AINFV: Analysis of Isolation (memory/packet) in Network Function Virtualization

Abdul Ahad Ayaz*,
*Paderborn University (ahad@mail.upb.de)

***Abstract*—NFV (Network Function Virtualization) is a new way of defining a network with the help of software that was previously done with hardware middle-boxes. But there is isolation and performance overhead between NFV and hardware middle-boxes. The current approach is run network function in either VM or Container, this ensures isolation but at the cost of performance degradation. In this seminar paper, the NetBricks framework is introduced. This framework provides a new way of developing and running the network function. This paper further discussed isolation issues and performance overheads. A detailed comparison is provided between using the NetBricks framework and VMs/Containers.**

I. Introduction

The starting days of networks, it was used to designed to send packets between two nodes. As the size of the network increased, technology evolved and many network services were introduced from time to time i.e. routing, forwarding, security, etc. Traditionally these network services were deployed using hardware middleboxes i.e. firewall, routers, etc. These traditional hardware middleboxes are in the market for a very long time and still serving their purposes. But there are disadvantages of using this approach, such as mentioned in [12]: •inflexibility: unable to modify the network services, proprietary issue;•Non-Scalability: one needs to buy the new middlebox if the load of the network increases for a certain period of time and the load stabilizes after some time, then the new middlebox is useless; and •Cost: expensive in terms of upgrading the network components by replacing old middleboxes with latest middleboxes to get the maximum throughput. These disadvantages encouraged the ETSI [4] (European Telecommunication Standards Institute) in 2012 and the idea of NFV (Network Function Virtualization) was proposed. The idea was to replace the hardware middleboxes with software-defined network services and deploy these network services as VM (Virtual Machine) on commodity servers. ETSI proposed that NFV will help the service providers as •swift deployment of network services; •comparatively cheap, by using the commodity servers; •more flexibility, upgrading of network service is in software. As mentioned in [2] NFV provides the blueprint of developing the network's data plane, which allows the developer to program every packet forwarding in the network. Same in SDN (Software Defined Networking), which provides the blueprint of managing the control plane, i.e. allows a developer to define the custom routing, managing network failures, etc. NFV framework provides the following features [2]:

1) Multiplexing: NFV framework should ensure that the NF (Network Functions) should be hardware independent, this helps in scaling of NF without changing the hardware.

2) Isolation: NF deployed in virtualized share the under the underlying hardware, NFV framework should ensure the memory and packet isolation without affecting the performance

3) High Performance: NF connected in series working as NF chains should have maximum throughput or equal to as of hardware middleboxes. NFV framework should ensure this throughput, as there is a major overhead of copying packets from one NF to others.

4) Efficiency: Framework should ensure minimal hardware utilization as the aim of NFV is to utilize the commodity servers efficiently.

5) Simplify NF Development: Framework should ensure the simplicity in development of NF, by separating the tasks into two categories i.e. user-defined functionality and preprocessing tasks. All of this should be automated.

6) Rapid Deployment: Framework should ensure the rapid deployment by production ready NFs (i.e. NF testing and deployment in the production environment on the go, to improve the performance). This saves a lot of time.

Problem Statment

NFV framework has many advantages but these frameworks are still a long way from perfection in terms of development and deployment. For the development part as addressed in [Noval approach] main issue is the performance trade-off due to low-level programming and optimization issues. Isolation. No standard model is defined, thus every vendor has its programming model making NF operation complex to work in a multi-tenant network environment. For deployment, the current idea is to deploy NFs as VMs or Containers to give isolation as it is the main security concern. But at the cost of performance loss. The main idea is to deploy the NFs as a process instead of VMs or Containers.

II. Background

Requirements

As discussed earlier, NFV purpose is to simplify the development and deployment of NFs without changing the functionality and performance offered by traditional middleboxes. As mentioned in [2], some requirements must be fulfilled:

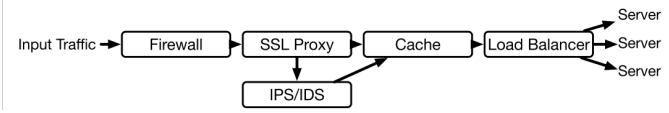

Fig. 1: NF chain for serving web traffic [2]

1) Performance: Framework should not take more than 10 seconds of a microsecond for processing packet. Single NF should be able to process 10-100Gbps of traffic. As the mentioned figures are equivalent to what we get with hardware middleboxes.

2) Efficiency: Deployment should be done using a single machine because deployment across multiple machines will result in poor resource utilization and performance loss.

3) Chaining: Framework should be able to combine multiple NF called chaining i.e. NF1 to NF2...till NFn. Packet processing starts from NF1 to NFn. Fig1 shows the NF chain for processing web traffic.

4) Multi-vendor: NFV framework should support the multi-vendor NF to exists in a network, with security measures i.e. isolation.

5) Multi-tenant: In the cloud environment multiple tenants exist, sharing the virtual resource provided by the service provider. NF should be deployed in such a way that the deployment for one tenant should not affect the operation of other tenants.

Mentioned requirements help in building well structured NFs and deployment ensuring the isolation. To get further insights into the NFV framework, NFV is divided into two parts, one part deals with the development model and the second part deals with the execution model.

A. Development Model

Throughput and latency are two major metrics affecting network performance. Throughput is packet processing in a given time whereas latency is time between sending and receiving of a packet. These two metrics depend on the number of things [12]i.e. context switching and copying, network card to cache copying, TLB (Translation Lookaside Buffer) misses and memory allocation. There are many libraries available for fast packet processing namely DPDK [5] and netmap etc. As mentioned in [2] DPDK (Data Plane Development Kit) libraries provide fast packet processing mechanism by •Using PMD (Poll-mode Driver) instead of depending on the CPU interrupts for acknowledgment of received packet; • assigning NIC (Network Interface Card) to single process instead of relying on kernel for NIC multiplexing; • provides the interface for connecting NIC directly to NF, instead of using intermediate elements (i.e. vSwitch) that required additional computation for packet movement. These libraries help in improving performance and developers to focus on optimization (i.e. how the packets should be batched). The use of vectorization, as proposed in [12], VPP (Vector Packet Processing) allows the processing on vectors of packets (i.e. up to 256 packets can be read at once.) As discussed in [2] The Click modular router [19] based on DPDK libraries, provides the abstraction to develop new NF in such a way by combining multiple packet processing elements. But does not define how packets flow between different elements. Click [9] provides the limited functionality for customization. Hence for every new NF, developers have to re-write those elements from scratch, a lot of time is spent on optimizing the elements. Development model should be modular, some module with fixed functionality and common for all NFs, whereas other modules should be user-defined for specific functionality. A developer is responsible for optimizing the user-defined modules only.

B. Execution Model

Current practice is to deploy NFs in VMs or Containers and for communication vSwitch is used. VMs and Containers ensure the memory isolation (i.e. operation on one NF will not affect the other NF in a network). vSwitch allows the NFs to periodically use the NIC for sending and receiving of packets in networks or between NFs. But all this processing of packets is just copying of packets in network and every NF has its copy of packets that violates the packet isolation (i.e. at any point in time, only one NF should have access to that particular packet) and considerable hard to achieve. Above mentioned technologies have a greater influence on performance degradation. As mentioned in [1], comparing the single process with dedicated NIC, per-core throughput drops by 3x when processing 64B packets using Containers and up to 7x while using VMs. This performance degraded furthermore when NFs are chained, Containers are 7x slower compared to NF chained in a single process and VMs are up to 11x slower. Furthermore, NF chained single process is 6x faster than NF chained Containers or VMs, where each NF having its dedicated core.

C. Isolation

Both packet and memory Isolation is the major challenge to achieve, as it directly affects the performance (i.e. latency and throughput). The main reason for this performance gap is: Firstly during packet processing, packets tend to cross the memory isolation barrier. Secondly the use of context switch that ensures that packet should cross core boundaries. These isolation issues can be catered as mentioned in [1]: for memory isolation, instead of using VMs and Containers. Checks can be introduced both compile-time and run-time. For packet isolation, “ZCSI (Zero-Copy Software Isolation)” is proposed by the author in [1] that ensures the “safe 0-copy” packet I/O between network functions. This is implemented using unique types [20]. To address the issues of development and execution model, the author in [1] proposed a solution called NetBricks. This solution showcased a different way of developing and executing network functions that contradict with the traditional approach.

III. Proposed Framework

Overview

NetBricks, a framework for developing and executing network functions on a single machine. It requires the re-writing of network function compatible with NetBricks development model. As mentioned in [1], it is not a limitation because of two reasons: • not enough development progress has been done for network functions; • NetBricks can also co-exists with the traditional network functions, at the cost of performance. The proposed framework provides both a development and execution environment. For the development model, it helps developers to work on the high-level abstraction of packet processing tasks and allows user-defined programmability. The execution model uses safe language and runtimes to ensure memory isolation, whereas the current approach uses scheduling for performance isolation [1]. Another important aspect to consider in the execution model is communication between network functions. Message passing [inter-process communication] must not be modified by network function to ensure packet isolation. To achieve this functionality, NetBricks uses "static checks" to avoid packet copying. The author in [1] named this functionality as ZCSI (Zero-Copy Software Isolation). ZCSI allows achieving the memory and packet isolation as compared to VMs and Containers with no performance degradation.

As discussed above, NetBricks is a complete package. In the section below, development model is explained in detailed, later section describes the execution model to deploy the network functions developed using development model

A. Main Components of Development Model

As described in [1], NetBricks allows the developer to focus on the high-level programmability of network function. Network function programmability is divided into five sections: packet processing, bytestream processing, control flow, state management, and event scheduling.

1) Packet Processing: In NetBricks, packets structure consist of (a) stack of header; (b) the payload; (c) reference to any per-packet metadata [1]. The header contains a structure that defines the length of a packet based on the functional computation of its contents. The payload is actual data carried by the packet. Metadata defines the internal communication within network function and it is customizable by the developer using user-defined functions. These user-defined functions are passed along with header structure and can access last deciphered header along with payload and related metadata. At the start, the header stack contains a "null" value, occupying zero byte space. The author provided the four packet operations as follows [1]:

Parse: This operation takes the header type and structure as an input. Later analyzes the payload accordingly by using header type and update the header stack. At the end header bytes are removed from the payload. thesis

Deparse: This operation is applied on header stack, it removes the bottom header from the stack and returns it to the payload.

Transform: This operation implements the user-defined functions on header and payload, allows developer to modify the packet size (i.e. by adding or removing bytes to payload as mentioned in "parse"). It also allows to add and modify the metadata of the packet.

Filter: This operation is used to remove packets to be dropped at a specific node. It is a boolean operation return either True or False. Filter operation is based on user-defined and it drops all the packets at the specific node when the user-defined function returns the false value.

2) Bytestream Processing: The main function of bytestream processing is to convert the bytes arrays into packets. User-defined functions are applied on the bytes arrays, In [1], the author provides the two bytestream operations as follows:

Window: This operation takes four parameters as input i.e. window size, sliding increment, timeout, and a stream user-defined function. This operation is responsible for receiving and re-arranging the cached packets and create a stream. A user-defined function is called whenever there is enough data received to form a window of appropriate size or connection is closed or the timeout expires. Window operation can also forward all received packets without modifying them or it can drop all the packets and generate the modified byte array using packetize node.

Packetize: This operation allows the conversion of byte arrays into packets. Providing the byte arrays and header stack, packetize converts and the data into packets and assign the relevant header. For the Implementation of the operations as mentioned above author uses the TCP (i.e. TCP sequence number for re-arranging, FIN packets to check connection closing and packetize operation on a header by modifying the relevant header fields) [1].

3) Control Flow: Control flow deals with the branching required in network function chains. Branching is used to define the conditions i.e. re-routing the packets to specific port etc. Another purpose of branching is to move packets across cores for processing. To get the maximum performance, there should be minimum caching of data between cores. Control flow provides the developer the abstraction for re-routing the packets as desired i.e. by user-defined functions, port, address, etc. As mentioned above, control flow branching is useful while implementing the NF chains, it allows the developer to select which packet should be routed to the next network function. The author provides the three operations for control flow [1] as follows:

Group By: This operation allows the branching with-in NF and branching across NF chains. It takes two input: the number of groups for packet re-routing and user-defined function returning the packets with the ID of a group to which it belongs. The author also provided some pre-defined grouping functions based on criterion i.e. TCP flow.

Shuffle: This operation adds additional functionality to "Group By" operation i.e. branching is done based on cores. At "Runtime", Group ID generated by shuffle is used to decide which core to be used for packet processing. Shuffle allows both user-defined and pre-defined grouping. The main point

to consider is group id generated by the shuffle is not known at the "Compile time".

Merge: This operation provides a junction, where all the different branches can be merged i.e. all packets from different branches entering a junction and exist as a single group.

4) State Management: When data is processed across multiple cores, performance degradation can be observed. Due to communication between core i.e. cache coherence etc. Typically Developer program the network function to partition state and avoid cross-core access or allow minimal access when required without using partition state. NetBrick's state management allows access across multiple cores. Within core accesses are synchronized but for cross-core accesses author proposed following options [1]: • no-external-access i.e. one core for each partition; • bounded inconsistency [2] i.e. where one core has write access to partition andSimple other cores only have read access; • strict consistency i.e. allows multi-read and multi-write access.

5) Event Scheduling: Event Scheduling allows the developer to create user-defined functions, that can be run repeatedly i.e. to monitor the NF and get the performance logs periodically, etc.

B. Main Components of Execution Model

NetBricks provides the execution environment to run the network function. This model ensures isolation and also deals with network function placement and scheduling [1].

Isolation:

1) Memory Isolation: Traditionally isolation is obtained by using VMs and Containers, at the cost of performance loss for simple network function. Considering the complex network function, this performance loss dominates the other factors. To tackle these performance degradations, NetBricks used a different approach to achieve isolation. It makes use of RUST [6] safe language that ensures the type checks and LLVM [21] as a runtime. This combination of safe language and runtime achieves the memory isolation similar to that is obtained using the hardware MMU (Memory Management Unit). As mentioned in [1], safe language and runtime ensures the following: • disallow pointer arithmetic; • bound checking on array accesses i.e. preventing random memory accesses; • disallow accesses to null object i.e. preventing undefined behavior to ensure memory isolation; • type casts are safe. These above features can be achieved using high-level programming languages i.e. Java, C#, etc, but these languages are not system friendly.

2) Packet Isolation: The traditional mechanism is to send packets in a physical network, NFV follows the same footstep. Network function sent packets in the network by copying. This copying mechanism results in performance degradation of packet processing. NetBricks uses the unique types [20] instead of using the copying mechanism. As mentioned in [1], unique types are used to cater data races i.e. disallowing the simultaneous accesses to the same data. This approach is applied while implementing the network functions when network function sends a packet, sender function losses the accesses to that packet and only relevant network function should have accesses to that packet. This ensures packet isolation without any copying of packets.

Above mentioned techniques used in NetBricks for isolation are referred to as ZCSI (Zero-Copy Soft Isolation) [1]. NetBricks runs as a single process, that can be assigned to one or more cores for processing and use one or more NICs for packets I/O. Packets are transferred between network functions using "function calls", In the case of network function chains, a queue is maintained at the receiving end.

Placement and Scheduling: As mentioned before NetBricks operates as a single process in which many network functions can be run. Consider the network function chains running as a directed graph having access to multiple available NIC interfaces. Before implementing, NetBricks have to decide at which core the network chains should be run. Based on this NetBricks make the scheduling decisions for packet processing. As the author mentioned in [1], currently NetBricks places all the network functions on a single core to get the maximum performance. For more complex placements, NetBricks make use of "shuffle" operation to process packets across multiple cores. In [1], author uses the "run-to-completion" scheduling for NetBricks i.e. packets entering NF, it starts processing till it exists. Scheduling is needed when dealing with more than one packet, considering this all packets are added to "window" operation i.e. receiving and re-arranging till enough packets have been collected and "group by" operation i.e. stack up packets to be processed by branch. NetBricks uses "round-robin" scheduling for these operations.

C. Testing

To test and validate the NetBricks, two network functions were re-programmed [1]. Firstly, the NF that decrements the IP TTL (Time To Live) and drops the packet with TTL zero. Secondly, google's Maglev [3] (i.e. load balancer).

For the first example network function is build using NetBricks as shown in fig.2. Network functions are described as a public function in the RUST module. Line 1 of fig.2 shows how the new instance of network function is created using the "ttl_nf" function. During the network function processing, at line 3 ethernet (MAC) header is parsed from the packet and later at line 4 IP header is parsed from the packet. "Transform" operation is applied on the parsed IP header decrementing the packet's TTL (line 5). After the computation of packet's TTL, "Filter" operation is performed that drops the packets having TTL zero (line 9). In the end "Compose" operation at line 12 indicate the end of the description of network function and also allows the chaining of network function. As shown in fig.2, no "Shuffle" operation is applied while defining the network function. By default, NetBricks routes all the packet processing to a single core. Fig.3 shows the code that is used to execute the user-defined functions. "NetbricksContext" is used to execute the user configuration. As mentioned in [1], at line 4 pipeline is created that : (a) receives packets from input queue; (b) these packets are forwarded to network function i.e. "ttl_nf" for processing; (c) later processed packets are again to

```
pub fn ttl_nf<T: 'static + NbNode>(input: T)
                          -> CompositionNode {
  input.parse::<MacHeader>()
       .parse::<IpHeader>()
       .transform(box |pkt| {
            let ttl = pkt.hdr().ttl() - 1;
            pkt.mut_hdr().set_ttl(ttl);
       })
       .filter(box |pkt| {
            pkt.hdr().ttl() != 0
       })
       .compose()
}
```

Fig. 2: NetBricks code for Example NF [1]

```
// cfg is configuration including
// the set of ports to use.
let ctx = NetbricksContext::from_cfg(cfg);
ctx.queues.map(|p| ttl_nf(p).send(p));
```

Fig. 3: NetBricks code for Executing Example NF [1]

the same queue. Placement of the pipeline is defined in user configuration.

For the second example, the author used load balancer (Maglev) network function and tried to implement it using NetBricks. Maglev is responsible for dividing the user requests among the back-end servers. Maglev ensures that: it can be deployed in a replicated cluster for scalability and fault tolerance: splits traffic; and handles failures. Maglev uses a hashing algorithm (i.e. based on a lookup table) to achieve the aims mentioned above [1]. Fig.4 shows the packet processing and forwarding part of Maglev. First, the lookup table is created (line 8) and then a cache for recording the backend (line12) that is used to start the instance of Maglev network function. Starting at line 15, the network function is declared. Second, the "Shuffle" operation (line 16) uses the built-in functionality of using a single core. At line 17 ethernet headers are being parsed. Later "Group by" operation (line18) is performed, it uses the "ipv4_flow_hash" built-in function to extract the flow hash i.e. consist of IP header and TCP or UDP header. This hash is used for two purposes: (a) ensuring that the received packet is TCP or UDP ; (b) to find the already assigned backend to flow (line 24) or to assign new backend to flow using a lookup table (line 25). In the end, network functions generate the vector of nodes relevant to the backend, specified by the operation.

D. Analysis Tool

NetBricks network function chains are build using RUST language and mechanism proposed by the author. That is in contradiction with currently available frameworks i.e. Open-MANO etc. These frameworks provide the developer with an interface to manage without knowing the underlying programmability used for building the network functions. Author

```
pub fn maglev_nf<T: 'static + NbNode>(
                input: T
                backends: &[str],
                ctx: nb_ctx,
                lut_size: usize)
                -> Vec<CompositionNode> {
  let backend_ct = backends.len();
  let lookup_table =
      Maglev::new_lut(ctx,
            backends,
            lut_size);
  let mut flow_cache =
    BoundedConsistencyMap::<usize, usize>::new();

  let groups =
    input.shuffle(BuiltInShuffle::flow)
         .parse::<MacHeader>()
         .group_by(backend_ct, ctx,
              box move |pkt| {
                let hash =
                  ipv4_flow_hash(pkt, 0);
                let backend_group =
                   flow_cache.entry(hash)
                   .or_insert_with(|| {
                  lookup_table.lookup(hash)});
                backend_group
     });
    groups.iter().map(|g| g.compose()).collect()
```

Fig. 4: NetBricks code for google's Magvel [1]

in [1] also proposed to use the same interface, as it provides the following advantages: (a) provides many optimization opportunities, currently using the RUST compiler's optimization for optimizing the chaining code, later LLVM's link-time optimization [21] can be used to optimize the whole program, improving performance of packet processing; (b) it can be used to execute the complex network function chains and branches.

IV. Framework Evaluation

A. Testbed

For evaluating the NetBricks, as mentioned in a [1] testbed with the following configuration was used: dual-socket servers with Intel Xeon E5-2660 2.6GHz CPUs, each having 10 cores. Each having 120GB of RAM, divided between sockets. Each server was also equipped with Intel XL710 QDA2 40Gb NIC. Hyper-threading was disabled and hardware virtualization was enabled i.e. Intel VT. Servers running Linux kernel 4.6.0-1 and DPDK 16.04 and programming language RUST nightly version. Two virtual switches were used i.e. first Open Vswitch with DPDK (OVS DPDK) [23] and second SoftNIC [22]. VMs used were running on KVM connected to a virtual switch with DPDK's "vhost-user" driver. As per the requirement of DPDK, Docker containers were used with privileged mode and for their connection to a virtual switch, DPDK's ring PMD driver was used. By default, PMD driver does not allow both virtual switches to copy packets and no packet isolation was observed as network function can modify the packets after they sent it. To achieve isolation the author made few modifications in virtual switches, allowing copying while connecting to containers. Even this copying using DPDK's PMD driver has

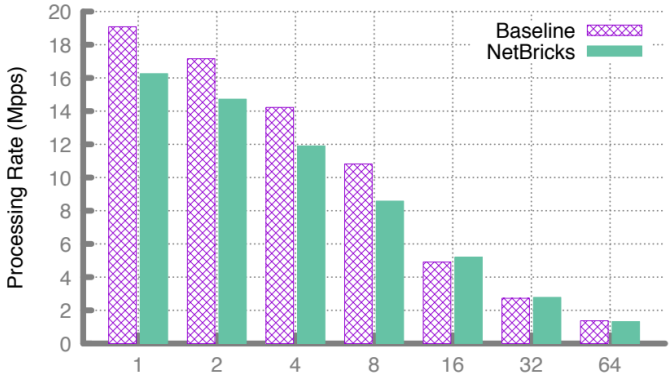

Fig. 5: Throughput comparison between NetBricks NF and Basline NF

[1]

higher performance than other approaches i.e. "veth" pair. To test the traffic, the DPDK-based packet generator was used running on a separate server having 40Gb NIC. This server was directly connected to the test server without any intermediate switches. This packet generator server acted as a source and sink. Performance (i.e. throughput and latency) results were collected at the sink.

B. Overheads and Results

This section describes the overheads imposed by the NetBricks and the results obtained besides these overheads comparing to baseline network functions build using other frameworks. For comparison purposes, both NIC and DPDK were configured the same for NetBricks and baseline network function. In [1], the author mentioned few overheads that are as follows: for simple network function; (b) for checking array bounds.

In the case of simple network function, the packet TTL network function was used as shown in fig.2 and explained in the section above. Both NetBricks version and normal version of network function were executed using a single core by send packets of 64 bytes and observed the throughput. The result of both network functions was almost the same as expected by the author. As mentioned in [1], after 10 experimental runs, the average throughput observed for baseline network function was 23.3 MPPS (million packets per second) whereas for NetBricks network function it was observed 23.2 MPPS. For latency at 80%, the RTT (round trip time) for baseline observed was 16.15 microseconds compared to NetBricks was 16.16 microseconds.

In the case of array accesses [1], the safe language was used that imposed some overheads i.e. array bound checking. These checks can be a major source of the overhead of safe language (i.e. Null-checks and other safety checks performed at runtime are difficult to separate). To test these checks overhead, NF was used that updates several cells in a 512KB array during packet processing. The update of these array cells depends on a UDP source port number of the packet under processing, making it difficult to ignore array bound checks.

NF	NetBricks	Baseline
Firewall	0.66	0.093
NAT	8.52	2.7
Sig, Matching	2.62	0.983
Monitor	5	1.785

Fig. 6: Performance figures of NetBricks NF w.r.t other NFs [1]

The author compared the NetBricks NF with a baseline NF executed using a single-core and used packets with random UDP source port. Fig.5 graph shows the throughput achieved by two network functions when memory accesses per packet increased. NetBrick's throughput is 20% less as compared to the baseline NF for 1 to 8 memory accesses. As the number of accesses increased i.e. 16 or more, the effect of checks overhead started to fade out. This is because of large memory accesses causing cache misses resulting in reduced throughput. These cache misses dominating the overhead imposed by the NetBricks.

C. Performance analysis of framework based different NFs

To further validate the development model of NetBricks, the author implemented different categories of network functions using the NetBricks framework. Some of them are mentioned below [1]:

Firewall: based on Click [9], performs the linear scan of an ACL (Access Control List) to find the relevant entry.
NAT: based on MazuNAT [ref] built using Click.
IDS: based on Snort [24], for signature matching.
Monitor: maintains per-flow counter similar to monitor module of Click.
Load balancer: Maglev used, already described in the above section.

Testing of the above-mentioned network functions was done using both NetBricks framework and original network function. These network functions were executed on a single core. Comparison results can be seen in fig.6, NetBricks performance is better as compared to other frameworks. For example, as mentioned in [1] NetBricks NAT has a 3x better performance than MazuNAT.

Further experiments were performed on the NetBricks version of Maglev to be tested on multi-core. Fig.7 show the comparison of NetBricks version of Maglev with google's Maglev. It was observed that NetBricks throughput is 2.9x to 3.5x better than the results mentioned in [3]. Average latency observed was 19.9 microsecond for NetBricks and 32 microseconds for the original Maglev. These throughput and latency figures are better but not to rely on because these two experiments (i.e. NetBrick's Maglev version and Google's Maglev) were performed on two different test-bed.

# of Cores	NetBricks Impl.	Reported
1	9.2	2.6
2	16.7	5.7
3	24.5	8.2
4	32.24	10.3

Fig. 7: Throughput achieved comparing NetBricks Multi-core with Maglev

[1]

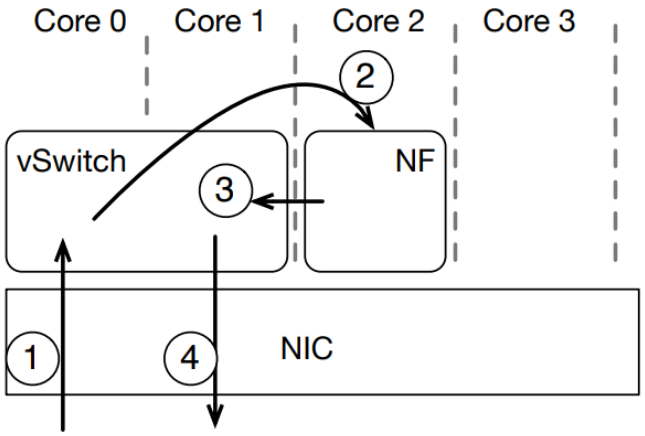

Fig. 8: VM/Container execution environment for simple NF

[1]

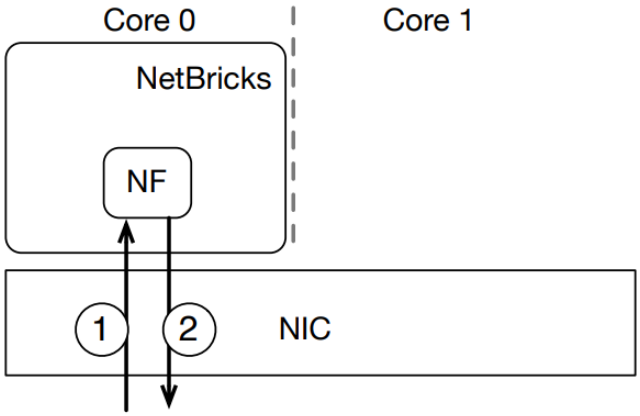

Fig. 9: NetBricks execution environment for simple NF

[1]

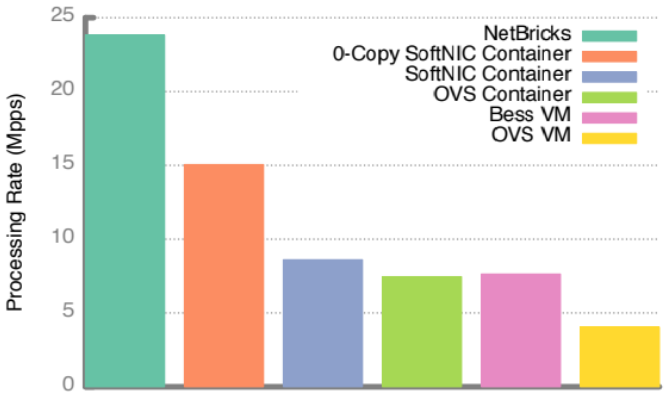

Fig. 10: Throughput achieved using different isolation techniques

[1]

D. Isolation Analysis

Isolation is ensured by NetBrick's safe language and run-time checks to avoid costs associated with core boundaries and crossing process [1]. The author first checked the cost involved with the single network function. Evaluation was done to check the results when the length of the packet TTL NF chain increases. One point to consider here is that these costs are only applicable for simple network functions, but when the computational cost of network function is higher then NetBricks execution environment becomes irrelevant.

NF vs NF Chains vs Complex NF: This section presents the analysis for single NF, NF chain and complex NF in term of isolation and for that author compared the NetBricks with VMs or Containers [1].

First case, single NF (i.e. "that swaps the source and destination ethernet address for received packets and forward them out the same port") built using NetBricks and other with C language were compared and executed in their execution environment i.e. VM, Container, and NetBricks. Fig.8 shows the test environment of VM and Container. A vSwitch is used that receives packets from NIC and forward them to network function running in either VM or Container. Network function process those packets and forward them back to vSwitch, which is then forwarded to NIC. Both vSwitch and network function runs on DPDK and depend upon polling. For better performance network function is assigned its dedicated CPU and two cores are assigned to vSwitch. As mentioned in [1], with isolation comes two kinds of overheads: first due to cache and context switching cost (i.e. cost associated with cross process or crossing core boundaries) and second due to the copying of packets. To analyze these isolation overheads, the author used SoftNIC to send packets between Containers without copying (i.e. 0-copy SoftNIC Container) violating packet isolation, later compared it with NetBricks. As shown in fig.9, NetBricks receives packets directly from NIC, process them using network function code and send them back to NIC, all this process runs on a single-core. Fig.10 graph shows the throughput achieved using different isolation techniques. As discussed in [1] Comparison of 0-copy SoftNIC Container and NetBricks concerning throughput shows that 0-copy SoftNIC Container is 1.6x slower then NetBricks due to crossing core boundaries. Even though NetBricks was running on single-core and other was using three cores for processing. SoftNIC Container is 2.7x slower than NetBricks

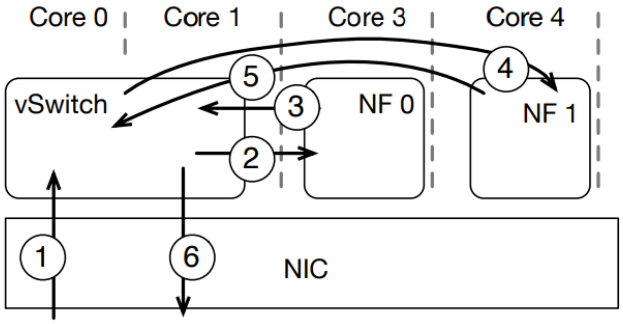

Fig. 11: VM/Container Execution environemt for NF chain [1]

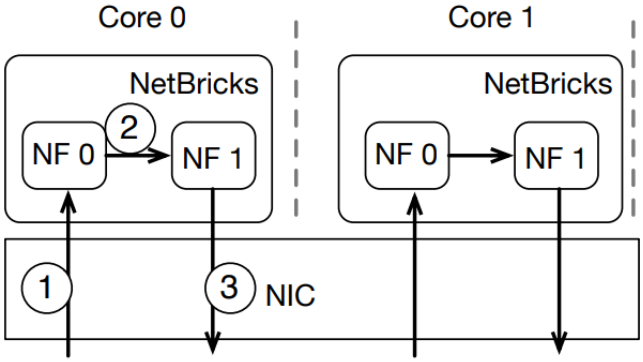

Fig. 12: NetBricks Execution environemt for NF chain [1]

due to packet copying. This performance further degrades when using VM instead of Containers. Because VM uses the "vhost_user" communication channel developed by DPDK from interacting with VM has more performance issues as compared to DPDK's ring PMD driver used for Container.

Second Case, NF chain (i.e. multiple instances of single network function, compute packet's TTL (time to live) and drops the packet with TTL 0). Fig.11 shows the execution environment using VM or Container to run the network function chain (i.e. NF0 and NF1). Assigned two cores to vSwitch and two cores for network function chain, one for each network function. Fig.12 shows the execution environment of NetBricks to run the network function chain (i.e. NF0 and NF1). NetBricks was tested for two cases: (a) with single-core; (b) with two-cores. Fig.13 shows the throughput comparison of NetBricks with single-core, multi-core, VM and Container. As mentioned in [1], NetBricks multi-core (NB-MC) is 7x better than a container with SoftNIC and 11x better compared to VM with SoftNIC. NetBricks with single-core (NB-SC) is 4x better than a container with SoftNIC and 6x better compared to VM with SoftNIC. The author also compared the 0-copy SoftNIC container and observed that with a packet size of 64 Bytes, it resulted in 3x times performance loss. As shown in fig.13, NetBricks multi-core throughput decreased when the size of the chain increased from 4. The author explained this decrease in throughput is due to more cores tries to accesses the NIC

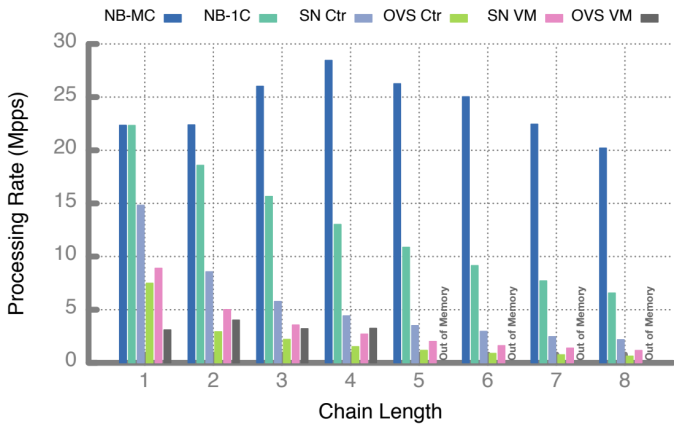

Fig. 13: Throughput of increasing NF chain, NetBricks single and multi core vs other technologies [1]

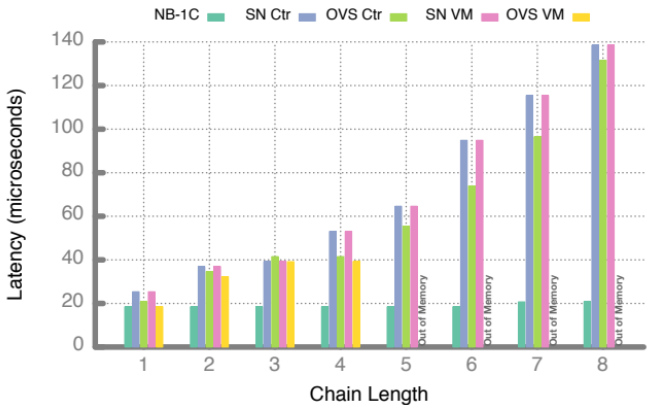

Fig. 14: Latency for increasing NF chain, NetBricks single/multi-core vs other technologies [1]

(i.e. 4 parallel I/O threads). Above throughput figures are for a packet size of 64 Bytes. Increasing the packet size degrades the performance by 15% [1]. Fig.14 shows the latency when using different isolation techniques.

Third case, complex NF (i.e. increases computation cycles for per-packet processing) and used the same execution environment as the first case (i.e. fig.8 and 9). Three core for VM and Container (i.e. one for network function and two for vSwitch) and in case NetBriciks two cases: single-core and three-cores. The author modified the network function to use busy loops for the number of cycles after packet processing. Fig.12 shows the throughput comparison of per-packet processing by using different isolation techniques. As mentioned in [1], increasing complexity of network function results in increased computation time that overcomes the improvements offered by NetBricks (i.e. 300 cycles per packet). As shown in fig.15, NetBricks isolation performs better as compared to other techniques (i.e. VM or Container).

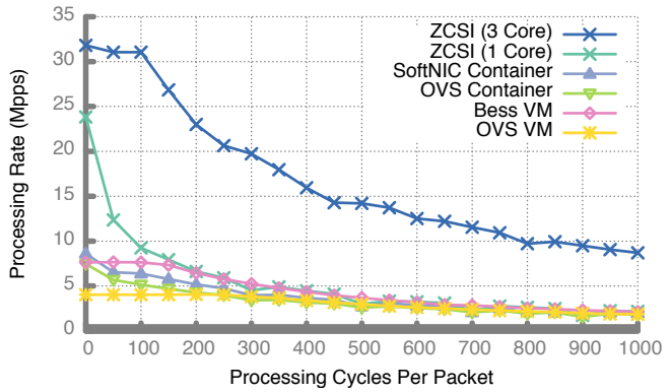

Fig. 15: Throughput of Complex NF using different isolation techniques

[1]

V. Comparison of Frameworks

In this section, NetBricks is compared with other frameworks available in the market. A comparison is done based on the development model, execution model, and isolation.

A. Development Model

In terms of development model, there are many frameworks available such as YANFF [12] helps in rapid network function development. It is based on DPDK and uses GO language that is high-level, safe with built-in support for concurrency. YANFF uses the scheduler for packet processing across multiple cores, whereas in NetBricks it is done by explicit "shuffle" operations. YANFF has no dependencies on the execution environment. Next is libVNF [13], a reusable library for developing high performance and scalable network functions. An API is used, that provides the high-level abstraction and manages low-level optimization. It separates the application-specific processing of network functions from the software stack that can be reusable or common for all network functions. Compared to NetBricks, libVNF supports clustered network functions deployment and horizontal scalability (i.e. ensuring high availability). FLICK [25] framework for development and deployment of network functions. FLICK structure is quite similar to NetBricks, both use their own programming languages for development and have their own execution environment to run the developed network functions. FLICK language allows to focus on only application-specific logic ignoring low-level details. Developed network function is compiled using the FLICK compiler that converts the FLICK program into C++, which is then compiled and linked with FLICK runtime for execution.

B. Execution Model

In terms of the execution model, NetVM [8] is a virtualization-based platform that uses shared memory to exploit the DPDK library (i.e. ensuring 0-copy between VMs and to VMs). It uses the hypervisor-based switch to control the flow of packets and inter-VM communication. Also, ensure isolation to some extent i.e. packet accesses to only trusted VMs. OpenNetVM [17] is based on the NetVM framework, it uses Docker Container instead of VM. OpenNetVM consists of two main components: first, NF Manager that interact with NICs and responsible for packet flow management and inter-container communication. Second NFlib API that is used to connect network function (i.e. container) to NF Manager, it also allows the development of user-defined network function and makes use of NFlib to connect with NF Manager. HyperNF [15] is VM based framework, fully utilizing the available resources (i.e. CPU cores). No dedicated cores assigned for packet I/O as per the merge model [15] and packet I/O are part of VM. It uses "hypercall" instead of packet forwarding and communication between VMs. Compared to NetBricks, HyperNF is based on standard NFV architecture whereas NetBricks completely rewrites the software middleboxes. G-NET [18] framework is based on GPU-virtualization. It consists of three main components: "Switch" is a virtual switch for packet I/O and forwarding packets between network functions; "Manager" is proxy for GPU, it receives a request from network function for GPU computation; "Scheduler" allocates the GPU resources. G-NET also ensure the data isolation by using "isoPointer".

Framework	Memory Isolation	Packet Isolation	Overheads
xOMB	X	X	Low (function call)
CoMB	X	X	Low (function call)
NetVM	Y	X	Very high (VM)
ClickOS	Y	Y	High (lightweight VM)
HyperSwitch	Y	Y	Very high (VM)
mSwitch	Y	Y	Very high (VM)
NetBricks	Y	Y	Low (function call)

Fig. 16: Isolation between different frameworks [1]

C. Isolation

In terms of Isolation, SafeBricks [16] framework is based on NetBricks with some modifications. It is more oriented to cloud security. It ensures only encrypted traffic is exposed to the cloud. SafeBricks used the concept of "hardware enclave" using Intel SGX [ref] (i.e. ensures that software even kernel or hypervisor outside the enclave cannot tamper with enclave). It allows the network function to run inside the enclave and in the cloud, it seems like a black box. Fig.16 shows the author's comparison of the different framework (i.e. xOMB [7], CoMB [26], ClickOS [9], HyperSwitch [10], mSwitch [11]) in context with isolation.

VI. Future Work

Future work of the NetBricks framework should be optimizing the RUST language to remove the overhead of the NetBricks framework. Currently, it supports multi-core processing, but further optimizations can be done to get even better throughput. Maintain the active development of new network functions and provide support. Currently, NetBricks works on the data plane and control plane functionality should

be added in the future to get better performance. Lastly, the integration with MANO systems.

REFERENCES

[1] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV." in *OSDI'16*, 2016, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[2] A. Panda, "A New Approach to Network Function Virtualization," Tech. Rep., 2017. [Online]. Available: https://cloudfront.escholarship.org/dist/prd/content/qt638687x3/qt638687x3.pdf?t=p2rjwy

[3] H. Yaghoubi, N. Barazi, and M. Reza, "Maglev: A Fast and Reliable Software Network Load Balancer ," in *Infrastructure Design, Signalling and Security in Railway*, 2012, pp. 523–535. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

[4] ETSI, "Network Functions Virtualisation,An Introduction, Benefits, Enablers, Challenges & Call for Action," Tech. Rep. 1, 2012. [Online]. Available: http://portal.etsi.org/NFV/NFV_White_Paper.pdf

[5] I. Corporation, "Intel ® Data Plane Development Kit," no. June, 2014. [Online]. Available: https://www.dpdk.org/

[6] The Rust Team, "Rust Programming Language," p. 329, 2016. [Online]. Available: https://www.rust-lang.org/

[7] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. 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*. ACM Press, 2012, p. 49. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2396556.2396566

[8] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, mar 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7036139/

[9] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, F. Huici, and I. Nsdi, "ClickOS and the Art of Network Function Virtualization," pp. 459–473, 2014. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[10] "Hyper-Switch: A Scalable Software Virtual Switching Architecture," *Atc '13*, pp. 13–24, 2013. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/ram

[11] M. Honda, F. Huici, G. Lettieri, and L. Rizzo, "mSwitch: a highly-scalable, modular software switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research - SOSR '15*. ACM Press, 2015, pp. 1–13. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2774993.2775065

[12] I. Philippov and A. Melik-Adamyan, "Novel approach to network function development," in *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia on - CEE-SECR '17*. ACM Press, 2017, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3166094.3166111

[13] P. Naik, A. Kanase, T. Patel, and M. Vutukuru, "libVNF: A Framework for Building Scalable High Performance Virtual Network Functions," in *Proceedings of the 8th Asia-Pacific Workshop on Systems - APSys '17*. ACM Press, 2017, pp. 212–224. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3124680.3124728

[14] J. Duan, X. Yi, J. Wang, C. Wu, and F. Le, "NetStar: A Future/Promise Framework for Asynchronous Network Functions," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 600–612, mar 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8635508/

[15] K. Yasukata, F. Huici, V. Maffione, G. Lettieri, and M. Honda, "HyperNF: building a high performance, high utilization and fair NFV platform," pp. 157–169, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3127479.3127489

[16] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "SafeBricks: Shielding Network Functions in the Cloud," pp. 201–216, 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/poddar

[17] M. Yurchenko, P. Cody, A. Coplan, R. Kennedy, T. Wood, and K. K. Ramakrishnan, "OpenNetVM: A Platform for High Performance Network Service Chains," in *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*. ACM, 2018, pp. 1–2. [Online]. Available: https://dl.acm.org/citation.cfm?id=2940155

[18] "G-NET : Effective GPU Sharing in NFV Systems," 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/zhang-kai

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactio@bookGordon, ns on Computer Systems*, vol. 18, no. 3, pp. 263–297, aug 2000. [Online]. Available: http://portal.acm.org/citation.cfm?doid=354871.354874

[20] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy, *Uniqueness and Reference Immutability for Safe Parallelism*. [Online]. Available: http://delivery.acm.org/10.1145/2390000/2384619/p21-gordon.pdf?ip=131.234.217.85{&}id=2384619{&}acc=ACTIVESERVICE{&}key=2BA2C432AB83DA15.FF86995C7D80A64D.4D4702B0C3E38B35.4D4702B0C3E38B35{&}{_}{_}acm{_}{_}=1555440985{_}bd2f52b0356eb72dfbe525781e0ad38f

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program anailysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, pp. 75–86. [Online]. Available: http://ieeexplore.ieee.org/document/1281665/

[22] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[23] J. Gross, "The Evolution of Open vSwitch LinuxCon 2014," Tech. Rep., 2014. [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/EvolutionofOVS.pdf

[24] M. Roesch, "Snort-Lightweight Intrusion Detection for Networks," Tech. Rep. [Online]. Available: https://www.usenix.org/legacy/event/lisa99/full{_}papers/roesch/roesch.pdf

[25] A. Alim, R. G. Clegg, L. Mai, L. Rupprecht, E. Seckler, P. Costa, P. Pietzuch, A. L. Wolf, N. Sultana, J. Crowcroft, A. Madhavapeddy, A. W. Moore, R. Mortier, M. Koleni, L. Oviedo, M. Migliavacca, and D. McAuley, "FLICK: Developing and running application-specific network services," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/alim

[26] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and Implementation of a Consolidated Middlebox Architecture," Tech. Rep. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final96.pdf