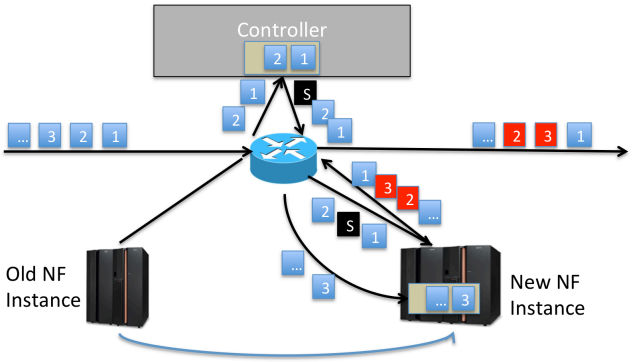

Figure 6: Order-preserving problem in OpenNF without FIFO assumption, the signal packet s and the data packets 1 and 2 maybe reordered and thus create reordering. Note the switch is one big switch abstraction.

instance is **FIFO**, i.e., the notification message sent after the last data packet is always received after the last data packet, whereas the **Order-Preserving** property is defined as the following: *All packets should be processed in the order they were forwarded to the NF instance by the switch (network)* [16].

5.3 Substream Separation and Order Preserving

Creating a separation of two substreams for a byte stream (e.g., TCP) is beneficial to avoid false alert from NFs and/or improve the NF efficiency. When we plan to migrate a flow from one IDS to another, we may want to ensure that all the SYNs and corresponding ACKs are going through the same IDS, otherwise, it may raise an alert like "ACK before SYN". When the flow passes through a deep packet inspection (DPI), both string matching and reg-exp matching builds a Deterministic Finite Automaton (DFA) first, and traverses from the root to certain state based on the byte stream [10, 12]. If we want to use different DPI instances for a byte stream, and the first instance has not seen a complete substream, it can only check the largest continuous byte stream, buffer the remaining bytes and send to the second instance.

To lock the traffic and separate the substream, the left neighbor of the moving point use the maximum seen sequence number as a *checkpoint*, and buffers the packets with a higher sequence number than the checkpoint and forwards the packets with a lower sequence number. We do not lock and migrate the old NF instance state until seeing an ACK at sequence number *checkpoint*. This guarantees the delivery of the packets at the old path since ACK is sent from an endpoint. Note this may create a reordering from the perspective of the left neighbor middlebox, however this is not a reordering from the perspective of endpoints. See Algorithm 2 for details.

We may need to split the packets in the case of *SYN-ACK piggyback and packet coalescing*: In the first case, if both directions ask for substream separation, we may get in a deadlock since (MB) may choose the new path for a substream with higher sequence number in one direction and the old path for ACK as it is acking a substream with lower sequence number in the reversed direction; in the second case, TCP client may coalesce packets and the payload may cross the *checkpoint* boundary in the byte stream if retransmission occurs.

Algorithm 2: Substream Separation and Order Preserving

```
Event_Handler recv_SYN(TCP_packet p)
    checkpoint = hash_lookup(p);
    if p.seq > checkpoint then
        Buffer (p);
    else
        Forward(p);
Event_Handler recv_ACK(TCP_packet p)
    if p.ack > checkpoint then
        Migrate NF state;
        //wait until migration finishes;
        sendto(left neighbor, release_buffer);
Event_Handler release_queue()
    while !buffer.empty() do
        Forward(buffer.dequeue());
    Reset(hash_table);
```

A combination of order preserving and substream separation provides us with a stronger order preserving during an update: *Packets should be processed by different NF instances in the order they were sent from the sender*. OpenNF cannot guarantee order preserving, when certain network links are not FIFO [2], nor can OpenNF be extended due to its design choice (i.e., routing based), as shown in Figure 6 . Our protocol is able to provide this property precisely because the system architecture is designed to be aware of and rely on transport protocols.

6. IMPLEMENTATION

6.1 Prototype

We have an in-kernel data plane and user space control/management plane via TCP/UDP sockets, with a total of about 4000 lines of code in C and C++. There are three different options for in-kernel implementation of the data plane: (i) OS native network stack; (ii) NIC driver (e.g., DPDK [2]); and (iii) customized in-kernel software switch (e.g., openvswitch [20]). We choose option (i) due to (a) (MB) has to choose a right interface before sending to the driver when having multiple interfaces; and (b) customized in-kernel software switch has high overhead. We use kernel modules to register callback functions with Linux `netfilter` for data plane operation. The control/management plane has an extensible complex control logic, and the data plane does specific simple actions (e.g., header rewriting, queuing to packets user space and update kernel hash table). The user and the kernel agent communicate via `netlink`, a native Linux Inter Process Communication (IPC) function. Currently the policy server is a simple TCP server that proactively pushes the policies to the agents.

[2]OpenNF tech report relies on this assumption in its proof.

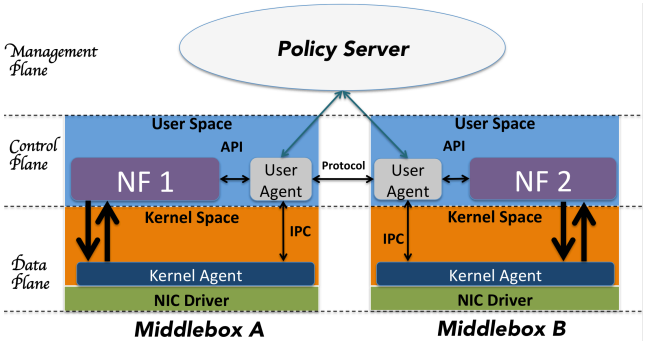

Figure 7: Layout of the implementation blocks

6.2 Network Function Support

We categorize NFs into two types — active and passive functions — based on whether the NF acts on the original packet. Table 1 lists several commonly used NFs with the types, key functions to fetch session information and binding libraries.

Many passive NFs get a clone of the original packet and decide based on the copy. As described in §3, we restore the supersession header of the copy before delivering the packets to NF, so called vertical NAT. Based on Table 1, supersession restoring can be realized by extending libpcap [25] for a wide range of NFs.

For active NFs, if they only act on the payload, supersession restoring also works. However, if the network function acts on the packet 5-tuple, we have to modify NF functions to notify the (MB) agent its own mapping. For example, a transparent cache proxy [7] gets the packets from its listening port and sets up a new TCP socket to the final destination. (MB) will break the supersession into two if it is unware of the mapping between two sessions. On the other hand, if the NF informs the (MB) agent the mapping between its listening and sending TCP sockets, the (MB) agent can stitch the two together "subsessions" into one supersession.

7. EVALUATION

In our evaluation, we demonstrate that:

- The system can sustain very high throughput
- The system has extremely low latency for flow migrations

Name	Type	Key Functions	Binding Library
PRADS [5] (P)	Monitoring	got_packet()	libpcap
Bro [8] (P)	IDS	DumpPacket()	libpcap
Snort [6] (P)	IDS	PQ_Show()	libpcap
Balance [1] (A)	Load Balancer	recv(), writen()	user socket
Squid [7] (A)	Proxy	getsockopt()	user socket
Traffic Squeezer [9] (A)	WAN-Optimizer	net_receive_skb()	Linux skbuff

Table 1: Popular Network Functions; (A) means active, (P) means passive NF.

- The system is resilient to lossy network and update failure.

The testbed we used for evaluation consists of two sets of machines: (i) four mid-range workstations (Quad-core Intel Xeon 3.7GHz, 32GB, 1Gbps NIC) and a conventional switch; and (ii) four high-end servers (16-core Intel Xeon 2.2GHz, 64GB, 40Gbps NIC) with an OpenFlow-enabled switch (we only use L-2 switching here). We conducted throughput stress tests in the four high-end servers, and unless specified, experiments are done in setting (i).

7.1 Throughput

We show that an in-kernel implementation of the shim layer has very low overhead in the overall system.

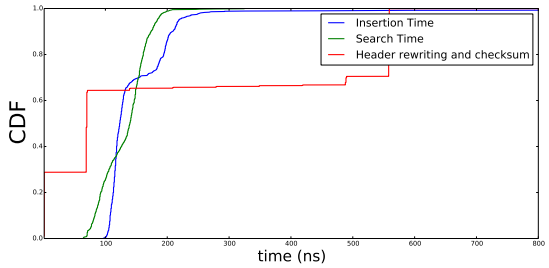

Figure 8: Time CDF for different functions

The in-kernel implementation of the data plane includes three functions: (i) action hash lookup, (ii) header rewriting and re-checksum, (iii) rule installation. We also have a few optimizations in the data-plane implementation (e.g., partial checksum to reduce CPU cycle, a small hash table to fit in L3 cache) result in an extremely efficient system with a very low overhead.

We conducted a microbenchmark to see the delay different functions add to the system. We insert 100K rules in the kernel hash table, and conducted 100K lookup. We also microbenchmarked the header rewriting and partial checksum of the data packets. We had the two following methods to eliminate Linux timer's overhead: (i) batch and time the insertion and lookup for every one hundred rules; (ii) get the pure timer lookup time and then subtract that base number. The result shows that lookup, insertion and header rewriting takes on average 131, 251, and 214 ns, this gives us $\approx$ 2.8M packets per seconds with lookup and header rewriting, this is equivalent to 33.6 Gbps per core with MTU of 1500 Bytes.

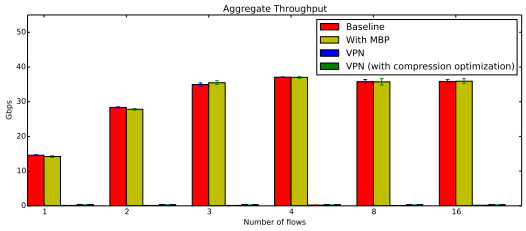

Figure 9: Throughput of different approaches. The throughputs are averaged over 20 samples and the error bars show 95% confidence intervals. Note each flow is hashed to one queue and thus binded to one core during CPU interrupts.

To test the throughput, we run iperf [41] with MTU of 1500 Bytes on three high-end machines with one **40 Gbps** port. The topology is client-to-middlebox-to-server, and we use a simple three-hop routing as baseline. We also changed the NIC's ring buffer hash function to an XOR hash function to avoid core interrupt collision, since the default hash function is designed for large number of flows, we see noticeable collision, i.e., hash two flows to the same ring buffer when the number is small.

The in-kernel data plane implementation can sustain **14.2 Gbps** for a single core, and scale near-linearly to the number of cores, and reaches **37.1 Gbps** at its peak. (The gap between this and 40 Gbps is due to the way how the bandwidth is calculated, this only computes payload $\leq$ 1460 Bytes versus ethernet frame 1518 Bytes).

To put in perspective, the baseline without running our kernel module can support **14.6 Gbps**, so the kernel module's overhead is under **3%**. The gap is closing as we increase the number of flows, since it incurs less frequent interrupts per core (the frequency of per-core interrupt for 4 flows with 9 Gbps per flow is 66% of a single flow with 14.5 Gbps), and thus the stress per core is lighter. We envision if we increase the NIC bandwidth to 100 Gbps or higher, the gap will stay the same and the throughput will grow near-linearly before the PCI becomes the bottleneck.

One interesting observation is that when there are three flows, the system offers a higher throughput than the baseline. The reason behind it is that the "extra work" tends to keep the cores busy and thus gets more CPU cycles for processing. The competing flows from TCP congestion control may also affect the throughput.

We compare our approach with off-the-shelf VPN-encapsulation mechanism, which is used in APLOMB [39] system to achieve its redirection. Not surprisingly, VPN

uses encryption and thus offers much lower throughput. OpenVPN [15], which is used in APLOMB system can only achieve <**400 Mbps** for a single thread, even with the compression optimization. Current OpenVPN does not support multi-threading, even if OpenVPN supports multi-threading in the future and assume it scales linearly to 16 cores, it can only offer 17% of the max throughput.

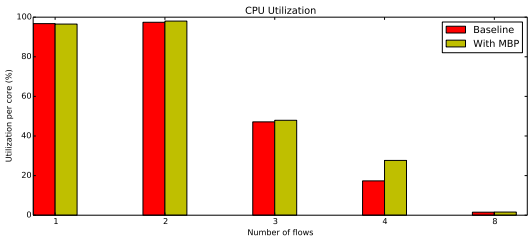

Figure 10: Average CPU utilization in stress test. Note we only include the cores that are interrupted by the NIC.

We plotted the CPU utilization of the middlebox in the same experiments. Note since the middlebox acts as a software router without the kernel module, it also consumes a certain amount of CPU cycles. The highest difference between the baseline and the system lies in the setting of 4 flows, the utilization is 10% higher than that of the vanilla system (27.6% versus 17.2%), but under most circumstances the gap is within 2%. One thing worth noting is that as the number of flow increases, the CPU utilization per core plummets, our explanation is that since the testbed machine has 16 cores and 32 hyper-threads, it spreads out quite evenly, and the CPU utilization scales linearly at high load but it is not the case at low load (e.g., 1M packets/s causes much higher CPU utilization than 500K packets/s).

The experimental result is promising and proves that the in-kernel data plane implementation can offer very high throughput at the cost of modest increase of CPU utilization.

7.2 Latency

We evaluate our system's latency in the following order: (i) how much overhead it has for a super-session setup? (ii) how much overhead it has during loss-free, order-preserving, and (some extra property) update?

7.2.1 Three-Way handshake

We first measure the latency for session establishment, in a topology of three hops (client - middlebox - server). We have two different implementation, (i) out of band UDP ahead to TCP handshake, (ii) piggyback TCP handshake at the payload. Not surprisingly, TCP piggyback can greatly reduce the extra latency due to propagation delay, with a modest increase less than 40μs. On the other hand, out-of-band UDP incurs 688μs delay ahead of TCP handshake.

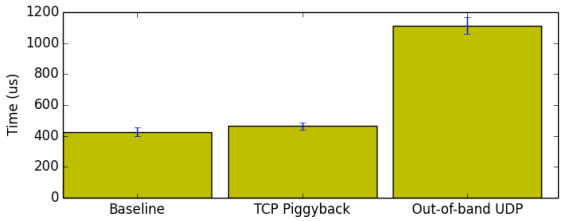

Figure 11: Latency for session establishment

7.2.2 Flow Migration

One big advantage for network function virtualization is its resource elasticity [16, 35, 24]: since network functions can run on general hardware, we can spin up or spin down a middlebox VM quickly and migrate flow and/or middlebox states cross middleboxes. In our evaluation, we focus on the latency of flow migration, in particular, how long it takes to move the subsession of a flow.

We run our experiments in a topology with four middleboxes (A,B,C,D), there are two paths (A-B-D) and (A-C-D) that each consists of three middleboxes. The goal of the flow migration is to replace middlebox B with D. We measure the latency for such migration, using (a) loss-free migration, (b) order-preserving migration, and (c) (a different property). The latency are evaluated under different loads on a network with all 1Gbps links.

As shown in the table, we can see that not surprisingly, as the load of the network increases, the update latency increases. We attribute the increase to two causes, the queuing delay in the network and the scheduling delay in local servers. In loss-free migration, the old path is saturated; however, the new path is idle and thus the increase is due to the high load in server. In order-preserving migration, the update is on hold until both the new and old paths have sent notification, and thus the queuing delay also contributes to the latency.

Load	Loss-Free	Order-Preserving
0%	1.7ms	2.3ms
50%	2.1ms	2.6ms
100%	3.6ms	14ms

Table 2: Flow migration update time under different network load

The results demonstrate that the system can deal with flow migrations in the order of tens of milliseconds. To put in perspective, the time for a middlebox state migration is above 100ms [35, 16] and a fast consistent update takes up to a second [19], a more detailed breakdown is in table 3.

[3] at packet rate 37.5 kpps, time are consumed across flow migration and state migration

[4] not supported

[5] flow update time at packet rate 2.5-10 kpps, (a) is for flow migration time, and (b) is for middlebox state migration time

	Loss-Free	Order-Preserving
Split/Merge [3]	500ms	⊗[4]
OpenNF(a) [5]	84ms	96ms
OpenNF(b)	134ms	208ms
Dionysus [6]	600-1200ms	⊗
Our system [7]	4ms	14ms

Table 3: Comparison with other dynamic middlebox and flow migration systems

8. DISCUSSION

incremental deployment
security
fault tolerance

9. RELATED WORK

Middlebox in SDN: SIMPLE [32] and FlowTags [14] take advantage of the switches with a fine-grained rule support in software-defined network (SDN), and support network function policy chaining in traditional and NFV setting respectively. Both approaches are constrained by the TCAM size, a hardware limitation in terms of fine-grained policy, and neither not support NF migration or host mobility.

OpenNF [16] and Split- Merge [35] leverage the SDN controller to manage NF's state migration and NF's flow migration. However, since the central controller is involved in both control plane (update the network rules) and data plane (buffer the packets on the fly during migration), they suffer from high latency and low scalability. They also suffer from hardware limitation for fine-grained policies.

Middlebox using naming service: DoA [42] uses a delegation-oriented global naming space architecture, that explicitly specifies intermediary middleboxes on the path. Two key distinction between DoA and (MB) is that (i) (MB) does not require any new naming service and (ii) DoA does not support dynamic policy service. APLOMB [39] outsources the network functions to the cloud with a naming service indirection. It uses VPN to tunnel the traffic to the cloud and use DNS-based indirection to decide which cloud to enforce the middlebox policy. It assumes the cloud can provide elastic NF service but it cannot explicitly handle dynamic policy chain.

Middlebox consolidation: CoMB [38] and click [24, 28] both consolidate network functions as an application or a VM image, and one host machine can run multiple instances of different network functions. Both approaches are mainly focused a feasibility and scalability of network functions on a single generalized servers. Both solutions consider neither scale-out across servers nor NF and flow migration.

Mobility Protocols: Mobility protocols use (i) a fixed indirection point (e.g., Mobile IP [3]), (ii) redirecting through DNS (e.g., TCP Migrate [40]), (iii) indirection infrastructure (e.g., ROAM [44]) or (iv) indirection at the link layer (e.g., cellular mobility). None of them consider the existence of middleboxes. Coexistence of network functions and new protocols is especially important for deployment, as a study in multipath TCP shows [33]. (MB) shares some similarity with many mobility protocols and has support for network functions.

[6] Large scale networks
[7] at packet rate 75 kpps (900 Mbps)

10. REFERENCES

[1] Balance. http://www.inlab.de/balance.html.
[2] DPDK. http://dpdk.org/.
[3] Mobile IP. https://tools.ietf.org/html/rfc3220.
[4] Niagara. Tech. rep. http://www.cs.princeton.edu/research/techreps/TR-973-14.
[5] PRADS. http://gamelinux.github.io/prads/.
[6] Snort. https://www.snort.org/.
[7] squid. http://www.squid-cache.org/Intro/.
[8] The Bro Network Security Monitor. https://www.bro.org/.
[9] Traffic Squeezer. http://www.trafficsqueezer.org/.
[10] Aho, A. V., and Corasick, M. J. Efficient string matching: an aid to bibliographic search. *Communications of the ACM 18*, 6 (1975), 333–340.
[11] Balakrishnan, H., Seshan, S., Amir, E., and Katz, R. H. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 1995), MobiCom '95, ACM, pp. 2–11.
[12] Bremler-Barr, A., Harchol, Y., Hay, D., and Koral, Y. Deep Packet Inspection as a Service. In *CoNEXT* (Sydney, Australia, 2-5 December 2014).
[13] Egevang, K., and Francis, P. The ip network address translator (nat). Tech. rep., RFC 1631, may, 1994.
[14] Fayazbakhsh, S. K., Chiang, L., Sekar, V., Yu, M., and Mogul, J. C. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), USENIX Association, pp. 543–546.
[15] Feilner, M. *OpenVPN: Building and integrating virtual private networks*. Packt Publishing Ltd, 2006.
[16] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM '14, ACM, pp. 163–174.
[17] Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. Achieving High Utilization with Software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 15–26.
[18] Jin, X., Li, L. E., Vanbever, L., and Rexford, J. SoftCell: Scalable and Flexible Cellular Core Network Architecture. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, ACM, pp. 163–174.
[19] Jin, X., Liu, H. H., Gandhi, R., Kandula, S., Mahajan, R., Zhang, M., Rexford, J., and Wattenhofer, R. Dynamic Scheduling of Network Updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM '14, ACM, pp. 539–550.
[20] Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Ingram, P., Jackson, E., Lambeth, A., Lenglet, R., Li, S.-H., Padmanabhan, A., Pettit, J., Pfaff, B., Ramanathan, R., Shenker, S., Shieh, A., Stribling, J.,

	Scalable Fine-Grained NF Policy	Low Performance Overhead	NF's Flow Migration	Host Mobility	Incremental Deployment
NF Policy Enforcing [32, 14]	TCAM size, TCAM update speed		not discussed	not discussed	inter-domain middlebox
NF Dynamic Control [16, 35]	TCAM size, TCAM update speed	Handle packets at the controller		not discussed	inter-domain middlebox
Naming service based [42, 39]		VPN, or per packet encapsulate and identify	not discussed	not discussed	[some] new naming system new socket abstraction

Table 4: Reasons that different NF policy steering and mobility solutions failed to fulfill the properties

Thakkar, P., Wendlandt, D., Yip, A., and Zhang, R. Network Virtualization in Multi-tenant Datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), USENIX Association, pp. 203–216.

[21] Liu, H. H., Wu, X., Zhang, M., Yuan, L., Wattenhofer, R., and Maltz, D. zUpdate: Updating Data Center Networks with Zero Loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 411–422.

[22] Mahajan, R., and Wattenhofer, R. On Consistent Updates in Software Dened Networks. In *12th ACM Workshop on Hot Topics in Networks (HotNets), College Park, Maryland* (November 2013).

[23] Marmorstein, R., Kearns, P., and William, T. C. An Open Source Solution for Testing NATd and Nested iptables Firewalls. In *19th Large Installation Systems Administration Conference (LISA 05* (2005), pp. 103–112.

[24] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), USENIX Association, pp. 459–473.

[25] McCanne, S., and Jacobson, V. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (Berkeley, CA, USA, 1993), USENIX'93, USENIX Association, pp. 2–2.

[26] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[27] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing Software-defined Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 1–14.

[28] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1999), SOSP '99, ACM, pp. 217–231.

[29] Natal, A. R., Jakab, L., Portolés, M., Ermagan, V., Natarajan, P., Maino, F., Meyer, D., and Aparicio, A. C. LISP-MN: mobile networking through LISP. *Wireless personal communications 70*, 1 (2013), 253–266.

[30] Nikander, P., Gurtov, A., and Henderson, T. Host Identity Protocol (HIP): Connectivity, Mobility, Multi-Homing, Security, and Privacy over IPv4 and IPv6 Networks. *Communications Surveys Tutorials, IEEE 12*, 2 (Second 2010), 186–204.

[31] Nordström, E., Shue, D., Gopalan, P., Kiefer, R., Arye, M., Ko, S., Rexford, J., and Freedman, M. J. Serval: An End-Host Stack for Service-Centric Networking. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), USENIX, pp. 85–98.

[32] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), SIGCOMM '13, ACM, pp. 27–38.

[33] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), USENIX, pp. 399–412.

[34] Rajagopalan, S., Williams, D., and Jamjoom, H. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 1:1–1:15.

[35] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), USENIX, pp. 227–240.

[36] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), SIGCOMM '12, ACM, pp. 323–334.

[37] Scarfone, K., and Mell, P. Guide to intrusion detection and prevention systems (idps). *NIST special publication 800*, 2007 (2007), 94.

[38] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), USENIX, pp. 323–336.

[39] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 13–24.

[40] Snoeren, A. C., and Balakrishnan, H. An End-to-end Approach to Host Mobility. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking* (2000), MobiCom '00, ACM, pp. 155–166.

[41] Tirumala, A., Qin, F., Dugan, J., Ferguson, J., and Gibbs, K. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects* (2005).

[42] Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., and Shenker, S. Middleboxes No Longer Considered Harmful. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 15–15.

[43] Zave, P., and Rexford, J. Compositional network mobility. In *Verified Software: Theories, Tools, Experiments*. Springer, 2014, pp. 68–87.

[44] Zhuang, S., Lai, K., Stoica, I., Katz, R., and Shenker,

S. Host Mobility Using an Internet Indirection Infrastructure. *Wirel. Netw. 11*, 6 (Nov. 2005), 741–756.