A Userspace Transport Stack Doesn't Have to Mean Losing Linux Processing

Marcelo Abranches
University of Colorado, Boulder

Eric Keller
University of Colorado, Boulder

***Abstract*—While we cannot question the high performance capabilities of the kernel bypass approach in the network functions world, we recognize that the Linux kernel provides a rich ecosystem with an efficient resource management and an effective resource sharing ability that cannot be ignored. In this work we argue that by mixing kernel-bypass and in kernel processing can benefit applications and network function middleboxes. We leverage a high-performance user space TCP stack and recent additions to the Linux kernel to propose a hybrid approach (kernel-user space) to accelerate SDN/NFV deployments leveraging services of the reliable transport layer (i.e., stateful middleboxes, Layer 7 network functions and applications). Our results show that this approach enables high-performance, high CPU efficiency, and enhanced integration with the kernel ecosystem. We build our solution by extending mTCP which is the basis of some state-of-the-art L4-L7 NFV frameworks. By having more efficient CPU usage, NFV applications can have more CPU cycles available to run the network functions and applications logic. We show that for a CPU intense workload, mTCP/AF_XDP can have up to 64% more throughput than the previous implementation. We also show that by receiving cooperation from the kernel, mTCP/AF_XDP enables the creation of protection mechanisms for mTCP. We create a simulated DDoS attack and show that mTCP/AF_XDP can maintain up to 287% more throughput than the unprotected system during the attack.**

I. Introduction

Stateful middleboxes and Layer 7 (L7) network functions (NFs) are fundamental elements of modern networks and datacenters [13]. Stateful middleboxes are responsible for services like proxying, TCP splicing, stateful network address translation, firewalling, application load balancing, network intrusion detection systems (IDS), and content caching. These elements rely on services provided by the transport layer (e.g., TCP) to track Layer 4 (L4) state and inspect data content at the flow level (through data reassembly). With the increased pressure on the network from video conferencing, group collaboration [6], and digital entertainment [3], datacenters and cloud providers need to offer adequate infrastructure to support these trends. Given the dynamics and need for scalability, this trends towards software-based NFs, in the form of network functions virtualization (NFV).

Recent works [18], [14], [11], [12] have shown that it is currently hard for the operating system's kernel to provide the necessary performance to support these modern network services. This is mostly because today's operating systems add non-negligible overheads to packet I/O due to inefficiencies in data-structures and memory allocation (e.g., sk_buff, file descriptors, etc.), extra memory copies, unnecessary protocol processing, and an inability to react to microsecond scale bursts due to coarse temporal granularity in scheduling CPUs to the network applications threads.

To overcome these inefficiencies and enhance packet processing programmability, user space network processing toolkits, such as DPDK [4], have been introduced and are gaining in popularity. These toolkits give complete control of the networking hardware to the user space network processing application, enabling the development of high performance packet processing applications as many of the those kernel network stack inefficiencies can be avoided. Several projects have been built on the top of DPDK [18], [11], and they generally show that this approach can indeed improve the performance of NFs and network applications substantially. Following this trend, mTCP [14] proposed a highly scalable user space TCP stack that is optimized to multi-core systems and able to outperform the Linux TCP stack by up to 320%. This created opportunities to L4-L7 network functions frameworks [13], [17] to innovate in terms of functionality while also achieving high performance.

However, this gain in flexibility to build high performance network functions comes with costs. First, there is a need to dedicate CPU cores and network interfaces to the network application. Further, the application needs to busy poll the network interface queues in order to receive packets. Each of these causes high CPU consumption and leaves less CPU cycles for the network function logic. Second, as the kernel is completely bypassed, all of the configuration, monitoring, security, and network (protocol processing, bonds, etc.) features provided by modern kernels are also bypassed - leading to NFs needing to completely re-implement them in user space.

Observing the challenges introduced by network kernel bypass technologies, the kernel community introduced a new programmable packet processing framework, called the eXpress Data Path (XDP) [12], that enables efficient and safe custom packet processing inside the kernel. The main idea of XDP is to provide custom programs early access to the packet (before the packet reaches the Linux kernel network stack), giving the XDP program the ability to modify the packet and also to define a verdict to it (including rewriting packets and applying drop or redirect actions). As XDP is part of the kernel, the packet processing application runs inside the kernel context, having access to some of the capabilities provided by it. With XDP there is no need to dedicate resources (e.g., CPU and network interfaces) to the packet

processing application, and the XDP program can selectively make use of kernel services and therefore avoid the need to re-implement functionality.

On the top of XDP, the kernel community proposed a new high-performance network socket type called AF_XDP [16] that enables sending raw packets received at the XDP layer to user space. This opens up the possibility of building hybrid packet processing and NFV solutions where packets can be processed by user space applications, but with cooperation and support of XDP, NFs can leverage all of the integration and features that the kernel provides.

Leveraging the addition of XDP and AF_XDP to the Linux kernel, in this paper we argue that accelerating the transport layer using a hybrid (kernel-user space) approach is of benefit to SDN/NFV deployments as this scenario can leverage the performance of a high performance user space TCP stack without completely bypassing the kernel. This work is the first that we are aware of to introduce a high-performance TCP stack to AF_XDP. In particular, we make the following contributions:

- We propose an architecture and implementation which provides a hybrid packet processing model in which NFs can leverage both a high-performance user space TCP stack, and rich kernel functionality (Section IV).
- We demonstrate that this system is able to achieve a better CPU consumption profile that leaves more cycles to execute application and NFs code – e.g., for a CPU intense workload, mTCP/AF_XDP can have up to 64% more throughput due to the extra CPU cycles available. (Section V)
- We demonstrate the benefit of hybrid processing where we show that an network application is able to maintain up to 287% more throughput in the face of a DDoS attack, through using filtering at the kernel level. (Section VI)

The remainder of the paper is organized as follows: Section II describes related work and depicts the main challenges of their current architecture. Section III describes several scenarios that the hybrid kernel-userspace transport approach can benefit the network functions and applications world. Section IV describes the architecture and implementation of the solution. In Section V, we compare the hybrid kernel-userspace transport stack (mTCP/AF_XDP), with a full userspace transport stack that uses DPDK as the packet IO subsystem (mTCP/DPDK). We show in Section VI the ability to protect network applications from DDoS attacks through the hybrid-processing. Finally, in Section VII we conclude and discuss future work.

II. Related Work and Challenges

A. Kernel Bypass Approach for NFVs

The Linux operating system was designed to be as general as possible, and to support a wide range of applications and configurations. This means that the kernel network stack will perform costly processing and allocate heavy weight data-structures even if the packet only needs a few steps to processes lower-level protocols [12]. For example, for every packet that arrives, a data structure called *sk_buff* will be allocated to enable further protocol processing leveraging the rich semantic provided by this data structure [10]. After that, other costly processes (e.g., *__netif_receive_skb_core*) will be triggered to process the layered protocol stack, and to filter packets. As observed in [12], [10], this process slows down packet processing and could be avoided if it is possible to build custom packet processing applications that shortcut unnecessary processing.

To avoid these and other inefficiencies while providing maximum performance, modern NFs frameworks are built on top of kernel bypass technologies [14], [13], [15], [17]. This approach has the benefit of allowing highly customizable packet processing applications to avoid, for example, unnecessary protocol processing and provide more efficient processing pipelines.

mTCP is a high-performance user space TCP stack that is built on kernel bypass technologies, to improve performance, and in turn, can leverage multicore systems to improve scalability. To enable multi-core scalability mTCP is built with a series of optimization techniques (e.g., lock-free, per-core cache-friendly data structures). The mTCP process operates by running as distinct threads (one for the application and one for the mTCP logic) on each CPU. mTCP leverages RSS to distribute incoming packets from different flows among different CPU cores, while handling core affinity. Being a user level implementation, mTCP decouples the TCP logic and development from the kernel complexity, which smooths the development of new features to the stack itself and enables building new solutions on the top of it. For example, mOS [13] is a framework built on top of mTCP that allows building stateful middleboxes with full support for L4-L7 processing. mOS currently supports DPDK [4] and Netmap [19] as packet I/O subsystems.

B. Challenges of the Current Approach

As we saw in the previous subsection, several network function frameworks have benefited from the performance enabled by the kernel bypass approach. However, it is important to recognize possible limitations of a complete kernel bypass approach and look for new opportunities to evolve the current solutions. In the next paragraphs we will list the limitations that motivated this work.

Inefficient CPU usage: Being a kernel bypass framework, DPDK does not rely on on the kernel networking mechanisms to receive packets (e.g., interrupts, ksoftirqs and NAPI) [12], [16]. Instead, DPDK needs to busy poll the NIC queues in order to receive packets. Although this mechanism can provide better latency profiles to applications, this causes high CPU consumption, leaving less cycles to process the NFs logic as we will demonstrate in Section V.

Lack of system integration: The Linux kernel has a rich ecosystem to provide network connectivity, monitoring,

configuration, resource sharing, isolation, and security. Generally, kernel bypass technologies are blind to this ecosystem [10], which may slow down progress in this context as much of this functionality needs to be re-implemented in user space. As we will see in the next sections, a better alternative is to selectively use kernel functionalities, while still leveraging the high performance and flexibility achieved by user space technologies.

III. Motivation

In this work, we ask if we can leverage the recent additions to the Linux kernel to address the challenges listed in Section II and benefit a high performance user space TCP stack [14], which in turn extends to the NFs frameworks built on the top of it that provide for processing capabilities all the way up to the application layer [13], [17].

XDP enables flexible and efficient programmable packet processing inside the kernel [12]. The key enabler for XDP is the eBPF virtual machine, that allows only verifiable eBPF code to be loaded inside the kernel. XDP enables attaching eBPF programs to process packets at the earliest point inside the kernel (i.e., at NIC driver level, before the packet reaches the kernel network stack). If the NIC supports it, the eBPF programs can be offloaded to the NIC hardware. XDP programs allow, for example, rewriting packet headers and accessing packet metadata (e.g., queue number on multiqueue NICs and custom metadata). The XDP hook execution finishes by assigning a verdict to a packet. Possible verdicts are to drop the packet, transmit the packet back on the same interface as it arrived, pass the packet to be processed by the kernel stack, and redirect the packet to another interface (physical or virtual), another CPU for further processing, or even to a special socket that sends the packet to user space (i.e., AF_XDP).

AF_XDP is another addition to recent Linux kernels. It enables sending raw packets to user space at high-rates through zero-copy transfers (as long as the NIC driver supports this [16]). To send and receive packets, AF_XDP interacts with the kernel via specialized rings (i.e., fill, completion, Tx and Rx rings), and uses a special memory area called UMEM. Those rings are used by the userspace network application and kernel to switch control of UMEM areas (which stores packet data) between each other (i.e., fill and completion rings). They are also used by the application to receive packets, and inform the kernel the packets that should be sent (i.e., Rx and Tx queues).

We use XDP and AF_XDP to provide a new packet I/O subsystem for mTCP. This new subsystem provides an efficient CPU consumption profile for mTCP applications and NFs, and also provides better system integration.

Providing efficient CPU consumption: Middleboxes need to perform packet IO, but they also need available CPU cycles to process the network function's logic, and as we will see in Section V, this is a challenge for DPDK. On the other hand, XDP enables a better CPU consumption profile as it does not need to rely on busy polling to perform packet I/O, because it has the Linux interrupt infrastructure and syscalls available.

Providing system integration: While kernel bypass packet I/O systems like Netmap [19] may bring a better CPU consumption profile, it lacks good Linux system integration. For example, currently Netmap is not part of the Linux kernel, so it may be a burden to maintain Netmap based applications [12]. Moreover, it does not support XDP which limits its data-path programmability.

XDP does not take over the ownership of the NIC as DPDK, so it is possible to share the interface among multiple applications (providing the necessary XDP/eBPF logic). It is also possible to use the Linux network configuration and monitoring tools like ethtool, iproute2 which may easy the integration of XDP based network solutions with automation tools like Puppet, Ansible and Chef. Container technology plays an important role in NFV deployments [15], and as it relies heavily on kernel functionalities to provide resource isolation and configuration, XDP based deployments can easy the integration of fast packet processing and enhanced networking capabilities to the containers world. As DPDK completely bypasses the kernel, enabling these functionalities to DPDK based applications is challenging [8].

Leveraging the support of the rich kernel ecosystem: As AF_XDP sockets can send raw packets to userspace after they are processed on the XDP hook, they enable a hybrid networking stack approach. XDP can be used as a first layer that provides enhanced network functionalities to the high performance transport layer running in user space. This first layer can be used to protect the upper user space stack [2] and also to provide kernel integration functionality leveraging BPF maps and kernel helper functions [12], [10]. Kernel helper functions can be used, for example, to support packet checksum calculation and also to access kernel routing tables [5], [10]. BPF maps can be used by the XDP redirect logic to react to events occurring at different kernel subsystems and different resource monitoring points, including in user space (e.g., CPU load and cgroups) [12], [7] opening up opportunities to create new load-balancing mechanisms. Furthermore, XDP can also provide a flexible mechanism to implement access control lists (ACLs), packet filters, and other functionalities to protect the user level transport layer.

This approach brings flexibility to NF and applications that leverage high performance user space transport stacks, as the XDP logic allows selecting only the needed kernel network functionalities to be used. It does this via kernel helpers and does not require the packet to traverse the whole Linux network stack.

IV. Architecture and Implementation

A. mTCP/AF_XDP Integration

Now that it is clear the motivation behind having a hybrid kernel-userpace TCP stack we present the architecture of the mTCP/AF_XDP stack in Figure 1. This figure shows the basic interactions between the different components of the

solution, as we will explain in the next paragraph. Notice that to obtain maximum performance, we decided to have one UMEM per AF_XDP socket, and also one AF_XDP socket per mTCP thread, so we could completely avoid synchronization overheads and obtain maximum performance.

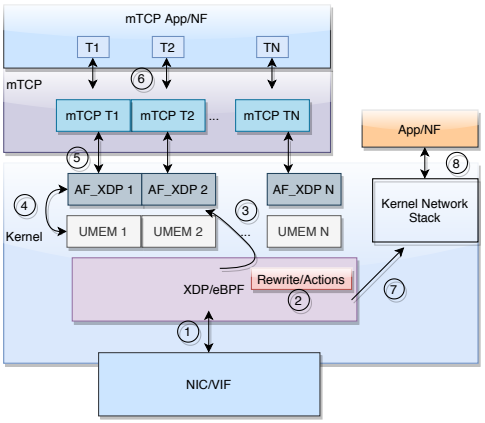

Fig. 1: mTCP/AF_XDP architecture

The life of a packet inside mTCP/AF_XDP (see Figure 1): The XDP program ②, decides to which AF_XDP socket a packet should be sent. In our implementation, we use hardware packet steering, and the NIC queue in which the packet arrived ① is used as the index in the BPF map to select the target AF_XDP socket for the packet ③. We avoid extra cache overheads by pinning a mTCP thread on the same core that handles the *ksoftirq* on behalf of a packet received by the multi-queue NIC. The kernel places the packet on the UMEM area using one of the addresses available on the fill ring associated to that socket (if the NIC driver supports zero-copy, the NIC will place the packet at UMEM via DMA). After that, the kernel places a file descriptor on the socket RX ring. The mTCP/AF_XDP packet I/O subsystem uses the *(*poll) system call to monitor this ring ⑤, and our implementation enables sending and receiving packets in batches for best performance. The batch of packets is received by the mTCP's stack main loop which performs the TCP stack logic, and makes data available to the application threads through the mTCP events system and userspace function calls (e.g., mtcp_read) ⑥. After successfully receiving the packets mTCP/AF_XDP returns ownership of these UMEM areas to the kernel by posting their descriptors in the fill ring.

The sending path is similar. The mTCP/AF_XDP packet I/O subsystem uses the TX ring to place file descriptors pointing to the packet buffers it wants to send ⑤. The kernel then assumes control of this UMEM region and sends the packet to the NIC which sends the packet out. After successful transmission the kernel makes this UMEM area available for sending new packets by posting a memory descriptor on the completion ring ④. mTCP/AF_XDP consumes these descriptors and uses them on the next iteration of the packet sending routine. Finally, the XDP program may also be customized to provide extra functionality (e.g., stack protection features), or even to redirect the packet to an NF or application using the Linux kernel network stack, allowing coexistence ⑦, ⑧.

To implement this solution we added about 500 lines of C code to the mTCP code base. We have specific eBPF/XDP code (afxdp_kern.c), which is responsible for sending the desired packets to the AF_XDP sockets, making them available at the mTCP layer. We leverage the modular mTCP packet I/O design, to add a new packet I/O module (afxdp_module.c) and make targeted modifications to other mTCP components to support it. The code is available in our mTCP fork [1], and we expect to merge it to the main mTCP repository soon.

B. NFV Deployments

We envision that L4-L7 network functions built on top of mTCP based frameworks (e.g., [13], [17]) can benefit from our proposal by leveraging cooperation scenarios with XDP/eBPF (see section VI as an example), the high performance provided by mTCP [13], [17] and the better CPU consumption profile enabled by our solution - see section V - (which will ultimately produce more resource and power efficient solutions). In this scenario, NFs like L7 caches, protocol accelerators, and IDS can leverage services provided by XDP/eBPF to control how packets flow, determining, for example, which packets should be sent to a specific NF, which of them should be sent to the Linux stack (e.g., to handle corner cases), which packets should be dropped and which of them should be routed to the next hop or forwarded to an application in the case the flow does not need NF processing. Interactions between the userspace NFs and XDP/eBPF should occur via eBPF maps, and enhanced integration with the kernel should be achieved through the kernel helpers available to the XDP layer.

V. Evaluation

To demonstrate the value and feasibility of the proposed approach, in terms of performance and added functionality, in our evaluation we answer the following questions regarding using an AF_XDP based packet IO subsystem on a high performance user space TCP stack:

- Can the proposed system provide high performance?
- Can we have a better CPU consumption profile that enables more CPU cycles to be consumed running application code?

Experimental Setup: To answer these questions, we setup two testing environments on Cloudlab Wisconsin [9], one for mTCP/DPDK and another for mTCP/AF_XDP. Each environment is composed by 1 physical server machine that runs mTCP code (type c220g5 [9], Ubuntu 18.04.1 LTS Kernel 5.3.0-61-generic) and 5 physical client machines (type c220g1 [9], Ubuntu 18.04.1 LTS Kernel 5.3.0-61-generic). The server machines run the HTTP server that ships with mTCP (*epserver*). The client machines run *ab* (Apache Benchmark). As each client host has 16 cores available, we run 16 instances of *ab* on each host. We have observed that in

our setup, each client *ab* instance has maximum performance when sending 50 parallel HTTP connections, so we use this configuration on all tests. The server machines have 2 sockets with 10 cores each and each socket is attached to one NUMA node. These machines have only one dual-port 10 GbE NIC attached to NUMA node 0, so we only report results for threads running on the processor on the first socket. We used the I40e Intel NIC driver, and all of the AF_XDP experiments use zero-copy mode. Also, we observed that DPDK performs poorly when the number of cores dedicated to mTCP is not a power of two (we do not observe this limitation on the AF_XDP implementation). So, to have a fair evaluation we run our experiments using up to 8 cores on the server machines. In this work, we did not implement hardware TCP checksum offload for mTCP/AF_XDP, so we also report results for DPDK with it disabled (refered to as DPDK on the labels). For maximum performance, we disable hyper-threading and CPU power saving for each core. To isolate these cores, and avoid the kernel scheduling other user level threads on them, we use the *isolcpus* statement at boot time.

The Spectre and Meltdown mitigations affected the performance of eBPF programs, so AF_XDP is also impacted [16]. Users in controlled environments, where only trusted code can be executed, may opt to disable these mitigations. As we saw maximum performance for mTCP/AF_XDP when we disabled the mitigations, we include this scenario in the results on Figure 2. In our experiments, DPDK performance does not seem to be affected by those mitigations, so we only include these results for AF_XDP. As we cannot expect that all environments to be controlled and only run trusted code, for all other experiments we leave the mitigations enabled. In [16] the authors proposed a socket option called *XSK_ATTACH*, that automatically loads a minimal XDP code that only redirects packets that arrive on a *queue_id* to an AF_XDP socket avoiding the user to have to provide a custom XDP code. This code is optimized and minimizes the impacts of the mitigations. In our tests, we do not use this socket option, as we want to maintain the flexibility of having custom XDP code in our hybrid stack.

For each test we setup each client instance to send 1 million requests (50 in parallel for each instance). Unless specified differently, in each test we use all five client hosts with 16 *ab* instances and each client instance sends HTTP requests to download a 64B file from the server. Each test is repeated 5 times, and we report our results using the average of each metric and standard error (although the bars are too small to be noticed on the graphs). In [14], the authors show that mTCP can outperform the Linux TCP stack by several orders of magnitude (up to 25 times for small messages), so we do not include Linux TCP in our evaluation. Finally, we use Linux *Perf* tool to analyze the overheads of each implementation and other metrics that may affect their performance (e.g., number of CPU cache misses, context switches and so on).

Raw performance evaluation: To demonstrate that

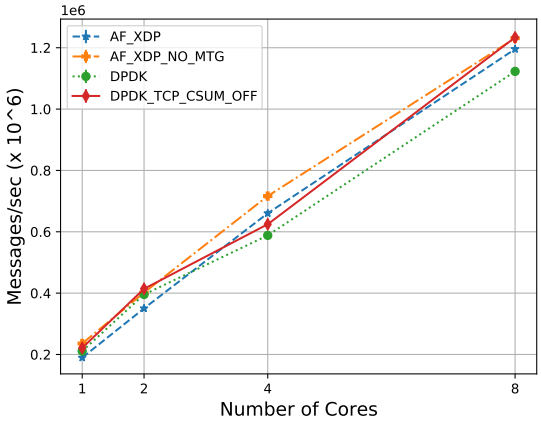

Fig. 2: Different number of Cores

mTCP/ AF_XDP can support high performance and scale, in the first experiment we compare mTCP's core scalability for mTCP/ DPDK and mTCP/AF_XDP. We can see in Figure 2 that mTCP's throughput scales almost linearly with the number of cores for all implementations. mTCP/DPDK with HW TCP checksum offload enabled (DPDK_TCP_CSUM_OFF) has the best performance for 2 and 8 cores, with AF_XDP with mitigations disabled (AF_XDP_NO_MTG) having performance almost as good as it for 1, 2 and 8 cores. We observe that for 4 cores, mTCP/DPDK has some drop in performance. Because of that, mTCP/AF_XDP outperforms mTCP/DPDK for 4 cores. It is important to notice that as we do not implement hardware TCP checksum offload to AF_XDP, the application has to spend CPU cycles to calculate it. In fact, we observe that when we disable the hardware TCP checksum offload on mTCP, it can spend up to 8% of its processing time performing those calculations. Observing these results, we expect mTCP/AF_XDP to improve its performance as we integrate hardware TCP checksum offload for mTCP/AF_XDP, which we leave as future work.

Efficient CPU consumption profile: In this experiment we evaluate if mTCP/AF_XDP can provide an efficient CPU consumption profile (figure 3) and the effects of having more CPU cycles available to execute application code (figure 4). We can see in Figure 3 that mTCP/AF_XDP gradually increases CPU consumption as the number of client hosts increases. In contrast mTCP/DPDK relies on busy polling to receive packets, so it always consumes 100% of CPU. mTCP/ AF_XDP does not rely on busy polling to perform packet IO, which saves precious CPU cycles that can be spent to run the mTCP stack and application code. We analyze where each implementation spends more time, and we observe that mTCP/DPDK spends non-negligible time on the receiving path busy polling loop. In contrast, mTCP/AF_XDP spends more time executing application code and also handling important events on the mTCP stack. We can also observe in this figure that mTCP/AF_XDP does not hit 100% CPU

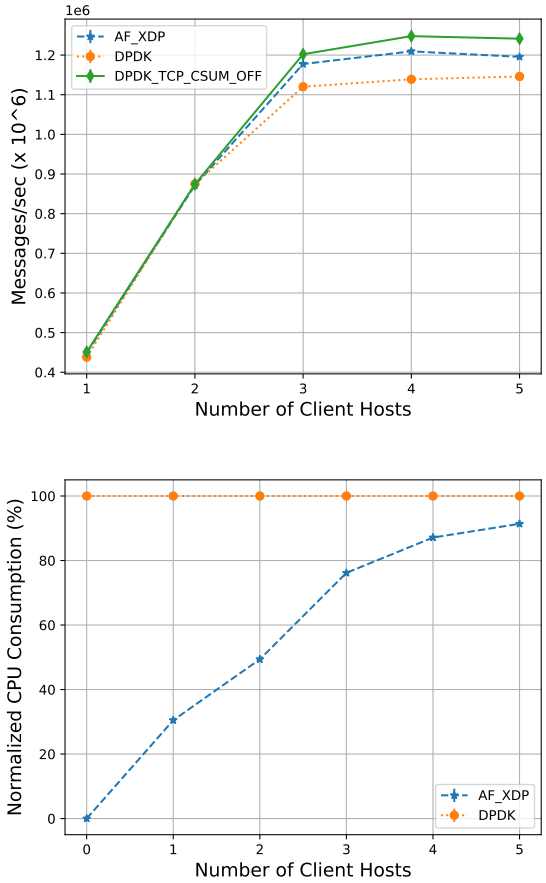

Fig. 3: CPU Consumption vs Load

consumption for 5 client hosts, even though the server is saturated. This is because our CPU measurements start when there is no load on the server, and goes until all clients finish sending the HTTP requests, so at the end of the experiment there is also a drop in load and this reflects on the average CPU consumption in this test.

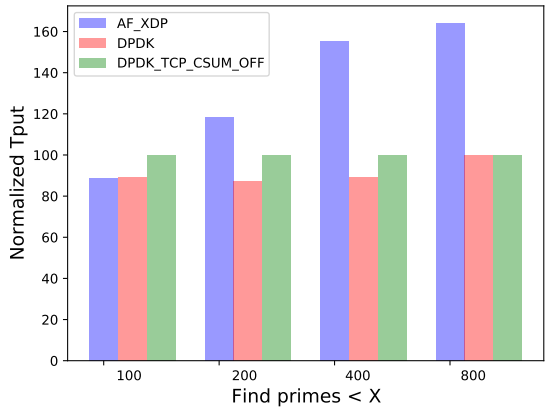

Fig. 4: CPU Intensive Workload

To observe the mTCP/AF_XDP benefits of having more available cycles to process application logic, we add a simulated CPU intensive HTTP application by executing a function to find all the prime numbers smaller than a given X in each HTTP request. The results can be seen in Figure 4, which shows the normalized throughput using mTCP/DPDK with HW TCP checksum enabled as the baseline. We start the the server with 8 cores. To find primes lower than 100, the application does not get CPU bound enough for the mTCP/ AF_XDP benefits to be perceived. But, as we increase X above 200, we observe that more CPU power is needed to find the prime numbers, and in this scenario mTCP/ AF_XDP can have up to 64% more throughput than mTCP/ DPDK (for X = 800).

Having more CPU cycles available may benefit mTCP to run CPU intensive applications (e.g., node.js) and network functions logic, and also enables mTCP to better support SSL/TLS.

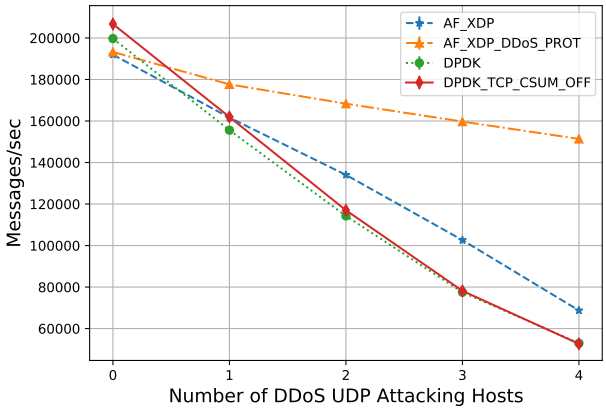

Fig. 5: XDP Protection to DDoS Attack

VI. Protecting the Userspace TCP Stack

As we showed on section III, one of the benefits of mTCP/ AF_XDP is the possibility to leverage XDP/eBPF to enhance and protect the mTCP stack. To show this, we implement a simulated DDoS attack and an XDP DDoS protection similar to the ones described in [12] and [2]. In this experiment we use 4 of the 5 client hosts to generate UDP packets targeting the mTCP server. Each attacking host uses 16 *nping* instances (one for each core) to generate UDP packets at a rate of 10 thousand packets/second. We gradually increase the intensity of the attack by joining new client hosts to the attack. The other client host runs *ab* to generate HTTP requests to the server. We start the mTCP servers on a single core to make the attack more pronounced.

As mTCP is a user space TCP stack, all of the security and isolation mechanisms provided by the Linux kernel are bypassed, so mTCP is responsible for dropping the UDP packets. This is not the case with mTCP/AF_XDP. To protect the mTCP stack from this attack, we change the XDP/eBPF

code that sends the packets to the AF_XDP sockets (see Section IV) to parse each received packet and if it is a UDP packet, apply the XDP_DROP verdict, and otherwise send the packet to the AF_XDP socket so it can be normally processed by the mTCP stack.

Figure 5 shows the impacts of the attack. In this experiment we measure the total HTTP requests completed for each attack intensity. For mTCP/DPDK with hardware TCP checksum offload enabled, mTCP's throughput can drop 3.9 times when the attack is on its maximum intensity. At the same time, mTCP with XDP DDoS protection is able to maintain 2.87 times more throughput than mTCP/DPDK versions when the attack is at its maximum load. It is interesting to notice that mTCP/AF_XDP with no DDoS protection (AF_XDP label) can handle the attack better than mTCP/DPDK versions. This is because mTCP/AF_XDP has more CPU cycles available to drop the malicious packets.

By applying this XDP protection mechanism we free mTCP threads from the burden to process the malicious UDP packets, so the impact of the attack is minimized.

VII. Conclusion and Future Work

In this work we have enabled the power of eBPF and Linux system integration to cooperate with a high-performance user space transport layer. We have shown that this approach can have performance compatible with a high-performance kernel bypass approach, but providing enhanced capabilities that come from the OS kernel. This opens up the opportunity to innovate L2-L7 network functions in terms of functionality, deployment, performance and security.

As future work, the first step is to enable TCP hardware checksum offloading for mTCP/AF_XDP. We expect that this will greatly improve mTCP/AF_XDP's performance, as we described in Section V. Another opportunity is to work on NF stacks built on top of mTCP ([13], [17]) to investigate cooperation scenarios between XDP/eBPF and network functions such as L7 caches, protocol accelerators, and IDS (e.g., advanced forwarding mechanisms, eBPF hardware offloads, etc.). In this context we also want to investigate new NF deployment scenarios, for example, how can the NF containers world leverage the enhanced networking capabilities provided by XDP while providing state of the art high performance L4-L7 services.

Acknowledgments

This work was supported in part by NSF Grants 1652698 (CAREER) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] Author's mTCP fork - AF_XDP support to mTCP. https://github.com/mcabranches/mtcp.

[2] Cloudflare, How to drop 10 million packets per second. https://blog.cloudflare.com/how-to-drop-10-million-packets/.

[3] Deloitte, Media and entertainment industry outlook trends. https://www2.deloitte.com/us/en/pages/technology-media-and-telecommunications/articles/media-and-entertainment-industry-outlook-trends.html.

[4] DPDK, Data Plane Development Kit. https://www.dpdk.org/.

[5] Linux, bpf-helpers(7) — Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[6] Micrsoft, Remote work trend report. https://www.microsoft.com/en-us/microsoft-365/blog/2020/04/09/remote-work-trend-report-meetings/.

[7] Prototype Kernel, eBPF - extended Berkeley Packet Filter. https://prototype-kernel.readthedocs.io/en/latest/bpf/.

[8] Red Hat, Mobile Networks - Performance and Optimization. https://access.redhat.com/documentation/en-us/reference_architectures/2017/html/deploying_mobile_networks_using_network_functions_virtualization/performance_and_optimization.

[9] The Cloudlab Manual, Hardware. https://docs.cloudlab.us/hardware.html.

[10] D. Ahern. Leveraging kernel tables with xdp. In *Linux Plumbers Conference*, 2018.

[11] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16. IEEE, 2015.

[12] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 54–66, 2018.

[13] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 113–129, 2017.

[14] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 489–502, 2014.

[15] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 97–112, 2017.

[16] M. Karlsson and B. Töpel. The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, 2018.

[17] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood. Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 504–517. ACM, 2018.

[18] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 361–378, 2019.

[19] L. Rizzo. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, 2012. USENIX Association.