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

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

***Abstract*—Network Function Virtualization (NFV) 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 to run network function in either Virtual machine or Container, this ensures isolation but at the cost of performance degradation. In this seminar paper, NetBricks framework is discussed. 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 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 e.g. 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 as mentioned in [1]: a) Inflexibility: unable to modify network services (proprietary issue); b) Non-Scalability: one needs to buy new middlebox if the load of network increases for a certain period of time and load stabilizes after some time then the new middlebox is useless; c) 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 European Telecommunication Standards Institute (ETSI) [2] in 2012 and the idea of Network Function Virtualization (NFV) was proposed. The idea was to replace the hardware middleboxes with software-defined network services and deploy these network services as Virtual Machine (VM) on commodity servers. ETSI proposed that NFV will help the service providers with swift deployment of network services comparatively cheap and more flexible. NFV framework provides the following features [3]:

- Multiplexing: NFV framework should ensure that Network Functions (NF) should be hardware independent. This helps in scaling of NF without changing the hardware.
- Isolation: NF deployed in virtualized environment shares the underlying hardware. NFV framework should ensure the memory and packet isolation without affecting the performance.
- High Performance: NF connected in series working as NF chains should have maximum throughput or equal to hardware middleboxes. NFV framework should ensure this throughput, as there is a major overhead of copying packets from one NF to others.
- Efficiency: Framework should ensure minimal hardware utilization as the aim of NFV is to utilize the commodity servers efficiently.
- 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 pre-processing tasks. All of this should be automated.
- Rapid Deployment: NFV Framework should ensure the rapid deployment by production ready NFs (e.g. NF testing and deployment in the production environment on the go, to improve the performance). This saves a lot of time.

NFV framework have many advantages but it is still long way from perfection in terms of development and deployment. For development part as addressed in [1] main issue is the performance trade-off due to low-level programming and optimization issues. 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 ensure isolation as it is the main security concern. But at the cost of loss in performance loss. The main idea of this seminar paper is to ensure isolation without using VMs/Containers and deploy the NFs as a single process using NetBricks framework [4] mentioned in sectionIII.

II. Background

This section provides the background of traditional practices used for developing and deploying the NFs. To get the further insights, these practices are divided into development model, execution model and isolation.

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 [1]i.e. context switching, Translation Lookaside Buffer (TLB) misses and memory allocation. There are many libraries available for fast packet processing namely Data

Plane Development Kit (DPDK) [5] and netmap etc. These libraries help in improving performance and help developers to focus on optimization e.g. how the packets should be batched. As discussed in [4], The Click modular router [6] 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 the flow of packets between different elements. Click [7] 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. VMs and Containers have greater influence on performance degradation. As mentioned in [4], comparing single process of NetBricks 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 of NetBricks 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. These results are further discussed in sectionIV-C

C. Isolation

Both packet and memory Isolation is the major challenge to achieve, as it directly affects the performance e.g. 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 packet should cross core boundaries. These isolation issues can be catered as mentioned in [4]: for memory isolation, instead of using VMs and Containers. Checks can be introduced both compile-time and run-time. For packet isolation, “Zero-Copy Software Isolation (ZCSI)” is proposed by the author in [4] that ensures the “safe 0-copy” packet I/O between NFs. This is implemented using unique types [8].

In order to address the issues of development and execution model, author proposed a framework called NetBricks [4]. This framework showcased a different way of developing and executing network functions that contradict with the traditional approach.

III. NetBricks Framework

Overview

NetBricks, a framework for developing and executing NFs on a single machine. It requires the re-writing of NF compatible with NetBricks development model. As mentioned in [4], it is not a limitation because of two reasons: a) not enough development progress has been done for NFs; b) NetBricks can also co-exists with traditional NFs, at the cost of performance. This framework provides both development and execution environment. For development model, it help developers to work on 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 i.e. VMs and Containers uses scheduling for isolation [4]. Another important aspect to consider in execution model is communication between NFs. 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 [4] named this functionality as ”ZCSI”. ”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 section below, development model is explained in detailed, later section describes the execution model to deploy the NFs.

A. Main Components of Development Model

As described in [4], NetBricks allows 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, packet structure consist of a) stack of header; b) the payload; c) reference to any per-packet metadata [4]. The header contains a structure that defines the length of a packet based on functional computation of its contents. The payload is actual data carried by the packet. Metadata defines internal communication within NF 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. The author provided the four packet operations as follows [4]:

- Parse: This operation takes the header type and structure as an input. Later analyzes the payload accordingly by using header type and update header stack. At the end header bytes are removed from the payload.
- Deparse: This operation is applied on header stack, it removes the bottom header from stack and return it to payload.
- Transform: This operation implements 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 drops all the packets at specific node when the user-defined function returns the false value.

2) Bytestream Processing: Main function of bytestream processing is to convert bytes arrays into packets. User-defined functions are applied on the bytes arrays. In [4], author provided the two bytestream operations as follows:

- Window: It takes four parameters as input i.e. window size, sliding increment, timeout, and stream user-defined function. Window is responsible for receiving and rearranging 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, connection is closed or timeout expires. Window operation also forward all received packets without modifying them or drop all packets and generate the modified byte array using packetize [4].
- Packetize: It allows the conversion of byte arrays into packets. Providing byte arrays and header stack, packetize converts the data into packets and assign relevant header [4].

3) Control Flow: Control flow deals with branching required in NF 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 developer the abstraction for re-routing packets as desired e.g. by user-defined functions, port, address, etc. While implementing the NF chains, it allows the developer to select which packet should be routed to next NF. The author provides three operations for control flow [4] as follows:

- Group By: It allows the branching with-in NF and branching across NF chains. It takes two input: number of groups for packet re-routing and user-defined function returning the packets with the ID of a group to which it belongs.
- 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.
- Merge: This operation provides a junction, where all the different branches can be merged e.g. 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 NF to partition state and avoid cross-core access or allow minimal access when required without using partition state. State management allows access across multiple cores. Within core accesses are synchronized but for cross-core accesses author proposed following options [4]: a) No-external-access i.e. one core for each partition; b) Bounded inconsistency [3] i.e. where one core has write access to partition and other cores only have read access; c) 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 also provides the execution environment to run the NF. This model ensures isolation and also deals with NF placement and scheduling [4].

1) Isolation:

- Memory Isolation: Traditionally, isolation is obtained by using VMs and Containers at the cost of performance loss for simple NF. Considering the complex NF, this performance loss dominates. In order to tackle this issue, NetBricks used a different approach to achieve isolation. It makes use of RUST [9] safe language that ensures type checks and LLVM [10] as a runtime. This combination of safe language and runtime achieves the memory isolation similar to that is obtained using the hardware Memory Management Unit (MMU). As mentioned in [4], safe language and runtime ensures following: a) disallow pointer arithmetic; b) bound checking on array accesses i.e. preventing random memory accesses; c) disallow accesses to null object i.e. preventing undefined behavior to ensure memory isolation; d) 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.
- Packet Isolation: The traditional mechanism is to send packets in a physical network, NFV follows the same footstep. NF send packets in the network by copying. This copying mechanism results in performance degradation of packet processing. NetBricks uses the unique types [8] instead of using the copying mechanism. As mentioned in [4], unique types are used to cater data races e.g. disallowing the simultaneous accesses to the same data. This approach is applied while implementing the NFs. When NF sends a packet, sender function losses the accesses to that packet and only relevant NF 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 [4]. NetBricks runs as a single process, packets are transferred between NFs using "function calls". In case of NF chain, a queue is maintained at receiving end.

2) Placement and Scheduling: NetBricks operates as a single process in which many NFs can be run. Consider the NF chains running as a directed graph having access to multiple available NIC interfaces. Before implementing, NetBricks have to decide at which core NF chains should run. Based on this, NetBricks make the scheduling decisions for packet processing. As the author mentioned in [4], currently NetBricks places all the network functions on a single core to get the maximum performance. In case of complex placements, NetBricks use ”shuffle” operation to process packets across multiple cores. In [4], 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” and “group by” operation. NetBricks uses “round-robin” scheduling for these operations.

IV. Framework Evaluation

This sections describes about the overheads of NFs developed using NetBricks compared to baseline NF build using other frameworks that was deployed on a testbed with specification mentioned in [4]. Furthermore it presents the performance and isolation analysis of NetBricks NF comapared to NF build with other frameworks.

A. Overheads and Results

This section describes the development model overheads imposed by the NetBricks and the results obtained. For comparison purposes, both NIC and DPDK were configured the same for NetBricks and baseline NF. In [4], the author mentioned few overheads that are as follows: a) for simple NF; b) for checking array bounds.

In the case of simple NF, the packet TTL NF was used. This NF decrements the time-to-live (TTL) parameter of packet upon receiving packets and drops the packet with TTL equal to zero. Both NetBricks version and normal version of NF were executed using a single core by send packets of 64 bytes and observed the throughput. The result of both NFs were almost the same as expected by the author. As mentioned in [4], after 10 experimental runs, the average throughput observed for baseline NF was 23.3 million packets per second (MPPS) whereas for NetBricks NF it was observed 23.2 MPPS. For latency at 80%, the round trip time (RTT) for baseline NF observed was 16.15 µs compared to NetBricks NF that was 16.16 µs.

In the case of array accesses [4], 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 [4]. The update of these array cells depends on a UDP source port number of packet under processing, making it difficult to ignore array bound checks. The author compared the NetBricks NF with baseline NF executed using single-core and used packets with random UDP source port [4]. Fig.1 shows the throughput achieved by two NFs when memory accesses per packet increased. NetBrick’s throughput was 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 [4]. This is because of large memory accesses causing cache misses resulting in reduced throughput. These cache misses dominating the overhead imposed by the NetBricks. Array bound checking is an overhead of NetBricks resulting in less throughput.

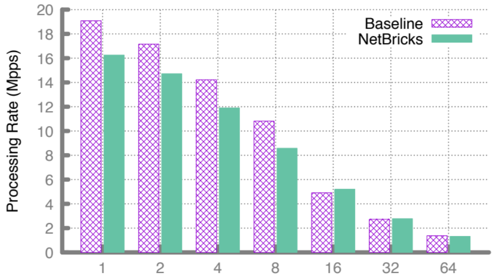


Fig. 1: Throughput comparison between NetBricks NF and Basline NF [4]

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

Fig. 2: Performance figures of NetBricks NF w.r.t other NFs [4]

B. Performance analysis of framework based different NFs

To further validate the development model of NetBricks, author implemented different categories of network functions using the NetBricks framework. Some of them are mentioned below [4]:

- Firewall: Based on Click [7], performs the linear scan of an Access Control List (ACL) to find relevant entry.
- NAT: Based on MazuNAT [], Network Address Translation (NAT) built using Click.
- IDS: Based on Snort [11], for signature matching.
- Monitor: Maintains per-flow counter similar to monitor module of Click.
- Load balancer: based on Maglev [12] , load balancer used by google.

Testing of the above-mentioned NFs were performed using both NetBricks framework and original NF. These NFs were executed on a single core [4]. Comparison results can be seen in fig.2, NetBricks NF performance is better as compared to other framework NF. As mentioned in [4], NetBricks NAT has a 3x better performance than MazuNAT.

# 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. 3: Throughput (MPPS) achieved comparing NetBricks Multi-core with Maglev [4]

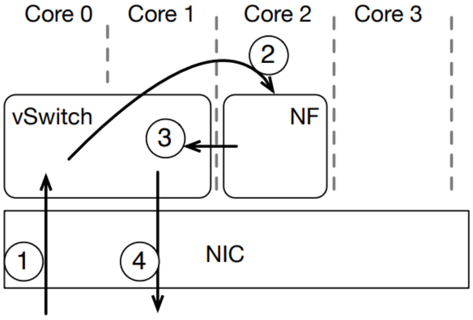


Fig. 4: VM/Container execution environment for simple NF [4]

Further experiments were performed on the NetBricks version of Maglev. This testing were on multi-core [4]. Fig.3 show the comparison of NetBricks version of Maglev with google Maglev. It was observed that NetBricks throughput (MPPS) is 2.9x to 3.5x better compared to results mentioned in [12]. Average latency observed was 19.9 µs for NetBricks and 32 µs for the original Maglev. These throughput and latency results are better but not to rely on because these two experiments (i.e. NetBricks Maglev version and Google Maglev) were performed on two different test-bed [4].

C. Isolation Analysis

This section discussed about the isolation achieved using the NetBricks isolation model.Isolation is ensured by NetBricks using safe language and runtime checks to avoid costs associated with core boundaries and crossing process [4]. The author first checked the cost involved with the single NF. 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 NFs, but when the computational cost of NF is higher then NetBricks execution environment becomes irrelevant [4].

NF vs NF Chains vs Complex NF

This section presents the analysis of single NF, NF chain and complex NF in term of isolation and for that author compared the NetBricks with VMs or Containers [4].

First case, single NF "that swaps the source and destination ethernet address for received packets and forward them out the same port" [4] was built using NetBricks. The other NF was built using C language were compared and executed in their execution environment i.e. VM, Container, and NetBricks. Fig.4 shows the test environment of VM and Container. A vSwitch was used that receives packets from NIC and forward them to network function running in either VM or Container. NF process those packets and forward them back to vSwitch, which was then forwarded to NIC. Both vSwitch and NF was running on DPDK and depend upon polling [4]. For better performance NF was assigned its dedicated CPU and two cores were assigned to vSwitch. As mentioned in [4], 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. Second due to packets copying. 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 [4]. As shown in fig.5, NetBricks receives packets directly from NIC, process them using NF code and send them back to NIC, all this process was running on single-core. Fig.6 shows the throughput achieved by different isolation techniques. As discussed in [4], Comparison of 0-copy SoftNIC Container with NetBricks with respect to 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 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

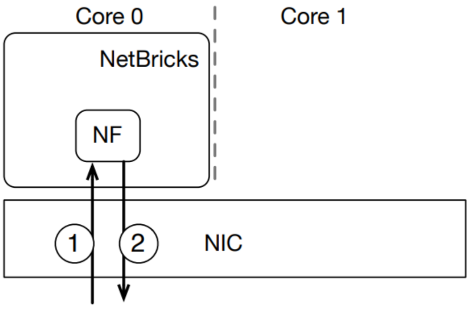


Fig. 5: NetBricks execution environment for simple NF [4]

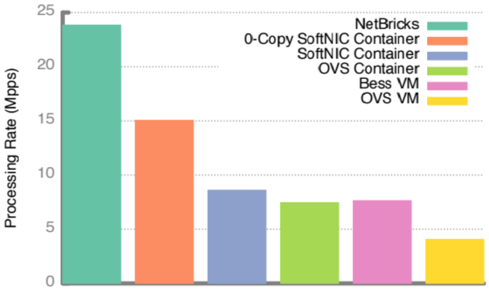


Fig. 6: Throughput achieved using different isolation techniques [4]

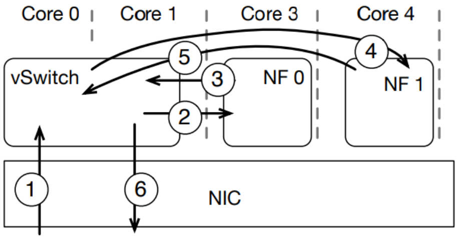


Fig. 7: VM/Container Execution environemt for NF chain [4]

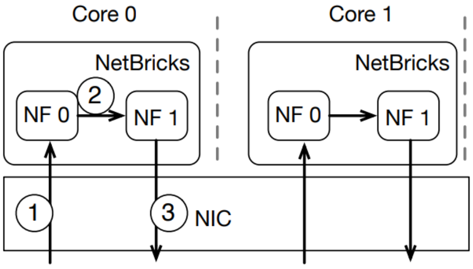


Fig. 8: NetBricks Execution environemt for NF chain [4]

as compared to DPDK's ring PMD driver used for Container [4].

Second Case, NF chain i.e. multiple instances of single NF, compute packet's TTL and drops the packet with TTL 0 [4]. Fig.7 shows the execution environment using VM or Container to run the network function chain e.g. NF0 and NF1. Author assigned two cores to vSwitch and two cores for NF chain, one for each NF. Fig.8 shows the execution environment of NetBricks to run the NF chain e.g. NF0 and NF1. NetBricks was tested for two cases [4]: a) with single-core; b) with two-cores. Fig.9 shows the throughput comparison of NetBricks with single-core, multi-core, VM and Container. As mentioned in [4], NetBricks multi-core (NB-MC) is 7x better than a container with SoftNIC and 11x better compared to VM with SoftNIC [4]. NetBricks with single-core (NB-SC) is 4x better than a container with SoftNIC and 6x better compared to VM with SoftNIC [4]. 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.9, 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 (i.e. 4 parallel I/O threads) [4]. Above throughput figures are for a packet size of 64 Bytes. Increasing the packet size degrades the performance by 15% [4]. Fig.10 shows the latency when using different isolation techniques.

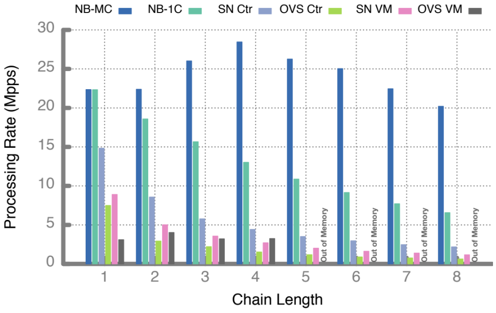


Fig. 9: Throughput of increasing NF chain, NetBricks single and multi core vs other technologies [4]

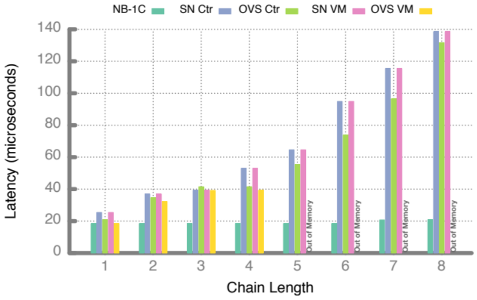


Fig. 10: Latency for increasing NF chain, NetBricks single/multi-core vs other technologies [4]

Third case, complex NF i.e. increases computation cycles for per-packet processing and used the same execution environment as the first case fig.4 and 5 as mentioned in [4]. Three core were assigned for VM and Container i.e. one for NF and two for vSwitch and in case of NetBriciks two cases were tested [4]: single-core and three-cores. The author modified the NF to use busy loops for the number of cycles after packet processing. Fig.10 shows the throughput comparison of per-packet processing by using different isolation techniques. As mentioned in [4], increasing complexity of network function results in increased computation time that overcomes the improvements offered by NetBricks e.g. 300 cycles per packet. As shown in fig.11, NetBricks isolation performs better as compared to other techniques e.g. VM or Container. NetBricks multi-core throughput is 4x better as compared to other technologies when complexity level is low e.g. 100 cycles per packet. As the complexity increase upto 600 cycles per packet, NetBricks mutli-core still performs as better as 2x comapred to VM and Containers.

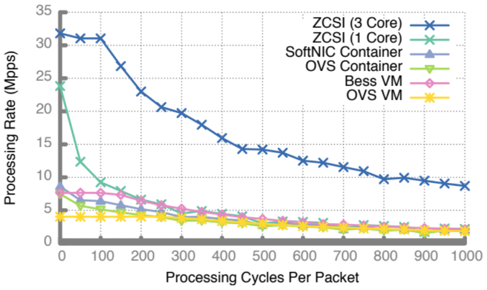


Fig. 11: Throughput of Complex NF using different isolation techniques [4]

V. Comparison of Frameworks

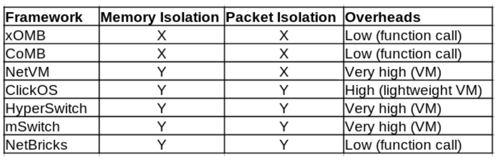
In this section, NetBricks is compared with other frameworks available in the market. This comparison is done based on the development model, execution model, and isolation.

A. Development Model

In terms of development model, there are many frameworks available such as YANFF [1] that helps in rapid NF 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 NFs. An API is used, that provides the high-level abstraction and manages low-level optimization. It separates the application-specific processing of NFs from the software stack that can be reusable or common for all NFs. Compared to NetBricks, libVNF supports clustered NFs deployment and horizontal scalability i.e. ensuring high availability. FLICK [14] framework is used for development and deployment of NF. 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 NF. FLICK language allows to focus on only application-specific logic ignoring low-level details. Developed NF 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 [15] is a virtualization-based platform that uses shared memory to exploit the DPDK library i.e. ensuring 0-copy between VMs. It uses the hypervisor-based switch to control the flow of packets and inter-VM communication. Also ensure isolation to some extent e.g. packet accesses to only trusted VMs. OpenNetVM [16] 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 NF i.e. container to NF Manager. It also allows the development of user-defined NF and makes use of NFlib to connect with NF Manager. HyperNF [17] is VM based framework, fully utilizing the available resources e.g. CPU cores. No dedicated cores assigned for packet I/O as per the merge model mentioned in [17] 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" is used for GPU resource allocation. 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. 12: Isolation between different frameworks [4]

C. Isolation

In terms of Isolation, SafeBricks [19] 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 i.e. ensures that software even kernel or hypervisor outside the enclave cannot tamper with enclave. It allows the NF to run inside the enclave and in the cloud, it seems like a black box. Fig.12 shows the author's comparison of the different framework (i.e. xOMB [20], CoMB [21], ClickOS [7], HyperSwitch [22], mSwitch [23]) in context with isolation.

VI. Conclusion

This seminar paper presented the detailed analysis of isolation. Current approaches e.g. VMs and Container ensures isolation but at the cost of performance degradation. This performance loss encouraged the entry of NetBricks. NetBricks contradicts with the current approaches. It runs the NF as a single process and for isolation it uses Type checking, array bound checking and unique types. The results mentioned in this paper shows that NetBricks performance is far better as compared to current approaches while ensuring the isolation.

VII. Further Research Topics

Optimation of 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. Currently, NetBricks works on the data plane and control plane functionality can be added in the future to get better performance. Lastly, the current approaches integrates with MANO systems for management and Orchestration of NFs. NetBricks should also support MANO systems to compete in the market.

References

[1] 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

[2] 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

[3] 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

[4] 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

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

[6] 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

[7] 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

[8] 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

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

[10] 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/

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

[12] 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

[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] 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

[15] 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/

[16] 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

[17] 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

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

[19] 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

[20] 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

[21] 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

[22] "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

[23] 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