SERA: SEgment Routing Aware Firewall for Service Function Chaining scenarios

Ahmed Abdelsalam*, Stefano Salsano†, Francois Clad‡, Pablo Camarillo‡, Clarence Filsfils‡,
*Gran Sasso Science Institute, †University of Rome Tor Vergata, ‡Cisco Systems

***Abstract*—In this paper we consider the use of IPv6 Segment Routing (SRv6) for *Service Function Chaining* (SFC) in an NFV infrastructure. We first analyze the issues of deploying *Virtual Network Functions* (VNFs) based on SR-unaware applications, which require the introduction of SR proxies in the NFV infrastructure, leading to high complexity in the configuration and in the packet processing. Then we consider the advantages of SR-aware applications, focusing on a firewall application. We present the design and implementation of the SERA (SEgment Routing Aware) firewall, which extends the Linux iptables firewall. In its basic mode the SERA firewall works like the legacy iptables firewall (it can reuse an identical set of rules), but with the great advantage that it can operate on the SR encapsulated packets with no need of an SR proxy. Moreover we define an advanced mode, in which the SERA firewall can inspect all the fields of an SR encapsulated packet and can perform SR-specific actions. In the advanced mode the SERA firewall can fully exploit the features of the IPv6 Segment Routing network programming model. A performance evaluation of the SERA firewall is discussed, based on its result a further optimized prototype has been implemented and evaluated.**

***Index Terms*—Service Function Chaining (SFC), NFV, Segment Routing, Linux networking, Firewall, Iptables**

I. Introduction

The advent of Network Function Virtualization (NFV) [1] is dramatically changing the way in which telecommunication networks are designed and operated. Traditional specialized physical appliances are replaced with software modules running on a virtualization infrastructure made up of general purpose servers. Such virtualization infrastructure can even be composed of a set of geographically distributed data centers. In traditional "pre-NFV" networking, the physical appliances were placed *en-route*, i.e. along the path of the flows. In NFV scenarios, the Virtual Network Functions (VNFs) that replace the physical appliances can be arbitrarily located in the distributed virtualization infrastructure, hence the need of steering the traffic flows through the sequence of VNFs to be accessed. The VNFs can also be denoted as Service Functions (SF) and the *Service Function Chaining* (SFC) [2] denotes the process of forwarding packets through the sequence of VNFs. Examples of VNFs categories are NATs (Network Address Translation), firewalls, DPIs (Deep Packet Inspection), IDSs (Intrusion Detection System), load balancers, HTTP proxies, CDN nodes.

The IETF SFC Working Group (WG) has investigated the SFC scenarios and issues [3] and proposed a reference architecture [4]. A specific mechanism, called Network Service Header (NSH) [5] has been proposed by the SFC WG to support the encapsulation of packets with a header that specifies the sequence of services (VNFs) to be crossed.

In this paper, we consider the use of the IPv6 Segment Routing (SRv6) architecture to support Service Function Chaining, as already discussed in [6]. In the SRv6 architecture an IPv6 extension header (the Segment Routing Header - SRH) allows including a list of *segments* in the IPv6 packet header [7]. This segment list can be used to steer the packet though a set of intermediate steps in the path from the source to the destination, following a (loose) source routing approach. The use of Segment Routing for SFC has been documented in [8]. The typical network scenario is that an edge node classifies the traffic and consequently includes a segment list in the IPv6 packet header. Note that the application of SRv6 is not limited to SFC, there are many other important use cases [9] like for example traffic engineering, fast restoration, support of Content Delivery Networks. The concept of SRv6 has been extended in [10], from the simple steering of packets across nodes to a general *network programming* approach. The idea is to encode *instructions* and not only *locations* in a segment list. This is feasible, thanks to the huge IPv6 addressing space. Under this network programming model, the edge node can *program* a sequence of nodes to be crossed and the packet processing/forwarding behaviors to be executed by the nodes on the packet.

In section II we discuss how the IPv6 Segment Routing can be used to support SFC, what are the implications of using SR-unaware applications and the potential advantages of having SR-aware applications. As we will extend the open source Linux iptables firewall, section III provides a short introduction to its architecture. In section IV we analyze some design requirements and use case scenarios for the SR-aware applications, focusing on a firewall application. An important contribution here is the inclusion of a scenario in which some instructions to the firewall (e.g. actions to be executed on some class of packets) can be included in the segment list associated to a packet, without the need of reconfiguring the rules in the firewall running in the core of the NFV infrastructure and in line with the SRv6 network programming approach [10]. From the requirements, we design the architecture of the proposed SEgment Routing Aware (SERA) firewall, which extends the iptables firewall. In section V some details of the implemen-

This work has been partially supported by the Cisco University Research Program Fund

ISBN 978-3-903176-08-9 © 2018 IFIP

tation are given. To the best of our knowledge, the SERA firewall can be considered the first SRv6-aware application. Section VI provides the description of the testbed and the result of the performance evaluation. Based on these results, we have identified some shortcomings of the iptables design for our use cases. We implemented and evaluated a proof-of-concept that shows a significant performance improvement.

II. SFC based on IPv6 Segment Routing

Following the terminology defined in [4], the *SFC encapsulation* carries the information to identify the sequence of Service Functions (VNFs) that are required for processing a given packet. In the SRv6 approach considered here, the IPv6 Segment Routing Header (SRH) [7] contains such information. The SRH contains a *segment list*, a segment in this list identifies a VNF. Moreover, additional information related to the VNF chain can be carried in the optional Tag-Length-Value (TLV) section at the end of the SRH.

When a VNF that processes the packets is a legacy VNF, which is not aware of the Segment Routing based SFC encapsulation, we refer to it as an *SR-unaware* application. In this case an *SFC proxy* is needed, to remove the SFC encapsulation and deliver a clean IP packet to the SR-unaware application. Considering our focus on the Segment Routing based solution, we refer to the SFC proxy as *SR-proxy*. For the packets that are sent by the SR-unaware application, the SR proxy needs to (re)apply the SFC encapsulation after proper classification of the received packets. The operations of the SR proxies tend to be complex and in general they are not efficient. The main issue is that the information contained in the SFC encapsulation is removed from the packet when the packet is delivered to the SR-unaware application and may need to be re-added to the packet. This process typically requires a lot of state information to be configured in the classifier components of all nodes of a VNF chain and can consume a considerable amount of packet processing resources in the nodes.

Different types of VNFs can process the IP packets in a VNF chain. Some VNFs only need to inspect IP packets (e.g. DPI or network monitoring applications), other can drop or admit packets (e.g. firewall), other can modify IP and transport layer headers (e.g. NATs), other may need to terminate transport layer connections and reopen new ones (e.g. HTTP proxies, TCP optimizers). In general, the operations of an SR proxy depend on the type of VNF. If the VNF is not operating on the connections at transport layer (i.e. it is not modifying the 5-tuple of IP and transport layer source and destination addresses) it is possible in principle to re-classify the packets, at the price of repeating the flow-level classification in all nodes of the VNF chain. If the VNF is terminating / opening new transport level connections, it is not always possible to re-classify the packets and associate them to a specific chain.

As described in the SRv6 network programming document [10], the SR information can be added to a packet in two different modes, insert or encap. Figure 1 shows the original IPv6 packet and how it is carried in the two different encapsulation modes. In the insert mode the SRH header is inserted in the original IPv6 packet, immediately after the IPv6 header and before the transport level header. Note that the original IPv6 header is modified, in particular the IPv6 destination address is replaced with the IPv6 address of the first segment in the segment list, while the original IPv6 destination address is carried in the SRH header as the last segment of the segment list. In the encap mode the original IPv6 packet is carried as the inner packet of an IPv6-in-IPv6 encapsulated packet. The outer IPv6 packet carries the SRH header with the segment list.

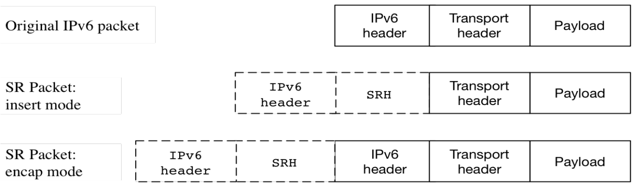

Fig. 1: SRv6 encapsulated packets

An SR-unaware application is not able to process the SRH information in the traffic it receives. An SR proxy is used to process the SRH information on behalf of the SR-unaware application. As discussed above, the behavior and the applicability of an SR proxy depend on the type of processing of the application. In [8], a set of behaviors of the SR-proxy have been defined, among them we mention:

- Static proxy
- Dynamic proxy
- Masquerading proxy

Both the Static and the Dynamic proxies support IPv6 SR packets encapsulated only in encap mode. They remove the SR information from packets before sending them to an SR-unaware application. These proxies receive back the packets from the SR-unaware application and reapply the SR encapsulation which includes the information on the VNF chain. They work under the assumption that the specific SR-unaware application running in the VNF is inserted in only one VNF chain, because all packets going out from the VNF are re-associated to the same chain. If multiple VNF chains needs to be supported, a different instance of the VNF is needed for each chain. The difference between the Static and the Dynamic proxies is that the SR information is statically configured in the Static case and is read from the incoming packets in the Dynamic case.

The Masquerading proxy supports SR packets encapsulated in insert mode. It masquerades SR packets before they are sent to an SR-unaware application, replacing the IPv6 destination address (which correspond to the current segment of the segment list) with the original IPv6 destination (i.e. the last segment in the segment list). When the packets are received back from the SR-unaware application, the Masquerading proxy retrieves the VNF chain information from the SRH header and changes the IPv6 destination address so that it reflects the current segment of the segment list. This process is refered to as *de-masquerading*. The assumption is that the SR-unaware application simply ignores the SRH header and that the SRH header is preserved in the processing. Moreover, this type of proxy can be only used with applications that do not change the packet headers and just inspect them.

Following the above discussion on the SR-unaware applications, we can state that their use in combination with SR proxies is conditioned by some constraints and characterized by high configuration complexity. It can also be affected by performance issues. Of course these problems will be faced and solved in practical use cases, considering the importance to support legacy SR-unaware applications in NFV deployments. On the other hand, in this work we take a more forward-looking approach and consider the design and development of SR-aware applications. Such applications are able to process the SFC encapsulation included in the IP packets, that is in our scenario the IPv6 SRH header that contains the segment list. The greatest benefit of using SR-aware applications is that the SR proxy is not needed and the SFC information carried in the SRH header is preserved when the packet is processed by the application. This approach avoids the need to maintain state information in the internal nodes. The configuration and management of the NFV infrastructure is simplified and the performance of the NFV enabled nodes is not affected by complex classification procedures. Moreover advanced features are possible by letting the applications interact with the SFC functionality offered by the network.

In this paper, we focus on the design and implementation of an SR-aware firewall application, but most of the design considerations have a more general applicability to other types of applications that can be deployed in SR based Service Function Chaining scenarios.

III. Linux iptables firewall

A firewall [11] essentially works according to a set of rules to accept or drop received packets. Each rule is composed of a condition and an action. The condition is based on set of attributes of received packets.

Once a packet satisfies the condition expressed by a rule condition, the associated action is performed on that packet. Iptables is a flexible and modular firewall and it is a standard component of most Linux distributions. It is built on top on the netfilter framework. In this section we provide a short tutorial on Iptables and netfilter architecture and implementation, which will be the base for the design of our solution.

A. Netfilter Framework

The netfilter framework [12] is a set of hooks in the packet traversal through the Linux protocol stack, which allows access to packets at different points. The current netfilter implementation provides five different hooks (*PREROUTING*, *INPUT*, *FORWARD*, *OUTPUT*, *POSTROUTING*) distributed along the receive and transmit path of packets as shown in Fig. 2. Kernel modules can register callback functions at any of these hooks. A callback function, after processing a packet, returns to the netfilter hook the action to be taken on the packet, such as DROP, ACCEPT, QUEUE (queue for user space processing).

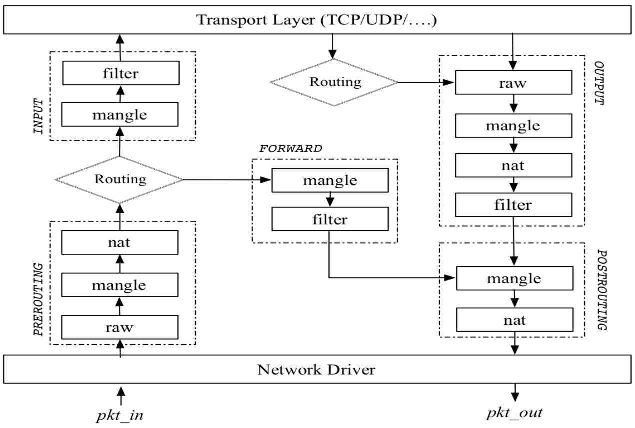

Fig. 2: Netfilter hooks and their associated tables

B. Iptables

Iptables represents the userspace implementation which allows access to the kernel-level netfilter framework hooks. It defines a set of rules that instruct the kernel what to do with packet coming to or traversing the protocol stack. The implementation of netfilter includes some pre-defined tables, as shown in Figure 2. Each table has a set of chains where iptables rules can be inserted. The supported tables are: filter (the default table, it contains rules that are used to filter IP packets); nat (mainly used to re-write the source and/or destination addresses of IP packets); mangle (a specialized table for mangling packet as they go through the kernel); raw (mainly used for connection tracking). Each iptables rule defines a set of matching criteria based on information from different layers of the protocol stack. Once the packet matches a rule, iptables takes an action on this packet. The standard actions are: ACCEPT, DROP, or QUEUE. Those correspond to the callback functions return values. Listing 1 shows examples of iptables rules.

C. Iptables extensions

The iptables framework is modular and extendible. New match extensions and target extensions can be developed separately and added to the iptables as new modules.

Listing 1: Examples of iptables rules

```
# Standard iptables rule
# Matches destination address of a packet
ip6tables -I INPUT -d fc00:d1::/64 -j DROP

# Extended iptables rule
# Matches destination address and hop-by-hop header
ip6tables -I INPUT -d fc00:d1::/64 \
-m hbh --hbh-len 40 -j DROP
```

Match extensions are used to add more matching options to iptables. They can be used alone or in combination with the default match options. They provide the ability to have sophisticated iptables rules in order to look deeper into IP packets. e.g., hbh, which matches the parameters in IPv6 Hop-by-Hop extensions header. An example of extended iptables rule is shown in Listing 1.

Target extensions are new actions added to the default ones of iptables. A new iptables target usually performs an action different from the default ones (ACCEPT, DROP, etc.,). It can be used for logging/profiling or it can modify the packet before returning it back to the netfilter framework. Destination NAT (DNAT) is an example of iptables target extension, which is used to modify the destination address of a packet.

IV. SEGMENT ROUTING AWARE (SERA) FIREWALL

In an SRv6 SFC scenario, the VNFs are deployed over the servers of the NFV infrastructure. The Segment Routing Header (SRH) is added to packets to enforce a VNF chain, i.e. the sequence of VNFs to be crossed by the packets. The SR-unaware applications rely on the SR-proxy that removes the SRH from the packet. On the other hand, the SR-aware applications are capable of processing the SR information in the packets. We focus on a specific type of SR-aware applications, namely a firewall application. In this section, we start by analyzing some design requirements and use case scenarios for the SR-aware applications. The following considerations are focused on a firewall application, but they have a more general value as they can be applied to similar applications that needs to be deployed on an SR based SFC environment (e.g. DPI, IDSs). We assume that an SR-aware firewall should support two working modes: *basic* mode and *advanced* mode.

In the basic mode the SR-aware firewall must be able to work as a legacy firewall, but with no need of the SR-proxy. In particular, the SR-aware firewall should be able to use the same set of rules defined for the legacy firewall and apply them directly to the SFC encapsulated packets that carry the SRH information. It must be able to handle SR packets encapsulated in encap as well as insert modes and logically apply the rules to the original packets rather than to the SFC encapsulated packets. To make a concrete example, if an existing rule includes a condition on the source IPv6 address and the original IPv6 packet has been encapsulated in (IPv6-in-IPv6) it makes no sense to consider the IPv6 source address of the received packet as the condition should be checked on the source address of the packet. The use case scenario is to virtualize the legacy firewalls, executing them in servers on the NFV infrastructure, without changing the legacy rules and with no need of SR-proxy functionality.

In the advanced mode the SR-aware firewall should support rules with extended conditions that can explicitly include attributes not only from the original packet but also from the SRH and the outer packet. In particular, the SR-aware firewall could leverage SRv6 SID arguments, TLVs, or TAG. It could also apply differentiated processing based on the active SRv6 SID (i.e., apply different rule sets for different SIDs). As for the actions, in the advanced mode the SR-aware firewall should be able to support some SR-specific actions. For example, an SR-specific action could be to skip the next SID in the segment list, so that it is possible to operate a "branching" instead of the usual linear exploration of the VNF chain, when some conditions on the packet are met. A use case scenario for this feature is to consider a service chain which includes a firewall followed by an Intrusion Detection System and allow skipping the IDS for a subset of traffic that matches some conditions. A further requirement is that the SR-aware firewall application should be able to select the actions to be performed based on information contained in the SID. This is aligned with the SRv6 network programming approach of minimizing the state information maintained in the nodes and storing explicit state information in the packets. The use case scenario in this case is that instead of re-configuring some firewall rules in a specific firewall running in the core of the NFV infrastructure, it is possible to obtain the same result by changing a SID in the SID list that is injected to the packet in the edge node. The big advantage is that the reconfiguration is only needed in the edge node, which in any case has to manage per-flow state to perform the classification operations.

In the following subsections we propose the architecture of the SERA firewall (for basic and advanced modes) that meets the above requirements, extending the Linux iptables.

A. SERA basic mode

In the basic mode, SERA is an SR-aware firewall that can apply the normal firewall processing to the original packets even if they have an SR based SFC encapsulation. The proposed packet processing architecture is shown in Figure 3. Each received packet goes through an *SR pre-processor* that splits traffic into SR and non-SR traffic. Non-SR traffic is processed as in an SR-unaware firewall, as represented with the solid-line path in Figure 3. SR traffic follows a different path through the firewall, represented with double-line path in Figure 3. In this path, the firewall evaluates the defined rules on the original packet, properly taking into account the impact of the SR encapsulation. It supports both encap and insert mode, which implies that the original IPv6 source and destination information of received packets may be encoded differently as follows:

- Encap mode: original source and destination are the ones of the packet.
- Insert mode: packets have only one IPv6 header. The original source information is in the source address of the IPv6 header, while the original destination is encoded as the last SID in the SRH.

The *Inner match* functional block is responsible for getting the original source and destination information from SR packets and compare them to the defined rules. Once a packet hits a condition of a rule, the associated standard action (ACCEPT, DROP, etc.) is triggered on that packet.

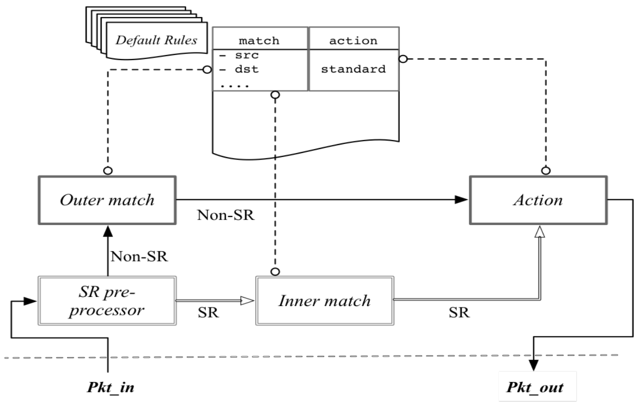

Fig. 3: Architecture of basic mode SERA

B. SERA advanced mode

In the advanced mode, SERA extends the iptables capabilities by offering new matching capabilities and new SR-specific actions. It introduces new iptables rules (SERA rules) that have extended conditions involving attributes from outer packet, inner packet, and the SRH header. The architecture of advanced mode (Figure 4) is defined incrementally with respect to the basic mode (Figure 3), by adding the *SRH match* functional block and replacing the *Action* block with the *Extended Action* block. Since the matching could be performed on both the original and the outer packet headers, the SR traffic follows a more complex path, as shown in Figure 4. Unlike in the basic mode SERA, all received packet are first processed by the *Outer match* block, which applies parts of the extended rules on the outer packet. The *SR pre-processor* does the same job as in the basic mode SERA by splitting traffic into non-SR and SR traffic. Non-SR traffic goes directly to the Action functional block while SR traffic is directed to the the *Inner match* block. The Inner match block works as in the basic mode, but the rules that drive its behavior are written in a different way. For example, with an extended rule it is possible to match on the outer source and destination IPv6 addresses (denoted as `src`, `dst`) and on the original ones (denoted as `inner-src`, `inner-dst`). The Inner match block takes care of the matching of the inner source and destination (the ones of the original packet). The *SRH match* block is concerned with the matching between SRH extension part of the rules and the SRH of received SR packets. Finally, each packet (SR or non-SR) that satisfies the matching condition of a rule goes to the *Extended Action* block. It extends the Action block present in the architecture of the Basic mode by allowing the introduction of *SR-specific* actions in addition to the standard ones.

An SR-specific action is an advanced action that can be applied to SR-encapsulated packets. It may modify or process SR-encapsulated packets based on SRH information. We list here some examples of SR-specific actions, but the set of these actions can be extended to cover more complex SFC use-cases.

- `seg6-go-next`: the default action of the SEG6 target. It is similar to the Endpoint function from the SRv6 network programming model [10]. It sends packets towards the next SID from SRH. The `seg6-go-next` serves as an ACCEPT action for SRv6 encapsulated packets.
- `seg6-skip-next`: it instructs the SERA firewall to skip the next SID in the SRH.
- `seg6-go-last`: it instructs the SERA firewall to skip the remaining part of the segment list and process the last segment.
- `seg6-eval-args`: the generic action that supports SRH programmed actions.

Following the traditional iptables model, the above defined SR-specific actions are included in *statically* configured rules which are executed in a SERA firewall running as a VNF. Taking into account the concepts of the SRv6 programming model, we have designed a more dynamic approach, which allows to define the action to be executed as a result of a match on a packet by packet basis, by putting information in the Segment IDentifier (SID). For this purpose, a special SR-specific action is defined, called `eval-args`. It does not represent a concrete action, but instructs the SERA firewall to look into the current SID to find the action to be executed. As described in [10], an SRv6 local SID is an IPv6 address that can be logically split into three fields: LOC:FUNCT:ARGS. LOC uses the L most significant bits, ARGS the R rightmost-bits and FUNCT the remaining $128 - (L + R)$ bits in the middle. In our case, the LOC part is used as a locator to forward the packets to the NFV node that runs the firewall, and it is advertised by the routing protocols. The FUNCT part identifies a specific VNF on the NFV node (in our case the SERA firewall instance). The ARGS part may contain information required by the VNF and may even change on a per-packet basis. Note that the ARGS part will be ignored in most cases (or omitted setting R=0), whenever there is no need to carry additional information in the SID. To give an example, the LOC field can be 64 bits long and uniquely identify an NFV node. This leaves 128-64=64 bits for the identification of the VNF in the NFV node and for the arguments if needed.

In the advanced mode of SERA it is possible to use the ARGS part of the SID to encode a firewall action to be executed in case of match. This requires that a set of rules with action `eval-args` is configured in the SERA firewall. For all packets that match one of these rules, the action to be executed is contained in the ARGS field of the SID. The advantage of this approach is that it is possible to (re)configure the action to be executed on a given subset of packets by operating at the network edge, with no need to update the configuration of the SERA firewall instance running in the core of the NFV infrastructure.

V. Implementation

We implemented the SERA firewall as an extension of Linux iptables described in section III.

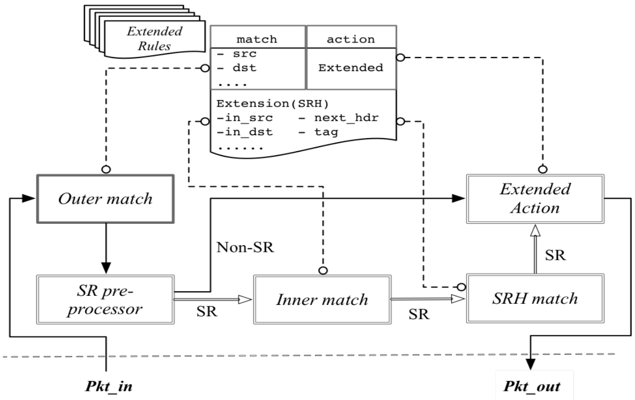

Fig. 4: Architecture of advanced mode SERA

A. Implementation of basic mode SERA

In the Linux kernel the `ip6_tables` module is responsible for checking the iptables rules against the received packets. It implements the `ip6_packet_match()` function that evaluates the defined iptables rules against the outermost IPv6 header of a received packet. In order to implement the basic mode of the SERA firewall, we extended the existing `ip6_tables` module to operate according to the architecture shown in Figure 3. We added the SR pre-processor block. The SR packets are forwarded to the *Inner match* functional block, implemented in the `inner_match()` function, which evaluates iptables rules against the original packet. It supports SR packets encapsulated in both encap and insert mode.

We added a new sysctl parameter (`ip6t_seg6`) to switch between legacy iptables mode and SERA basic mode. The system administrator can enable the SERA basic mode on the fly with the command: `sysctl -w net.ipv6.ip6t_seg6=1`, which activates the SR pre-processor.

We have realized a first version of basic mode SERA that implements only a subset of the normal classification rules, namely those involving the IP src and destination addresses. On this first version we have performed the evaluation that is reported in the paper. Then we have implemented a second version that supports all the classification rules and it is now available at [13].

B. Implementation of advanced mode SERA

We implemented the advanced mode SERA by exploiting the iptables extension features. We added a new match extension as well as a new target extension to the iptables implementation both at kernel and user-space levels. Thanks to these extensions it is possible to match on the SRH fields, this allow to have a full control on where the packets is directed (the next SIDs) and which nodes it has crossed before.

At kernel level, we implemented two additional kernel modules: the `ip6t_srh` as match extension and `ip6t_SEG6` as target extension. The `ip6t_srh` module implements the *SR pre-processor*, the *Inner match*, and the *SRH match* from the advanced SERA architecture. The `ip6t_SEG6` module implements the *Extended Action*. It is a new target (SEG6) for iptables rules that supports a set of SR-specific actions.

To support the advanced mode SERA at user-space level, we extended the iptables user-space utility with two new shared libraries: `libip6t_srh` and `libip6t_SEG6`. They allow the iptables user to define SERA rules. These rules can have attributes from outer packet, inner packet, and SRH. List. 2 shows a list of match options supported by the `libip6t_srh` extension.

The `ibip6t_SEG6` extension supports the new SR (SEG6) target with some SR-specific actions (shown in List. 3). For SRH programmed actions, we introduced a new sysctl variable (`ip6t_seg6_args`) that defines the number of rightmost bits in the active SID to be used as ARGS. The `SEG6` target decodes the ARGS bits to decide which action should be taken on the packet. If the decoded value does not correspond to any of the supported actions, SERA will send back an ICMP Parameter Problem message point to the active SID. Such ICMP message can be used to understand which actions are supported by the firewall.

Listing 2: Options of srh match extension

```
#ip6tables -m srh -h
 srh match options:
 [!] --inner-src      addr[/mask]  Inner packet src
 [!] --inner-dst      addr[/mask]  Inner packet dst
 [!] --srh-next-hdr   next-hdr     SRH Next Header
 [!] --srh-len-eq     hdr_len      SRH Hdr Ext Len
 [!] --srh-len-gt     hdr_len      SRH Hdr Ext Len
 [!] --srh-len-lt     hdr_len      SRH Hdr Ext Len
 [!] --srh-segs-eq    segs_left    SRH Segments Left
 [!] --srh-segs-gt    segs_left    SRH Segments Left
 [!] --srh-segs-lt    segs_left    SRH Segments Left
 [!] --srh-last-eq    last_entry   SRH Last Entry
 [!] --srh-last-gt    last_entry   SRH Last Entry
 [!] --srh-last-lt    last_entry   SRH Last Entry
 [!] --srh-tag        tag          SRH Tag
 [!] --srh-psid       addr[/mask]  SRH previous SID
 [!] --srh-nsid       addr[/mask]  SRH next SID
```

Listing 3: Options of SEG6 target extension

```
#ip6tables -j SEG6 -h
 SEG6 target options:
 [--seg6-action  action]
 Valid SEG6 actions:
 seg6-go-next       SEG6 go next
 seg6-skip-next     SEG6 skip next
 seg6-go-last       SEG6 go last
 seg6-eval-args     SEG6 eval args
```

VI. Performance Evaluation

A. Testbed description

In order to verify the correctness of SERA implementation and to evaluate the performance aspects, we designed a testbed environment that can be easily replicated, shown in

Fig. 5. For the experiments described in this section, we have deployed the testbed on CloudLab [14]. Cloudlab is a flexible infrastructure dedicated to scientific research on the future of cloud computing. Our testbed is composed of three identical nodes. Each node is a bare metal server with Intel Xeon E5-2630 v3 processor with 16 cores (hyper-threaded) clocked at 2.40GHz, 128 GB of RAM and two Intel 82599ES 10-Gigabit network interface cards. The three nodes are Linux servers and respectively represent an ingress node, NFV node, and egress node of an SRv6 based SFC scenario. The links between any two nodes X and Y are assigned IPv6 addresses in the form `fc00:xy::x/64` and `fc00:xy::y/64`. For example, the two interfaces of the link between the ingress node (node 1) and the NFV node (node 2) are assigned the addresses `fc00:12::1/64` and `fc00:12::2/64`. Each node owns an IPv6 prefix to be used for SRv6 local SID allocation. The prefix is in the form `fc00:n::/64`, where `n` represents the node number. For example, the NFV node (node 2) owns the IPv6 prefix `fc00:2::/64`. SRv6 local SIDs are in form LOC:FUNCT:ARGS, where LOC is the most significant 64-bits, ARGS is rightmost 16-bits and FUNCT is the 48-bits in between LOC and ARGS. The ingress node is used as a source for SR encapsulated traffic. The NFV node runs the SERA firewall inside a network namespace. The SERA firewall is instantiated on the SRv6 local SID `fc00:2::f1:0/112`. We have two destination servers $d1$ and $d2$ that are used as traffic sinks. Each destination server is assigned a prefix in the form `fc00:dn::/64`, where `n` is the destination server number. We configured the ingress node with two different SR SFC policies as shown in Listing 4. The first SR SFC policy is used to encapsulate traffic destined to `d1` as SR packets in encap mode, while the second one encapsulates traffic destined to `d2` as SR packets in insert mode. The SRv6 SFC policies are used to steer traffic through the SERA firewall, then to the egress node which removes SR encapsulation from packets as they leave the SR domain towards destinations (`d1` and `d2`). The ingress and egress nodes are running Linux kernel 4.14 [15] and have the 4.14 release of *iproute2* [16] installed. The NFV node runs a compiled Linux kernel 4.15-rc2 with SRv6 enabled and SERA firewall included [17]. In order to saturate the CPU of the NFV node, we used only one processor core for processing all the received packets by disabling the *irqbalance* service and assigning the *IRQ* for all interfaces to be served by the same CPU core. We used *iperf* [18] to generate traffic on the ingress node. All traffic generated by *iperf* goes through the SRv6 SFC policies configured on the ingress node.

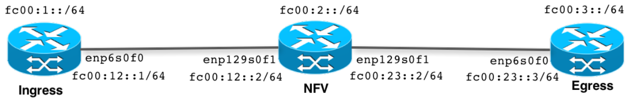

Fig. 5: Performance evaluation testbed.

Listing 4: SR SFC policy

```
# SR SFC policy - encap mode
ip -6 route add fc00:d1::/64 encap seg6 mode \
encap segs fc00:2::f1:0,fc00:3::d6 dev enp6s0f0

# SR SFC policy - insert mode
ip -6 route add fc00:d2::/64 encap seg6 mode \
inline segs fc00:2::f1:0,fc00:3::d6  dev enp6s0f0
```

Listing 5: SR pre-processor implementation

```
static inline bool
sr6_pre_processor(const struct sk_buff *skb,
                  int *innoff,int *srhoff,int *encap)
{
    /* SRv6 traffic (encap mode) detector
    if (ipv6_find_hdr(skb, innoff, IPPROTO_IPV6,
                      NULL, NULL) > 0)){
        *encap=1;
        return true;
    }
    /* SRv6 traffic (insert mode) detector */
    if (ipv6_find_hdr(skb, srhoff,IPPROTO_ROUTING,
                      NULL, NULL) > 0)
        return true;
    return false;
}
```

B. Measurements

In order to evaluate the performance of our implementation, we generated SR traffic with a rate of 1 Mpps (10^6 packets per second). Each packet has a payload size of 1 KB. We wanted to measure the processing capacity (or throughput) of the firewall in processed packets per second (pps). We configured iptables with a rule that drops all traffic going from ingress node towards the destinations. Therefore, the counter of this rule represents the number of SR packets that the firewall has been able to process. In order to evaluate the performance for different numbers of rules, we add a sequence of N-1 non-matching rules before the matching rule. In particular, we repeated each experiment for ten different number of rules N from 1 to 512. Each value plotted in Figures 6-9 represents the average of 30 runs, each run with duration of 60 seconds. The confidence intervals are so close to the average that we have not plotted them.

We conducted five experiments as follows:

- Exp. 1: default iptables on plain IP packets.
- Exp. 2: basic mode SERA with SR encap mode.
- Exp. 3: basic mode SERA with SR insert mode.
- Exp. 4: advanced mode SERA with SR encap mode.
- Exp. 5: advanced mode SERA with SR insert mode.

In experiment 1 (default iptables), we used a rule that matches the IPv6 source and destination address of the received packets. The non-matching rules have the same structure, but different source and destination addresses. With only one rule configured (N=1), the throughput is 911 Kpps. As expected, the achieved throughput decreases with the number

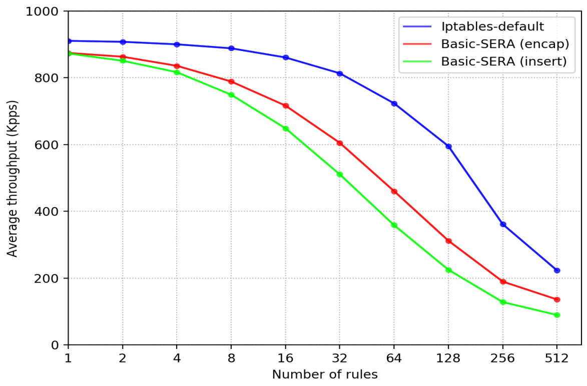

Fig. 6: Basic SERA vs. default iptables

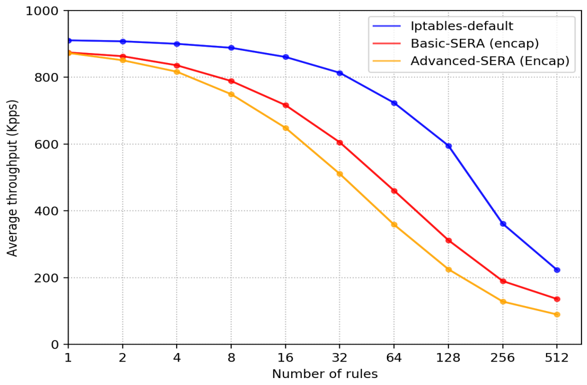

Fig. 7: Basic SERA vs. advanced SERA (encap mode)

of rules, as shown in Figure 6. This is due to the operations that are executed for each rule. In particular, the function `ip6_packet_match()` is called for each rule.

In experiments 2 and 3, we evaluate the throughput of basic mode SERA with the same rules as the ones in the experiment 1 (matching the source and destination address). In these experiments, we are considering SR encapsulated packets and we set the `ip6t_seg6` sysctl to apply the rule to the original packets. When there is only the matching rule ($N = 1$) the throughput is 875 Kpps in encap mode and 873 Kpps in insert mode (Figure 6). For larger N, the degradation of the performance is more evident. The performance reduction of basic SERA with respect to iptables default is due to the *SR pre-processor* functional block, whose implementation is reported in Listing 5. This block has the task to look for the inner IPv6 header in the packet or for the SRH header in case of insert mode (we have re-used the `ipv6_find_hdr` function used by iptables). These operations are computationally expensive and are the reason for the reduction of the throughput visible in Figure 6. According to the design philosophy of iptables, the *SR pre-processor* is executed once for each rule, because each rule operates in a stateless way and no state related to the packet is saved. From a performance point of view, this is clearly not efficient. Therefore, in order to improve the throughput result shown in Figure 6 we are considering alternate design choices which can achieve higher performance when a large number of rules may need to be applied to the packets. The insert mode has lower throughput than the encap mode due to our implementation of the *SR pre-processor* block, which detects SR packets in encap mode before those in insert mode (Listing 5). We decided to add the encap mode detection before the insert mode since it works also for *IPv6-in-IPv6* tunnels.

In experiments 4 and 5, we evaluated the throughput of advanced mode SERA. We considered an extended rule that matches source and destination address from both inner and outer packet. The results are similar to the basic mode SERA, the throughput is 857 Kpps in encap mode and 849 Kpps in insert mode when one rule is configured ($N = 1$) and the performance degradation with respect to the default iptables is higher when the number of rules N increases (the Figure is not reported for space reasons). In Figure 7, we compare the throughput of basic and advanced mode SERA, considering the SR packets in encap mode. Both in the basic and in the advanced mode the *SR pre-processor* is executed once for each rule, the advanced mode SERA achieves a lower throughput because it has to perform two match operations (Inner and Outer) rather than a single one.

We wanted to verify that the throughput reduction when several rules per packet are executed was not caused by problems in our implementation. Hence, we conducted a new experiment using an already existing iptables extension, the Routing Header extension (implemented in `ip6t_rt` kernel module). This extension is able to match the common fields of the IPv6 Routing Header, including the Routing Type field (but is not able to parse the content of the SRH header, for which we have developed the proposed extension). We run the test for the different numbers of rules N as in the previous experiments. For matching we used an extended rule that drops packets with Routing Type 4, i.e. the SR packets with the SRH header. As shown in Figure 8 the obtained throughput perfectly matches our SERA implementation, confirming that the poor performance is inherently related to the iptables design.

Finally, we tackled the issue of performance degradation and we were able to design and implement a solution focusing on one specific scenario, the basic mode SERA operating on SR packets encapsulated in encap mode. In this scenario, a set of existing rules needs to be applied to the original packets that are encapsulated with IPv6-in-IPv6. As shown in Figure 6, there is a performance penalty which becomes significant when the number of rules is large. We revised the design of our iptables extension so that we can execute the SR pre-processor once for each packet instead of re-executing it for every rule. The idea is to modify the pointers that point to the memory area in which the headers of the packet is stored once before executing all the rules and then to properly keep into account these modifications in the processing of the results of the matching. The throughput measurements of the revised design are shown in Figure 9. Only in case of a single rule, the throughput is slightly reduced do to the operations that are performed once for the packet. When the number of rules increases, there is no throughput degradation as for the basic mode SERA, and the performance approaches the one of the default iptables operating on plain (not encapsulated) IPv6 packets.

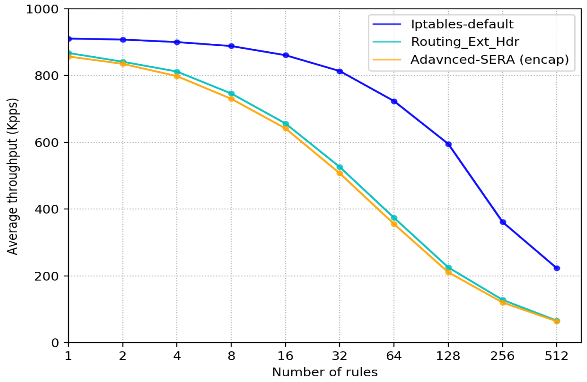

Fig. 8: Existing RH iptables extension vs. advanced SERA

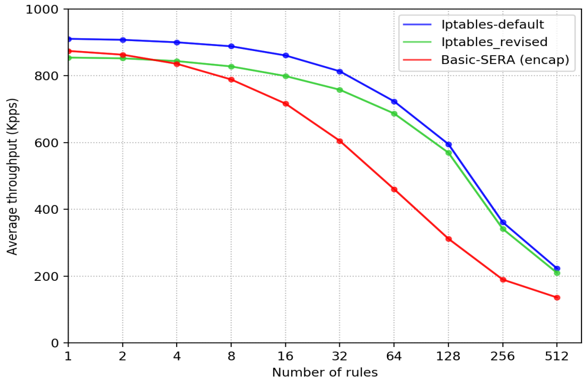

Fig. 9: Revised iptables design vs. Basic SERA

VII. Conclusions

In this work we have shown that it is possible to modify an existing application (the Linux iptables firewall) and make it Segment Routing aware. Thanks to this awareness, it is possible to setup chains of VNFs in a simple and efficient way, with no need of SR proxy. In the basic mode, the proposed SERA firewall solution avoids the need of (re)classification of packets in the intermediate NFV nodes that host the SR-aware firewall. In the advanced mode, new firewall actions can operate on the SR segment list, allowing to make branches in the VNF chain. We have also described how it is possible for the edge node to put instructions in the SR segment list, which can dynamically change the firewall actions that will be executed. This can be done by the edge node even on a packet-by-packet basis. In this way the firewall VNF could become stateless so that it can be scaled, replicated, moved arbitrarily in the NFV infrastructure.

From the performance analysis of the SERA implementation we have highlighted a throughput degradation when the number of rules to be checked for each packet increases. This is due to the iptables design that operates in a stateless way and repeats all operations per each rule. We have implemented a proof-of-concept that overcomes this issue in a specific scenario, showing the performance gain that can be obtained.

We provided an open source implementation for SERA [17]. We submitted our implementation to the the Linux kernel and a part of it has been merged into version 4.16 [13]. We also contributed to the *netfilter.org* project to extend the iptables user-space utility to support the new match options and SR-specific actions (part of our work is in release 1.6.2 of iptables [19]).

References

[1] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, “Network function virtualization: State-of-the-art and research challenges,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[2] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, “A survey on service function chaining,” *Journal of Network and Computer Applications*, vol. 75, pp. 138–155, 2016.

[3] P. Quinn and T. Nadeau, “Problem Statement for Service Function Chaining,” Internet Requests for Comments, RFC Editor, RFC 7498, April 2015.

[4] J. Halpern and C. Pignataro, “Service Function Chaining (SFC) Architecture,” Internet Requests for Comments, RFC Editor, RFC 7665, October 2015.

[5] P. Quinn, U. Elzur, and C. Pignataro, “Network Service Header (NSH),” Internet-Draft, November 2017. [Online]. Available: http://tools.ietf.org/html/draft-ietf-sfc-nsh

[6] A. AbdelSalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, and L. Veltri, “Implementation of Virtual Network Function Chaining through Segment Routing in a Linux-based NFV Infrastructure,” in *3rd IEEE Conference on Network Softwarization (NetSoft 2017), Bologna, Italy*, July 2017.

[7] S. Previdi (ed.) et al., “IPv6 Segment Routing Header (SRH),” Internet-Draft, September 2016. [Online]. Available: http://tools.ietf.org/html/draft-ietf-6man-segment-routing-header-02

[8] F. Clad *et al.*, “Segment Routing for Service Chaining,” Internet-Draft, October 2017. [Online]. Available: https://tools.ietf.org/html/draft-clad-spring-segment-routing-service-chaining-00

[9] J. Brzozowski et al., “IPv6 SPRING Use Cases,” Internet-Draft, December 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-spring-ipv6-use-cases

[10] C. Fisfils *et al.*, “SRv6 Network Programming,” Internet-Draft, March 2017. [Online]. Available: https://tools.ietf.org/html/draft-filsfils-spring-srv6-network-programming-04

[11] J. R. Vacca and S. Ellis, *Firewalls: Jump start for Network and Systems Administrators*. Elsevier, 2005.

[12] R. Russell and H. Welte, “Linux netfilter Hacking Howto.” [Online]. Available: http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html

[13] “Linux community. linux 4.16 changelog,” Web site. [Online]. Available: https://kernelnewbies.org/Linux_4.16

[14] “CloudLab home page,” Web site. [Online]. Available: https://www.cloudlab.us/

[15] “Kernel 4.14 release,” Web site. [Online]. Available: https://kernelnewbies.org/Linux_4.14

[16] “iproute2 4.14 release,” Web site. [Online]. Available: https://mirrors.edge.kernel.org/pub/linux/utils/net/iproute2/

[17] “SERA - SEgment Routing Aware Firewall,” Web site. [Online]. Available: https://github.com/SRouting/SERA

[18] “iPerf - The ultimate speed test tool for TCP, UDP and SCTP,” Web site. [Online]. Available: http://iperf.fr

[19] “iptables releases. iptables-1.6.2 changelog,” Web site, February 2018. [Online]. Available: https://netfilter.org/projects/iptables/downloads.html#iptables-1.6.2