SRv6Pipes: enabling in-network bytestream functions

Fabien Duchene, David Lebrun, Olivier Bonaventure
ICTEAM, Université catholique de Louvain
Louvain-la-Neuve, Belgium
Email: firstname.lastname@uclouvain.be

***Abstract*—IPv6 Segment Routing is a recent IPv6 extension that is generating a lot of interest among researchers and in industry. Thanks to IPv6 SR, network operators can better control the paths followed by packets inside their networks. This provides enhanced traffic engineering capabilities and is key to support Service Function Chaining (SFC). With SFC, an end-to-end service is the composition of a series of in-network services. Simple services such as NAT, accounting or stateless firewalls can be implemented on a per-packet basis. However, more interesting services like transparent proxies, transparent compression or encryption, transcoding, etc. require functions that operate on the bytestream.**

In this paper, we extend the IPv6 implementation of Segment Routing in the Linux kernel to enable network functions that operate on the bytestream and not on a per-packet basis. Our SRv6Pipes enable network architects to design end-to-end services as a series of in-network functions. We evaluate the performance of our implementation with different microbenchmarks.

I. Introduction

Middleboxes play an important role in today's enterprise and datacenter networks. In addition to the traditional switches and routers, enterprise networks contain other devices that forward, inspect, modify or control packets. There is a wide variety of middleboxes [1], ranging from simple NAT, IP firewalls, various forms of Deep Packet Inspection, TCP Performance Enhancing Proxies (PEP), load balancers, Application Level Gateways (ALG), proxies, caches, edge servers, etc. Measurement studies have shown that some networks have deployed as many middleboxes as the number of traditional routers [2].

Those middleboxes were not part of the original TCP/IP architecture. They are typically deployed by either placing the middleboxes on the path of the traffic that needs to be handled, *e.g.*, on the link between two adjacent routers, or by using specific routing configurations to force some packets to pass through a particular middlebox. These two deployment approaches are fragile and can cause failures that are hard to diagnose and correct in large networks. Pothraju and Jain have shown in [3] that middlebox failures are significant and that many of them belong to a grey zone, *i.e.*, they cause link flapping or connectivity errors that are difficult to debug and impact the end-to-end traffic. Researchers and vendors have proposed Network Function Virtualization (NFV) [4] and Service Function Chaining (SFC) [5] to solve some of the problems caused by middleboxes.

ISBN 978-3-903176-08-9 © 2018 IFIP

In a nutshell, the NFV paradigm argues that all network functions should be virtualised and executed on commodity hardware instead of requiring specific devices. On the other hand, SFC [5] proposes to support chains of network functions which can be applied to the packets exchanged between communicating hosts. Several realisations for SFC are being discussed within the IETF. The SFC working group is developing the Network Service Header [6]. This new header can be used to implement service chains and replaces already deployed proprietary solutions. Another approach is to leverage the extensibility of IPv6. Given the global deployment of IPv6 [7], several large enterprises have already announced plans to migrate their internal network or their datacenters to IPv6-only to avoid the burden of managing two different networking stacks [8]. In addition to having a larger addressing space than IPv4, IPv6 provides several interesting features to support middleboxes in enterprise and datacenter networks. One of these is the native support for Segment Routing [9], [10]. Segment Routing (SR) is a modern variant of source routing that enables network administrators to enforce specific network paths.

In this paper, we demonstrate the benefits that the IPv6 Segment Routing (SRv6) architecture can bring to support middleboxes in enterprise and datacenter networks. With SRv6, middleboxes can be exposed in the architecture and visible end-to-end. This significantly improves the manageability of the network and the detection of failures while enabling new use cases where applications can select to use specific middleboxes for some end-to-end flows. This paper is organized as follows.

In Section II, we describe some use cases that can benefit from middleboxes. In Section III, we present *SRv6Pipes*, a modular SRv6-based architecture to support arbitrary in-network Virtual Functions, that can be applied on bytestreams and chained together. In Section IV, we detail a prototype implementation of our architecture, running on Linux. In Section V, we demonstrate the feasibility of our approach and evaluate the performance of our prototype through various tests and microbenchmarks. Finally, we cover some related work in Section VI and conclude in Section VII. Future work is discussed in VIII.

II. Use Cases

Middleboxes can perform two different types of network functions: *per-packet* and *per-bytestream*. The *per-packet* functions operate on a per-packet basis. They include Network Address Translation and simple firewalls. These functions typically operate on the network and sometimes transport headers. The *per-bytestream* functions are more complex, but also more useful. These functions operate on the payload of the TCP packets. For example, firewalls and Intrusion Detection Systems (IDS) need to match patterns in the packet payload while transparent compression and/or encryption need to modify the payload of TCP packets. Such functions need to at least reorder the received TCP packets but often need to include an almost complete TCP implementation. We describe some of these *per-bytestream* functions in more details in this section.

A. Application-level Firewalling

To cope with various forms of packet reordering, application-level firewalls and Intrusion Detection/Prevention Systems need to at least normalize the received packets [11] before processing them. Another approach is to use a transparent TCP proxy on the firewall to terminate the TCP connection and let the firewall/IDS process the reassembled payload. An end-to-end connection would thus be composed of two sub-connections: one between the client and the middlebox and another one between the middlebox and the server.

Network operators often configure access lists to associate IP prefixes to some security checks performed by the IDS. For example, students would be subject to different policies than servers.

B. Multipath TCP Proxies

Multipath TCP [12] (MPTCP) is a recent TCP extension that enables hosts to send packets belonging to one connection over different paths. One of the benefits of MPTCP is that it allows to aggregate the bandwidth of multiple connections. This enables, *e.g.*, network operators to bond xDSL and LTE networks to better serve rural areas [13]. However, MPTCP being an end-to-end protocol, the client and the server require an MPTCP-enabled kernels. To leverage the benefits of MPTCP without modifying the client or server network stacks, operators started developing MPTCP-aware proxies [13], [14] to convert regular TCP to MPTCP and conversely.

To allow the bundling of xDSL and LTE, an NFV deployment could be leveraged to implement the same behavior, by placing a proxy in the CPE to convert regular TCP to MPTCP and a second proxy in the operator's network to convert MPTCP to regular TCP. This would allow non-MPTCP clients and servers to use different networks simultaneously.

In practice, network operators could want to support different services on the same proxy, e.g. *(i)* a business proxy that always maximizes bandwidth for business customers, *(ii)* a low-cost proxy that only uses the LTE network when the xDSL network is fully utilized or *(iii)* a gaming proxy that always uses the network that provides the lowest delay. Such proxies can be deployed by tuning the packet scheduler and the path manager of Multipath TCP implementations.

C. Multimedia transcoding

Multimedia transcoding has been a research topic for a long time [15], [16]. Since, it has been widely deployed by companies like Amazon [17]. In this context, a proxy placed between the client and the server that hosts the multimedia file can be used to transcode the multimedia file hosted on the server into a format compatible with the client. This allows to distribute the computation intensive task of transcoding the content over several proxies, while the server simply hosts the original files. In this setup, parameters could be passed to the proxy to specify for instance the maximum bitrate that a client is entitled to (based on technical or subscription limitations), the maximum number of streams allowed for this client or the type of content authorized for this client.

III. Architecture

Middleboxes and other in-network functions are installed, configured, and managed by network administrators according to business (e.g. security regulations impose the utilisation firewalls) and technical (e.g. performance issues force the utilisation of performance enhancing proxies, or addressing issues force the utilisation of NAT) needs. Usually, network administrators impose the utilisation of specific network functions by configuring routing policies or placing physical boxes on links that carry specific traffic (e.g. firewalls are often attached to egress links). This is both cumbersome and costly since all possible links must be covered by each intended network function.

Like NFV, our architecture assumes that network functions are software modules which can be executed anywhere in the network. A firewall function that only needs to process the external flows does not need to be installed on the egress router, it can be executed on any server or router inside an enterprise network. Each network function is identified by an IPv6 prefix which is advertised by the equipment hosting the function (see section III-C). For redundancy or load-balancing, the same function can be hosted on different equipments in the network.

To understand the different elements of our architecture, let us consider a simple scenario. A client host needs to open a TCP connection towards a remote server. The network administrator has decided that the packets belonging to such a connection must be processed by two network functions: *(i)* a stateless firewall which blocks prohibited ports and *(ii)* a DPI which inspects all external TCP connections. Three elements of our architecture are used to support this sequence of network functions in enterprise networks.

The first element is IPv6 Segment Routing (SRv6) [18]. Our architecture uses the SRv6 header (SRH) to enforce an end-to-end path between the client and the server which passes through the two equipments hosting the mandatory networking functions. We describe SRv6 in more details in section III-A.

The second element of our architecture is how the client learns the SRH suitable to reach a given destination. For this, we modify the enterprise DNS resolver. Instead of simply resolving names into addresses, our DNS resolver acts as a controller [19], [20] which has been configured by the network administrator with various network policies. When a client sends a DNS request to the resolver, it replies with the intended response and additional records which contain the SRH that the client has to apply to reach the specified addresses.

Thanks to the SRH which is attached by the client, all the packets belonging to the TCP connection will pass through the stateless firewall and the DPI. Consider now what happens if some packets are lost and need to be retransmitted. The stateless firewall is not affected since it only processes the network and transport headers that are present in each packet. On the other hand, the DPI function needs to include a TCP implementation to be able to detect out-of-order packets or other TCP artifacts. Instead of requiring each network function to include a TCP implementation, our architecture leverages the TCP stack that is already present in the Linux kernel. Each equipment that hosts a network function uses a transparent TCP proxy that transparently terminates the TCP connections and exposes bytestreams to the network functions as in FlowOS [21]. This greatly simplifies the implementation of per-bytestream network functions

A. IPv6 Segment Routing

Segment Routing (SR) is a modern variant of the source routing paradigm, currently under standardization at the IETF [18]. SR can be used on top of an MPLS or IPv6 dataplane to steer packets through an ordered list of *segments*. SR is now well-supported on commercial routers [23] and Linux hosts [24] and deployed by major ISPs [9].

The IPv6 flavor of Segment Routing (SRv6) leverages a dedicated IPv6 routing extension header, named Segment Routing Header (SRH) [10]. Each segment is an IPv6 address representing a node or link to traverse, or an intermediate function to be executed. The SRH contains a full list of segments. The active segment is referenced by an index, the segment pointer. As the list of segments is encoded in reverse order, the index is first initialized to the last element of the list (*i.e.*, the first segment of the path), and decremented at each *segment endpoint*. The segment pointer thus reaches zero when arriving at the last segment of the path. The active segment is also written as the destination address in the IPv6 header. As such, transit nodes on the path to an active segment simply needs to support plain IPv6 forwarding. SRv6 support is only required at the segment endpoints.

In SRv6Pipes, we leverage the SRv6 architecture to steer TCP flows through arbitrary network functions. See Figure 1 for an illustration. Consider that the client `C` establishes a connection to a server `S`, with two intermediate network functions at `P1` and `P2`. To realise that, the client attaches an SRH to its packets, containing three segments. The first two segments represent the functions to be executed at resp. `P1` and `P2`. The third segment is the address of `S`. When the

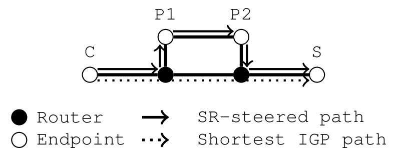


Fig. 1: Traffic steering through two off-path network functions `P1` and `P2` (*e.g.*, firewall and IDS).

$$\underbrace{\text{2001:0123:4567:8901:2345}}_{\text{Proxy range}}:\overbrace{\text{AAAA}}^{}_{\text{Function}}:\underbrace{\text{BCDE:FFFF}}_{\text{Parameters}}$$

Fig. 2: IPv6 address encoding.

packets are transiting between `C` and `P1`, and between `P1` and `P2`, their IPv6 destination address is thus the address of the function to execute at the corresponding proxy. Between `P2` and `S`, the segment pointer is decremented to zero and the IPv6 destination address of the packets is the address of `S`.

B. Transparent TCP Proxy

The proxy is the core component of our architecture to support per-bytestream network functions. It is transparent at the network layer, meaning that even if the proxy actually terminates the TCP connection with the client, the destination server will receive packets coming from the client's IP address, and not from the proxy's IP address. The transparent proxy is placed on path using the IPv6 Segment Routing Header (SRH) [10]. It intercepts each new connection that matches a given pattern (*e.g.*, a destination port) and terminates it. Then, the proxy establishes a downstream connection to the next segment specified in the SRH of the inbound connection. When the proxy receives data from the client, it applies a transformation function (*i.e.*, the Virtual Function) to the received data and forwards the result on its outbound connection to the next segment of the path. This process is then repeated until reaching the final destination of the path.

C. Encoding Functions and Parameters

As shown in section II, some parameters can be passed to the per-bytestream function. Such parameters are usually specified in the proxy configuration files. However, such configurations can be large and complex if some parameters can change on a per connection basis. Consider for example a first proxy that encrypts the payload and a second that decrypts it. Those encryption/decryption proxies would have to be configured with the encryption/decryption keys for each flow. A possible approach would be to define one key per host or set of hosts. A better approach is to configure a set of keys on the proxies and associate each key with a unique identifier. When a connection starts, the encryption proxy selects a random key and places the identifier of the chosen key in the SRH towards the decryption proxy.

To enable such a granularity in the choice of transformation functions and parameters, we leverage the large addressing space available in IPv6. Each proxy announces one or more IPv6 prefixes that correspond to the Virtual Functions it hosts.

Within the host part of the prefixes, we allocate a given amount of bits to encode the identifier of the function to apply as proposed in [25]. The remaining low order bits are used to specify parameters of the virtual function such as the decryption key in the above example. Consider Figure 2 for an illustration. The proxies announce `/80` prefixes. The first 80 bits of the address thus specify the proxy to traverse. The 16 following bits identify the function to apply to the payload, and the low-order 32 bits contain the parameters of these functions. The SRH then contains a list of proxies with their respective functions and parameters. This approach allows the clients to use any combination of function/parameter available in the network.

Consider the network described in figure 1. In this network, the client might require to encrypt the traffic between `P1` and `P2`. In our architecture, the client will use the `function` bits of the address of `P1` to specify the identifier of the `encrypt` function, and the `parameters` bits to specify the identifier of an encryption key. The same will be done in the address of `P2` with the `decrypt` function. This allows to have different encryption keys for different connections without having to store a configuration for each connection in the proxy. The processing of the return traffic is discussed in IV-E.

D. SRv6 Controller

In our architecture, a TCP client is able to specify arbitrary functions to apply to its traffic. However, keeping track of all the functions, parameters, and proxies addresses represents a significant amount of complexity. This complexity can be abstracted by a central SDN-like controller. We leverage the *SDN Resolver*, which is a DNS-based, SRv6 controller introduced in [19], [20]. Before establishing a connection, the client sends a request to the controller with the address of the server and a list of functions to apply to the traffic. The controller then computes a path that matches the request and returns an SRH to the client. A key element of this controller is that the SRH returned to the client does not contain the full list of segments. Instead, it contains only one segment, called the *binding segment*. The access router of the client is configured by the controller to translate this binding segment into the full list of segments. This abstraction enables the clients to be oblivious to changes in the SRH induced by, *e.g.*, a network failure. The architectural and implementation details of *SDN Resolver* are available in [19], [20]. Note that the DNS protocol serves as an example, that can be replaced by any ad-hoc application-facing protocol.

E. Security Considerations

The ability to execute and chain arbitrary functions in the network has obvious security implications. To restrict the privilege of using SRv6Pipes proxies, we can leverage the central controller presented in the previous section, as well as its *binding segment* mechanism. By configuring all access routers to accept only SRHs with known binding segments, we can effectively prevent an uncontrolled usage of network functions. The decision to accept or deny the use of a given set of functions is made by the controller, which can identify clients through independent channels [19].

IV. Implementation

To demonstrate the feasibility of our approach, we implemented a prototype of our solution by extending the implementation of IPv6 Segment Routing in the Linux kernel [24]. The main new component of our prototype is a transparent, SR-aware, TCP proxy. For this, we extended the kernel implementation of SRv6 with a new type of function. An overview of the various data paths in our prototype is shown in Figure 3.

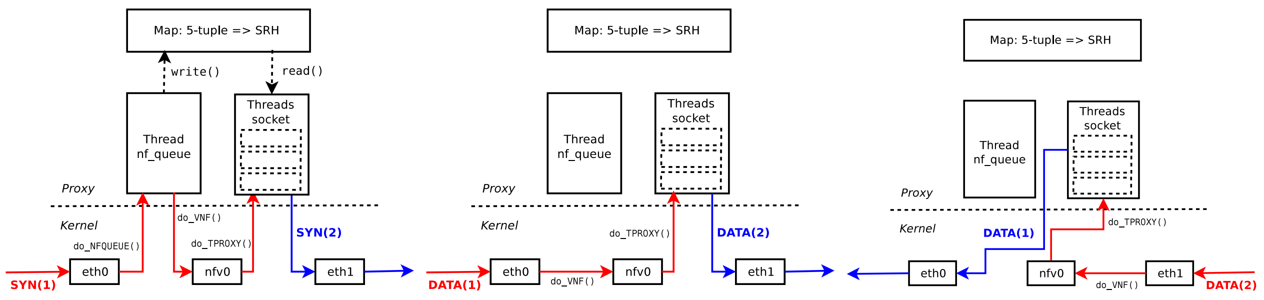
To ensure that our proof of concept could easily be used to reproduce our results on any off-the-shelf hardware, we implemented it using the regular Linux mechanisms. Alternatives solutions are discussed in Section VIII

A. Transparent SR-Aware TCP Proxy

The core objective of our proxy is to process and relay TCP streams between two segments of a segment routed path. To achieve this, the proxy must (i) intercept and terminate incoming TCP flows, (ii) optionally apply transformation functions to the bytestreams, and (iii) initiate and maintain the corresponding TCP flows to the next segment of the path.

To intercept TCP flows, the proxy must accept connections towards pairs of IP/port that are not local to the machine, which is not possible by default. The Linux kernel provides the `TPROXY` iptables extensions, enabling such interceptions. It works by redirecting all packets matching an iptables rule towards a local IP/port pair. The proxy is then able to intercept the corresponding TCP flows by listening to this local pair.

Once a TCP flow is intercepted and terminated, the proxy needs to retrieve the associated SRH, decrement its segment pointer, and install it on the corresponding outbound socket. The `IPV6_RECVRTHDR` socket option could be used to fetch any attached Routing Header (RH) as ancillary data, using the `recvmsg()` system call. However, this feature is only implemented for datagram protocols such as UDP, where a single RH is associated to each datagram. In bytestream protocols such as TCP, packets can be merged and the $1:1$ mapping to RHs is lost. In our prototype, we rely on the SRH included in the SYN packet of a given TCP flow. As the kernel does not expose Routing Headers for TCP flows, we leverage the `NFQUEUE` iptables extension to capture SYN packets in user space. The proxy opens a netlink channel with the kernel and receives through it all SYN packets matching the corresponding iptables rule. Then, the proxy extracts the 5-tuple and the SRH from the SYN packet and stores them in a `flows_srh` map. Finally, the packet is reinjected into the kernel. Following its normal data path, the SYN packet will trigger a connection request to the proxy. Using the 5-tuple, the proxy is then able to retrieve the SRH previously stored in the `flows_srh` map. While capturing every packet in user space can severely degrade the performances, our solution does not suffer from such degradation as we only capture the first packet of each flow.



(a) Traversal of a SYN packet through the proxy. The SRH is recorded for the 5-tuple.

(b) Traversal of data packets.

(c) Traversal of return packets.

Fig. 3: Overview of possible data paths within SRv6Pipes.

After having intercepted a TCP flow and extracted its SRH, the proxy must establish the corresponding TCP flow to the next iteration of the path. To achieve this, the proxy creates the outbound socket and attaches to it the corresponding SRH. Additionally, the connection must appear as originating from the actual source of the flow. Using the `IP_FREEBIND` socket option, the proxy is able to bind to a non-local IP/port pair. Finally, the connection is established and data can be exchanged.

Once both connections (inbound and outbound) are established, the proxy only needs to forward data coming from one socket to the other one, after going through an optional transformation function. In our prototype, we use an application-level buffer to transfer data from one connection to the other. Another possible solution would be to use the `splice()` system call to let the kernel directly move data between file descriptors. However, this solution prevents the proxy from actually modifying the data. Our approach allows the implementation of arbitrary transformation functions. The termination of connections is straightforward. Once one socket is closed, any in-flight data is flushed and the other socket is also closed.

We implemented a multi-threaded architecture, enabling the proxy to scale with the load. One dedicated thread handles the `NFQUEUE` channel, receives the SYN packets, and populates the `flows_srh` map accordingly. A configurable number of threads (typically one per CPU thread) accept incoming connections, establish the outbound connection, and process the data exchanged between them. Each of these threads leverages the `SO_REUSEPORT` socket option, enabling them to simultaneously listen to the same local IP/port. The result is that the kernel maintains distinct accept queues for each thread. Consequently, incoming connections are equally load-balanced across the running threads.

B. Kernel Extensions

When a packet to be processed by the proxy enters the kernel, its IPv6 destination address corresponds to the local proxy function. However, the TCP checksum was originally computed for the actual destination of the packet. As such, it is transiently incorrect, due to the SR-triggered change of destination address. Additionally, the packet will be associated to the proxy's local socket by the `TPROXY` module, and subsequently injected in the local stack. However, the segment pointer of the associated SRH is non-zero. The packet will thus enter the SRH processing and the kernel will attempt to forward it to the next segment, bypassing the local TCP processing [24].

To address those two issues, we extend the SRv6 kernel implementation available in Linux 4.14 and add a new type of function called `End.VNF`. This function takes one parameter (an egress interface) and performs the following actions. First, it updates the destination address of the packet to its final destination. Then, it sets the segment pointer to zero[1]. Finally, it injects the resulting packet into the specified egress interface using `netif_rx()`. In our prototype, we leverage a virtual dummy interface (`nfv0`). As a result, all packets to be intercepted by the proxy are received through this particular interface and are thus easily distinguished from background traffic.

C. System Configuration

To instantiate the proxy, a non-trivial configuration of iptables and routing tables is required. An example of this configuration is shown in Figure 4. The first two lines create the `nfv0` interface to receive all packets to be intercepted by the proxy. Lines $3-5$ create a `DIVERT` iptable chain that sets the mark `1` on packets and accepts them. Line 6 creates an `NFQUEUE` rule that matches all SYN packets whose destination address corresponds to the local proxy (`PROXY_FUNC_ADDR`) and sends them to the queue number 0. Line 7 matches all TCP packets received on interface `nfv0` and sends them to the `TPROXY` target. The latter will set the mark `1` on those packets and will associate them to a socket bound on a local `PROXY_LOCAL_PORT` port. Line 8 matches all TCP packets that can be associated to an open socket and sends them to the previously configured `DIVERT` chain. In practice, this rule will catch the inbound return packets that are not caught by the two previous rules. Line

[1]As the SRH of the SYN packet was previously extracted by the proxy, this information is not lost.

```
1: ip link add nfv0 type dummy
2: ifconfig nfv0 up
3: ip6tables -t mangle -N DIVERT
4: ip6tables -t mangle -A DIVERT -j MARK --set-mark 1
5: ip6tables -t mangle -A DIVERT -j ACCEPT
6: ip6tables -t mangle -A PREROUTING -d \$PROXY_FUNC_ADDR -p tcp --syn -j NFQUEUE --queue-num 0
7: ip6tables -t mangle -A PREROUTING -i nfv0 -p tcp -j TPROXY --tproxy-mark 0x1/0x1 --on-port \$PROXY_LOCAL_PORT
8: ip6tables -t mangle -A PREROUTING -p tcp -m socket -j DIVERT
9: ip -6 rule add fwmark 1 table 100
10: ip -6 route add local ::/0 dev lo table 100
11: ip -6 route add \$PROXY_FUNC_ADDR/128 encap seg6local action End.VNF oif nfv0 dev eth0
12: sysctl net.ipv6.conf.nfv0.seg6_enabled=1
```

Fig. 4: System configuration for the proxy.

9 creates a routing rule instructing the kernel to lookup table 100 for all packets having the mark `1`. Line 10 creates a single routing entry into table 100 that matches all packets and sends them in the local stack (instead of forwarding them). Line 11 creates an SRv6 routing entry that matches all packets towards `PROXY_FUNC_ADDR` and applies the `End.VNF` function, using `nfv0` as the egress interface[2]. Finally, line 12 enables the processing of SRv6 packets on interface `nfv0`.

D. Modular Transformation Functions

To support transformation functions in a modular way, our SRv6Pipes proxy leverages Linux dynamic libraries. Functions can be compiled in `.so` (shared object) files. Those files are independent modules that can be loaded and unloaded at run-time by the proxy. Each module exports an `all_funcs` symbol. This symbol refers to an array of `func_desc` structures. Each of those structures describes a single transformation function, through the following symbols. The `func_init()` symbol is called once, on module load. It registers the function with a given function identifier, which is passed in the IPv6 destination addresses (see Section III-C). The `func_spawn()` symbol is called each time a new intercepted TCP flow matches the function identifier. Any parameter passed in the low-order bits of the IPv6 destination address is passed as argument. The role of this symbol is to initialize per-connection data. The `func_process()` symbol is the actual transformation function. It reads data from an input buffer and writes the transformed data in an output buffer. The `func_despawn()` symbol is called at connection termination and it frees previously allocated per-connection data. Finally, the `func_deinit()` symbol is called at module unload and de-registers function identifiers.

Such an architecture enables to easily add, modify, and remove transformation functions, without updating nor restarting the proxy's binary.

E. Return Traffic

The previous sections detailed the processing of the upstream traffic (from client to server). However, if the middleboxes are not located in-path, the downstream traffic (from the server to client) must also be augmented with an SRH. This is also necessary to enable asymmetrical processing functions, *i.e.*, using different transformation functions depending on the direction of the traffic. To achieve this, multiple options exist.

The straightforward option is to simply "*reverse*" the SRH received from the client or from the previous proxy. Each proxy can simply apply the segments of the initial SRH in reverse order. While this solution is simple and does not incur a significant overhead, it as a major limitation: the segments must necessarily be symmetrical, making asymmetrical processing functions impossible.

To enable asymmetrical processing functions, another option is to embed the return SRH in a TLV extension of the initial SRH. With this solution, after inserting the SRH, the client inserts a TLV to the socket before establishing the connection. Then, each proxy and the server extract the SRH to be used on the return path from the TLV received in the initial packet (SYN). The TLV could also be transmitted with every upstream packet, but this would increase the overhead. With this TLV, it is important to note that the return path must include every proxy that is present in the upstream path, but that others segments, e.g. corresponding to specific paths or routers, can be added or suppressed.

In our prototype, we implemented the second solution by modifying the Linux kernel to add support for such a TLV. When a new TCP socket is created after receiving an SR-enabled SYN packet containing the return-path TLV, this return path is extracted and installed as an outbound SRH for the newly created socket. If the proxies are located in-path, our prototype can also work without an SRH on the return path. This is realized using the `DIVERT` rules shown in figure 4. In Section V, we evaluate this on-path mode.

V. Evaluation

In this section, we use microbenchmarks to evaluate the performance of our prototype in our lab. For this evaluation, we use three Linux PCs connected with 10Gbps interfaces as shown in figure 5.

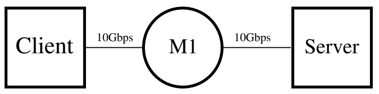


Fig. 5: Lab setup. M1 can be configured as router or proxy.

[2]While this interface is considered egress from the point of view of `End.VNF`, packets are actually received on that interface and it is thus considered ingress for the next components in the datapath.

The client is a 2,53Ghz Intel Xeon X3440 with 16GB of RAM. M1 and the server use the same hardware configuration but with only 8GB of RAM. They are all equipped with Intel 82599 10Gbps Ethernet adapters and use 9000 bytes MTU. They all use our modified version of the latest IPv6 Segment Routing kernel based on the Linux kernel version 4.14. The server runs `lighttpd` version 1.4.35. The client uses `wrk` [26] 4.0.2-5 to load the server with HTTP 1.1 requests. We slightly modified `wrk` to add an IPv6 SRH as a socket option when creating TCP connections. M1 can be configured either as a router or with our transparent proxy. When used as a router, we create static routes and use the standard Linux IPv6 forwarding.

A. Maximum throughput

First, we compare the performance of one of our proxies against the performance of a Linux router running on the same platform. In this setup, our client uses `wrk` [26] to simulate 200 web client downloading static web pages of given sizes during 120 seconds. It uses 8 threads with 25 connections per thread. The proxy was configured with a virtual function that directly copies that bytestreams without any processing.

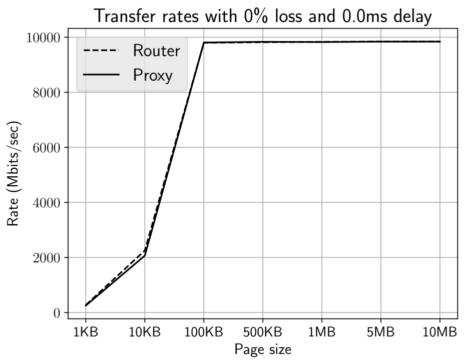


Fig. 6: Raw throughput.

Figure 6 shows the total transfer rate when the client is downloading web pages. This figure shows that there is no significant difference in transfer rates between our proxy and the router. With 10MB files, our proxy reaches a throughput of 9841 Mb/s where the router achieves 9838 Mb/s. A closer look at the small page sizes in figure 6, shows that our proxy slightly underperforms the router. With 1KB files, our proxy achieves a rate of 253 Mb/s, while the router achieves a rate of 272 Mb/s.

In term of requests per second, for 1KB files, our proxy completes 26634 requests per second, while the router completes 28613 requests per second. This difference in performance between large and small files can be explained by the fact that when our proxy receives a new connection from the client, it needs to establish a new connection to the server before starting to forward packets. With smaller files, there are significantly more three-way handshakes to perform, making this overhead more important while this cost is amortized for larger files. With 100KB files, the number of requests per seconds is already on par at ≈11945 requests per second for both the proxy and the router.

B. Impact of packet losses and latency on the proxies

The previous section explored the maximum rate that our proxies can sustain. In those measurements, the TCP stack running on M1 did not have to buffer packets or handle retransmissions. As those operations can affect its performance, we added `netem` to simulate different delays and different packet loss ratios.

We start by adding a 1% loss and a 25ms delay on the four links of figure 5. This corresponds to an end-to-end loss of ≈4%, and an end-to-end latency of 100ms. The results of this measurement are shown on figure 7. Under such circumstances, our proxy outperforms the router. This is not surprising since in this setup, our proxy acts as a Performance Enhancing Proxy (PEP). While figure 7 clearly shows a large improvement for large file sizes, our measurements indicated that this is also true for small file sizes. This can be explained by the fact that when M1 is configured as a router all packet losses need to be recovered end-to-end. When a packet is lost on the same link with our proxy, the retransmission is done by the proxy.

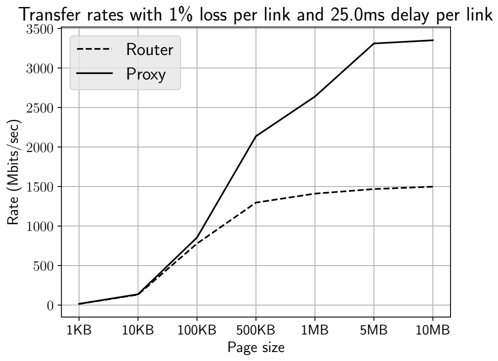


Fig. 7: Transfer rate with 1% of loss and 25ms of latency per link.

To confirm our findings, we run the same measurement, but adding latency and loss only on the link between the server and the proxy, the objective being to mimic a network where the loss would happen only on the link between the proxy and the server. To replicate our previous configuration, we add 2% of loss per link, to get an end-to-end loss of ≈4%, and 50ms of latency per link to get an end-to-end latency of 100ms. As shown by figure 8, under such conditions, the proxy and the router are both significantly affected by the performance degradation in the same fashion, confirming our findings.

C. CPU-intensive Virtual Functions

With our architecture, various types of Virtual Functions can be implemented. Some like a PEP simply proxy the connections and do not need to process the payload. Others like DPIs, transparent compression or transparent encryption need to process the payload and thus consume CPU cycles. To

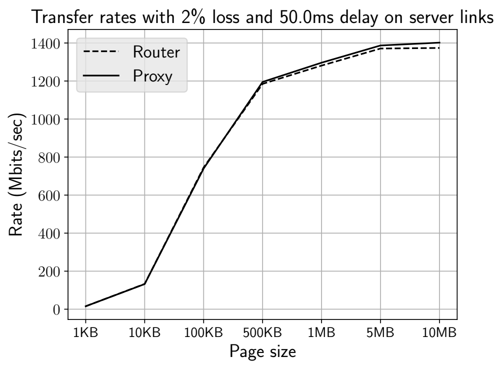


Fig. 8: **Transfer rate with 2% of loss per link and 50ms of latency per link between the proxy and the server.**

measure the impact of the Virtual Function on the performance of our proxy, we developed a simple microbenchmark that performs $2 \times n$ passes over the bytestream and XORs each byte with a key at each pass. This VF leaves the bytestream unmodified, but consumes both CPU and accesses memory.

The results with this microbenchmark are shown in figure 9. When our VF performs two passes on the bytestream, the maximum throughput is similar to the one we obtained without bytestream modification in figure 6. When the VF performs four passes on the bytestream, the maximum throughput with pages larger than 100KB is divided by 2. This throughput continues to drop with the CPU load on the VF. To confirm that the reduction in throughput was due to the CPU intensive computations, we ran `perf` [27] that yielded 96% of cycles spent in the XOR function.

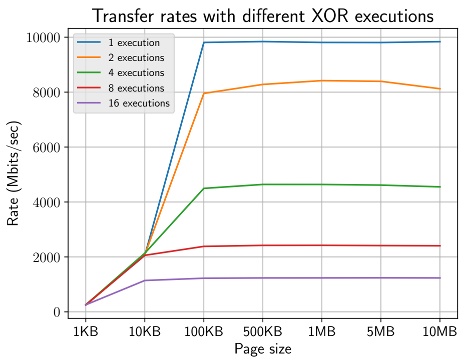


Fig. 9: Maximum throughput with Virtual Functions performing n passes over the bytestream.

D. Chaining middleboxes

With our architecture, middleboxes can be used in chains where one middlebox performs the opposite function of the previous one. Typical examples include transparent compression/decompression or transparent encryption. To demonstrate this use case, we implemented a VF that simply XORs each byte of the bytestream with a constant. When two such middleboxes are used in sequence, the bytestream output of the downstream one is the same of the input of the upstream one. This is illustrated in figure 10.

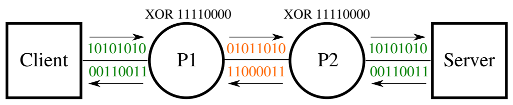


Fig. 10: Demonstration of middlebox chaining with simple XOR transformations.

Due to limitations of our lab, we could only perform this experiment over 1 Gbps links. Figure 11 shows that with the two chained middleboxes, the maximum throughput was the same as when passing through two routers. This is expected given the results of figure 9 with 10Gbps interfaces.

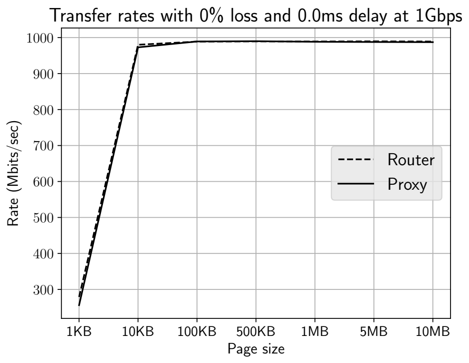


Fig. 11: Transfer rate of wrk with 2 proxies applying a XOR.

VI. Related Work

AbdelSalam et al. propose in [28] to use IPv6 Segment Routing to support Virtual Network Function Chaining and implement a prototype as a Linux kernel module. They leverage namespaces to support virtual network functions but only support packet-based functions while our solution leverages the Linux TCP stack to provide a bytestream abstraction to the network functions. In FlowOS, Bezahaf et al. [21] proposed a Linux kernel module that exposes a bytestream abstraction to network functions but they do not describe how flows are routed through the network functions. NetVM [29] leverages virtualization techniques and a user-space packet processing platform to provide fast, chainable network functions in virtual machines. Their work focuses on packet processing and does not consider bytestream functions. Other solutions such as XOMB [30] focus on the system aspects of implementing virtual functions to support middleboxes through a flexible programming model. Our architecture leverages IPv6 Segment Routing to forward the packets to the middleboxes. Another related work is `/dev/stdpkt` proposed by Utsumi et al in [31]. `/dev/stdpkt` uses the Linux Kernel Library to implement virtual functions that can be chained together.

VII. Conclusion

Given its ability to enforce precise network paths for specific flows, IPv6 Segment Routing appears to be an excellent candidate to support middleboxes in entreprise networks. We leverage this IPv6 extension in our architecture designed for enterprise networks. Its main benefit is that the middleboxes are explicitly exposed. This significantly improves the manageability of the network. Our architecture supports both middleboxes that operate on a per-packet basis (e.g. NAT, stateless firewalls) and those that need to process bytestreams (e.g. DPI, Application Level Gateways, ...). For the latter, we use transparent TCP proxies that process the IPv6 Segment Routing Header. We implement[3] this architecture in the Linux kernel and evaluate its performance with various benchmarks in our lab. Our measurements indicate that our architecture is well suited to support middleboxes that process bytestreams.

VIII. Future Work

In this paper, we implemented a proof of concept using the regular Linux mechanisms. While kernel bypass techniques such as DPDK or user-space TCP stacks like mTCP allow significant performance boosts, they are often specific to a subset of network hardware. By leveraging the kernel datapath, our solution remains generic and can be deployed on any Linux-supported hardware, ranging from high-end servers to home routers. Should an operator require performance only available through kernel bypass techniques, our high-level network architecture would remain identical and our userspace implementation of the proxy would require minimal changes to plug-in with a DPDK-like library. These modifications can be realized as future work.

IX. Acknowledgements

This work was partially supported by the ARC-SDN project and a Cisco URP grant.

References

[1] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. RFC 3234 (Informational), February 2002.

[2] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[3] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 9–22. ACM, 2013.

[4] Kaustubh Joshi and Theophilus Benson. Network function virtualization. *IEEE Internet Computing*, 20(6):7–9, 2016.

[5] J. Halpern and C. Pignataro. Service Function Chaining (SFC) Architecture. RFC 7665 (Informational), October 2015.

[6] P. Quinn, U. Elzur, and C. Pignataro. Network Service Header (NSH). Internet draft, draft-ietf-sfc-nsh-28, November 2017.

[7] Mehdi Nikkhah and Roch Guérin. Migrating the internet to ipv6: an exploration of the when and why. *IEEE/ACM Transactions on Networking*, 24(4):2291–2304, 2016.

[8] Mat Ford. Landmark ipv6 report published: State of deployment 2017. CircleID, http://www.circleid.com/posts/20170606_landmark_ipv6_report_published_state_of_deployment_2017/, June 2017.

[9] Clarence Filsfils et al. The segment routing architecture. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2015.

[3]To ensure the reproducibility of our results, our implementation and the measurement scripts will be released on http://segment-routing.org/index.php/SRv6Pipes at publication time.

[10] Stefano Previdi et al. IPv6 Segment Routing Header (SRH). Internet-Draft draft-ietf-6man-segment-routing-header-07, Internet Engineering Task Force, July 2017. Work in Progress.

[11] Christian Kreibich, Mark Handley, and V Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, volume 2001, 2001.

[12] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), January 2013.

[13] Olivier Bonaventure and SungHoon Seo. Multipath TCP deployments. *IETF Journal*, 2016, November 2016.

[14] Olivier Bonaventure, Mohamed Boucadair, Bart Peirens, SungHoon Seo, and Anandatirtha Nandugudi. 0-RTT TCP Converter. Internet-Draft draft-bonaventure-mptcp-converters-02, Internet Engineering Task Force, October 2017. Work in Progress.

[15] Elan Amir, Steven McCanne, and Randy Katz. An active service framework and its application to real-time multimedia transcoding. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 178–189. ACM, 1998.

[16] Jun Xin, Chia-Wen Lin, and Ming-Ting Sun. Digital video transcoding. *Proceedings of the IEEE*, 93(1):84–97, 2005.

[17] Amazon Elastic Transcoder. https://aws.amazon.com/fr/elastictranscoder/. Accessed: 2018-04-05.

[18] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. Internet-Draft draft-ietf-spring-segment-routing-14, Internet Engineering Task Force, December 2017. Work in Progress.

[19] David Lebrun. *Reaping the Benefits of IPv6 Segment Routing*. PhD thesis, UCLouvain / ICTEAM / EPL http://hdl.handle.net/2078.1/191759, October 2017.

[20] David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure. Software resolved networks: Rethinking enterprise networks with ipv6 segment routing. In *SOSR'18: Symposium on SDN Research*, 2018.

[21] Mehdi Bezahaf, Abdul Alim, and Laurent Mathy. Flowos: A flow-based platform for middleboxes. In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '13, pages 19–24, New York, NY, USA, 2013. ACM.

[22] Bhavish Agarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. Endre: An end-system redundancy elimination service for enterprises. In *NSDI*, pages 419–432, 2010.

[23] Clarence Filsfils, Francois Clad, Pablo Camarillo, Jose Liste, Prem Jonnalagadda, Milad Sharif, Stefano Salsano, and Ahmed AbdelSalam. Ipv6 segment routing. In *SIGCOMM'17, Industrial demos*, August 2017.

[24] David Lebrun and Olivier Bonaventure. Implementing IPv6 Segment Routing in the Linux Kernel. In *Proceedings of the 2017 Applied Networking Research Workshop*. ACM, July 2017.

[25] P. Camarillo et al. Srv6 network programming. Internet draft, draft-filsfils-spring-srv6-network-programming-02, work in progress, October 2017.

[26] wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. Accessed: 2017-12-31.

[27] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/. Accessed: 2018-03-29.

[28] Ahmed AbdelSalam, Francois Clad, Clarence Filsfils, Stefano Salsano, Giuseppe Siracusano, and Luca Veltri. Implementation of virtual network function chaining through segment routing in a linux-based nfv infrastructure. In *IEEE Conference on Network Softwarization (NetSoft)*, Bologna, Italy, July 2017.

[29] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 445–458, Berkeley, CA, USA, 2014. USENIX Association.

[30] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xomb: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 49–60, New York, NY, USA, 2012. ACM.

[31] Motomu Utsumi, Hajime Tazaki, , and Hiroshi Esaki. `/dev/stdpkt`: A service chaining architecture with pipelined operating system instances in a unix shell. In *AINTEC '17: Asian Internet Engineering Conference*, Bangkok, Thailand, November 20–22 2017.