Bringing the Power of eBPF to Open vSwitch

William Tu[1]   Joe Stringer[2]   Yifeng Sun[1]   Yi-Hung Wei[1]

u9012063@gmail.com   joe@cilium.io   pkusunyifeng@gmail.com   yihung.wei@gmail.com

[1]VMware Inc.   [2]Cilium.io

Abstract

eBPF is an emerging technology in Linux kernel with the goal of making Linux kernel extensible by providing an eBPF virtual machine with safety guarantee. Open vSwitch (OVS), is a software switch running majorly in Linux operating systems, its fast path packet process is implemented in Linux kernel module, openvswitch.ko. To provide greater flexibility and extensibility to OVS datapath, in this work, we present our design on making use of eBPF technology in OVS datapath development with two projects: the OVS-eBPF project and the OVS-AFXDP project. The goal of OVS-eBPF project is to re-write existing flow processing features in openvswitch kernel datapath into eBPF program, and attaching it to Linux TC. On the other hand, the OVS-AFXDP project aims to by-pass the kernel using an AF_XDP socket and moves most of the flow processing features into userspace. We demonstrate the feasibility of implementing OVS datapath with the aforementioned technologies and present the performance numbers in this paper.

1. Introduction

eBPF, extended Berkeley Packet Filter, enables userspace applications to customize and extend the Linux kernel's functionality. It provides flexible platform abstractions for network functions, and is being ported to a variety of platforms. In the Linux kernel, users can attach eBPF programs to TC and XDP hook points as shown in Fig 1. Based on this design, we explore the possibilities of utilizing eBPF to implement OVS datapath in threefold: 1) in-kernel flow processing by attaching eBPF programs to TC, 2) offloading a subset of flow processing to XDP (eXpress Data Path), and 3) moving the flow processing to userspace by using AF_XDP.

Firstly, in OVS-eBPF project, we attach flow processing eBPF programs to TC. We start with the most aggressive goal that we plan to re-implement the entire features of OVS kernel datapath under net/openvswitch/* into eBPF code. We work around a couple of eBPF limitations, for example, the lack of TLV (Type-Length-Value) support leads us to redefine a binary kernel-user API using a fixed-length array; without a dedicated way to execute a packet, we create a dedicated device that attached a eBPF program to handle


```
+----------------------+
|    Network Stack     |
+---------^------------+
          |
+---------|------------+
| Traffic Control (TC) | --> eBPF TC hook
+---------^------------+
          |
+---------|------------+
|     Driver (XDP)     | --> eBPF XDP hook
+---------^------------+
          |
+---------|------------+
|    Network card      |
+---------^------------+
          |
          | packets
```
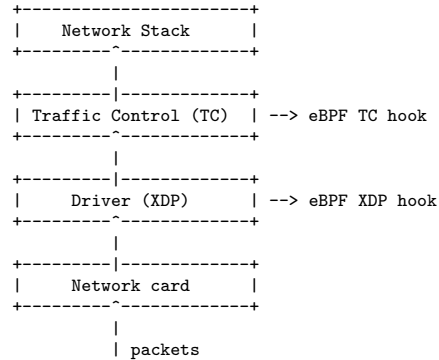
Figure 1: eBPF program hook points in the Linux kernel.

packet execute logic for userspace to kernel packet transmission. Currently, OVS-eBPF can satisfy most of the basic features for flow processing and tunneling protocol support, and we are investigating more complicated features such as connection tracking, NAT, (de)fragmentation, and ALG.

We can attach eBPF programs one layer below TC to XDP (eXpress Data Path). It is a much faster layer for packet processing, but there is almost no extra packet metadata available, and XDP only provides limited kernel helpers. Depending on the type of flows, OVS can offload a subset of its flow processing to XDP. However, the fact that XDP has fewer helper function support implies that either 1) only very limited number of flows are eligible for offload, or 2) more flow processing logic needed to be implemented in native eBPF code. For example, it is more difficult for OVS datapath to provide tunnel support in XDP, since lightweight tunnel kernel helpers are not available.

XDP provides another way for interacting with userspace programs, called AF_XDP. AF_XDP is a socket interface for control plane and a shared memory API for accessing packets from userspace application. Using AF_XDP, the OVS-AFXDP project redirects packets to userspace, and processes the packets using OVS's full-fledged userspace datapath implementation, dpif-netdev. In this approach, OVS-AFXDP project treats the AF_XDP as a fast packet-I/O channel.

This paper focuses on the OVS-eBPF and OVS-AFXDP projects. The remainder of this paper is organized as follows:

In Section 2, we first provide some background information on eBPF, XDP, and AF_XDP. We then present the design, implementation, and evaluation of the OVS-eBPF and OVS-AFXDP projects in Section 3 and Section 4 respectively. Finally, Section 5 concludes the paper and discusses the future work.

2. Background

2.1   OVS Forwarding Model

OVS is widely used in virtualized data center environments as a software switching layer inside various operating systems, including FreeBSD, Windows Hyper-V, Solaris and Linux. As shown in Figure 2, the architecture of OVS consists of two major components: a slow path and a fast path. OVS begins processing packets in its datapath, the fast path, shortly after the packet is received by the NIC in the host OS. The OVS datapath first performs packet parsing to extract relevant protocol headers from the packet and stores it locally in a manner that is efficient for performing lookups (flow key), then it uses this information to look into the match/action cache (flow table) and determines what needs to be done for this packet. If there is no match in the flow table, the datapath passes the packet from the kernel up to the slow path, `ovs-vswitchd`, which maintains the full determination of what needs to be executed to modify and forward the packet correctly. This process is called packet *upcall* and usually happens at the first packet of a flow seen by the OVS datapath. If the packet matches in this flow table, then the OVS datapath executes its corresponding actions from the flow table lookup result and updates its flow statistics.

In this model, the `ovs-vswitchd` determines how the packet should be handled, and passes the information to the datapath inside the kernel using a Linux generic netlink interface. Over the years the OVS datapath features evolved. The initial OVS datapath used a microflow cache for its flow table, essentially caching exact-match entries for each transport layer connection's forwarding decision. And in later versions, two layers of caching were used: a microflow cache and a megaflow cache, which caches forwarding decisions for traffic aggregates beyond individual connections. In recent versions of OVS, datapath implementations include features such as connection tracking, stateful network address translation, and support for layer 3 tunneling protocols.

2.2   eBPF Basics

Berkeley Packet Filter, BPF, is an instruction set architecture proposed by Steven McCanne and Van Jacobson in 1993 [14]. BPF was designed as a generic packet filtering solution and is widely used by every network operator today, through the well-known tcpdump/wireshark applications. A BPF interpreter is attached early in the packet receive call chain, and it executes a BPF program as a list of instructions. A BPF program typically parses a packet and decides
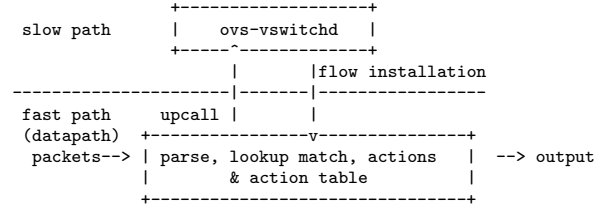
```
                +--------------------+
slow path       |    ovs-vswitchd    |
                +-----^--------------+
                      |         |flow installation
--------------------|-------|-----------------
fast path   upcall |       |
(datapath)  +---------------v---------------+
 packets--> | parse, lookup match, actions  | --> output
            |          & action table       |
            +-------------------------------+
```

Figure 2: The forwarding plane of OVS consists of two components; ovs-vswitchd handles the complexity of the OpenFlow protocol, and the datapath acts as a caching layer to optimize the performance. A flow missed by the match/action table in the datapath triggers an *upcall*, which forwards the information to ovs-vswitchd. ovs-vswitchd installs an appropriate flow entry into the datapath's match/action table.

whether to pass the packet to a userspace socket. With its simple architecture and early filtering decision logic, it can execute this logic efficiently.

For the past few years, the Linux kernel community has improved the traditional BPF (now renamed to classic BPF, cBPF) interpreter inside the kernel with additional instructions, known as extended BPF (eBPF). eBPF was introduced with the purpose of broadening the programmability of the Linux kernel. Within the kernel, eBPF instructions run in a virtual machine environment. The virtual machine provides a few registers, stack space, program counter, and a way to interact with the rest of the kernel through a mechanism called helper functions. Similar to cBPF, eBPF operates in an event-driven model on a particular hook point; each hook point has its own execution *context* and execution at the hook point only starts when a particular type of event fires. A BPF program is written against a specific context. For example, a BPF program attached to a raw socket interface has a context which includes the packet, and the program is only triggered to run when there is an incoming packet to the raw socket.

The eBPF virtual machine provides a completely isolated environment for its bytecode running inside; in other words, it cannot arbitrarily call other kernel functions or access into memory outside its own environment. To interact with the outside world, the eBPF architecture white-lists a set of helper functions that a BPF program can call, depending on the *context* of the BPF program. For example, a BPF program attached to raw socket in a packet context could invoke VLAN push or pop related helper functions, while a BPF program with a kernel tracing context could not.

To store and share state, eBPF provides a mechanism to interact with a variety of key/value stores, called *maps*. eBPF maps reside in the kernel, and can be shared and accessed from eBPF programs and userspace applications. eBPF programs can access maps through helper functions, while userspace applications can access maps through BPF system calls. There are a variety of map types for different use cases, such as hash tables or arrays. These are created by a userspace program and may be shared between multiple eBPF programs running in any hook point.

Userspace

```
┌──────────┐     ┌──────────┐     ┌──────────┐
│  Source  │ ──► │  LLVM +  │ ──► │ iproute  │
│   Code   │     │  Clang   │     │          │
└──────────┘     └──────────┘     └──────────┘
```

Kernel

```
┌──────────┐       ◇─────────◇       ┌──────┐
│ Network  │       │ Verifier │      │ maps │
│  Stack   │       ◇─────────◇       └──────┘
└──────────┘            │ pass
┌──────────┐      ┌──────────┐   ┌──────────┐
│    TC    │◄─────│   BPF    │   │   BPF    │
│  ingress │ skb  │  Prog1   │   │  Prog2   │
└──────────┘      └──────────┘   └──────────┘
                  ┌──────────┐   ┌──────────┐
┌──────────┐      │   XDP    │   │   XDP    │
│  Driver  │◄────►│  Prog1   │   │  Prog2   │
└──────────┘      └──────────┘   └──────────┘
packet                 Tail call
```

Figure 3: *The workflow of TC and XDP eBPF development process and its packet flow. The eBPF program compiled by LLVM+clang is loaded into the kernel using iproute. The kernel runs the program through a verification stage, and subsequently attaches the program to the TC/XDP ingress hook point. Once successfully loaded, an incoming packet received by XDP/TC ingress will execute the eBPF program.*

Finally, eBPF tail call [18] is a mechanism allowing one eBPF program to trigger execution of another eBPF program, providing users the flexibility of composing a chain of eBPF programs with each one focusing on particular features. Unlike a traditional function call, this tail call mechanism calls another program without returning back to the caller's program. The tail call reuses the caller's stack frame, which allows the eBPF implementation to minimize call overhead and simplifies verification of eBPF programs.

2.3 XDP: eXpress Data Path

There are several hook points where eBPF programs can be attached in recent Linux kernels. XDP [1, 7] is another eBPF program hook point where its attachment point is at the lowest level of the network stack. XDP demonstrates high performance that closed to the line rate of the device, since the eBPF programs attached to XDP hook point are triggered immediately in the network device driver's packet receiving code path. For the same reason, eBPF program in XDP can only access the packet data and a few metadata. XDP supports accessing to eBPF maps and tail calls, but much less number of helper functions is available compared to the TC hook point.

Figure 3 shows the typical workflow for installing an eBPF program to the TC/XDP hook point, and how packets trigger eBPF execution. Clang and LLVM takes a program written in C and compiles it down to the eBPF instruction set, then emits an ELF file that contains eBPF instructions. An eBPF loader, such as iproute, takes the ELF file, parses the programs and maps information from it and issues BPF syscalls to load the program. If the program passes the BPF verifier, then it is attached to the hook point (in this case, TC/XDP), and subsequent packets through the

TC/XDP ingress hook will trigger execution of the eBPF programs.

2.4 AF_XDP Socket

AF_XDP is a new Linux address family that aims for high packet I/O performance. It enables another way for a userspace program to receive packets from kernel through the socket API. For example, currently, creating a socket with address family AF_PACKET, userspace programs can receive/send the raw packets at the device driver layer. Although the AF_PACKET family has been using in many places such as tcpdump, its performance does not catch up with the recent high speed network devices, such as 40G/100G NICs. Performance evaluation [13, 17] of AF_PACKET shows less than 2 million packets per second using single core.

AF_XDP was proposed and upstreamed to Linux kernel since 4.18 [8]. The core idea behind the AF_XDP is to leverage the XDP eBPF program's early access to the raw packet, and provide a high speed channel from the XDP directly to a userspace socket interface. In other word, AF_XDP socket family connects the XDP packet receiving/sending path to the userspace, by-passing the rest of the Linux networking stacks. An AF_XDP socket, called XSK, is created by using the normal socket() system call. Unlike AF_PACKET which uses the send() and receive() syscalls with packet buffer as parameter, XSK introduces two rings in userspace: the Rx ring and the Tx ring. These two rings are per-XSK data structure that the userspace program needs to properly configure and maintain in order to receive and send packets. Moreover, to provide zerocopy support for AF_XDP, all the packet data buffers used in Tx/Rx rings are allocated from a specific memory region called *umem* which consists of a number of fixed size memory chunks. Two rings, the Fill ring and the Completion ring, are associated with a umem, and a umem can be shared between multiple XSKs. Each element in the Rx/Tx ring or in Fill/Completion ring is a descriptor that contains an address that points to a chunk in umem. The address is not system's virtual or physical address but simply an offset within the umem memory region.

For example, to receive packets from XSK, firstly, the userspace program firstly pushes a set of descriptors pointing to empty packet buffer into the Fill ring. When a packet arrives, kernel pops descriptors from the Fill ring, fills in the data into the memory chunks pointed by the descriptors, and pushes the descriptors back to the Rx ring. The userspace program then checks the Rx ring, fetches the packet data from the descriptors, and refills the empty buffer back to the Fill ring structure, so that kernel can fill in new incoming packets later on. For sending packets, the userspace program pushes a set of descriptors that point to the packet buffers to the Tx ring, then issues sendmsg() system call. Kernel consumes packet buffers asscoaited with the Tx ring, and pushes transmitted descriptors to the Tx ring. The userspace program then checks the Completion ring to determine whether

the packets have been sent. In summary, XSK users need to properly program following four rings:

- Fill ring: for users to fill umem addresses to kernel for receiving packets.
- Rx ring: for users to access received packets.
- Tx ring: for users to place packets needed to be sent.
- Completion ring: for users to check if packets are sent.

Unlike AF_PACKET which is bound to entire netdev, the binding of XSK is more fine-grained. XSK is bound to a specific queue on a device, so only the traffic sent to the queue shows up in the XSK.

3. OVS eBPF Datapath

3.1 eBPF Configuration and Maps

The OVS eBPF datapath consists of one ELF-formatted object file which provides the full functionality of an OVS datapath. This object defines a set of maps and a set of eBPF programs which implement a subset of the datapath functionality. To bootstrap, we load the eBPF program into the kernel using `iproute`. One of the programs is marked within the ELF file to indicate that it should be attached to the hook point; the other programs are only executed via tail calls rooted in the eBPF hook point. The ELF file also defines multiple persistent eBPF maps, which are pinned to the BPF filesystem [2] for sharing between different eBPF programs and `ovs-vswitchd`. The OVS datapath requires the following eBPF maps:

- Flow key. This is the internal representation of the protocol headers and metadata for the current packet being processed, defined in the P4 [6] language.
- Flow Table. This is a hash table whose key is the 32-bit hash value of the flow key, from both packet and its metadata, and value equals an array of actions to execute upon the flow.
- Stats Table. This is similar to the flow table, but rather than holding an array of actions to execute for the packet, it contains packet and byte statistics for the flow.
- Perf Ring Buffer [3, 5]. The perf event map allows an eBPF program to put user-defined data into a ring buffer which may be read from a userspace program. `ovs-vswitchd` memory maps this ring buffer to read packets and metadata from the eBPF program for flow miss upcall processing.
- Program Array. This map allows eBPF programs to tail call other eBPF programs. When the BPF loader inserts eBPF programs into the kernel, it assigns unique indexes for each program and stores these into the map. At run time, an eBPF program will tail call another program by referring to an index within this map.

The OVS eBPF program is triggered by the TC ingress hook associated with a particular network device. Multiple instances of the same eBPF program may be triggered simultaneously on different cores which are receiving traffic from different network devices. The eBPF maps, unless specified, have a single instance across all cores. Access to map entries are protected by the kernel RCU [15] mechanism which makes it safe to read concurrently. However, there are no built-in mechanisms to protect writers to the maps. For the flow table map, OVS avoids the race by ensuring that only `ovs-vswitchd` inserts or removes elements from the map from a single thread. For flow statistics, atomic operations are used to avoid race conditions. Other maps such as the flow key perf ring buffer maps use per-cpu instances to manage synchronization.

3.2 Header Parsing and Lookup

When a packet arrives on the TC ingress hook, the OVS eBPF datapath begins executing a series of programs, beginning with the parser/lookup program as shown in Figure 4. The eBPF parser program consists of two components; standard protocol parsing and Linux-specific metadata parsing. The protocol parsing is executed directly on the packet bytes based on standardized protocols, while the platform-specific metadata parsing must occur on the *context* provided by the eBPF environment.

The resulting code will assemble the protocol headers and metadata, collectively known as the flow key. This flow key is then used to look for an entry in the flow table map, to get an array of actions to execute. If there is no entry in the flow table map, then the packet and the flow key will be written to the perf event map for further processing by `ovs-vswitchd`.

3.3 Action Execution

When a lookup is successful the eBPF gets a list of actions to be executed, such as outputting the packet to a certain port, or pushing a VLAN tag. The list of actions is configured in `ovs-vswitchd` and may be a variable length depending on the desired network processing behaviour. For example, an L2 switch doing unknown broadcast sends packet to all its current ports. The OVS datapath's actions is derived from the OpenFlow action specification and the OVSDB schema for `ovs-vswitchd`.

One might expect to intuitively write an eBPF program to iterate across the list of actions to execute, with each iteration of the loop dealing with a particular action in the list. Unfortunately, this type of iteration implies dynamic loops, which are restricted within the eBPF forwarding model. Moreover, the variable number of actions also implies that there is no way to guarantee the bounded program size, which is limited to 4,096 eBPF bytecode instructions.

To solve these challenges, we first break each type of action logic into an independent eBPF program and tail call from one eBPF action program to another, as shown in Figure 4. This alleviates the problem from having 4k instruc-
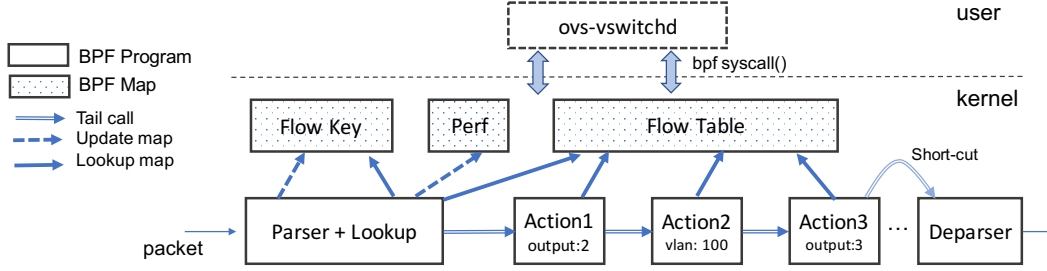
Figure 4: *The overall architecture of OVS eBPF datapath consists of multiple eBPF programs which are tail-called dynamically, maps which are shared between eBPF programs and userspace applications, and ovs-vswitchd as the management plane for all components.*

tions for the *entire* action list to 4k instructions *per action*. Our proof-of-concept implementation shows that this limitation is sufficient for all existing actions. Additionally, the design allows each action to be implemented and verified independently. Second, to solve the dynamic looping problem, we convert the variable length list of actions into a fixed length 32-element array. As a result, flow table lookup always returns an array of 32 actions to be executed, and the LLVM compiler unrolls the loop to pass the BPF verifier. If a matching flow has less than 32 actions to execute, the rest of the actions is no-op, and we short-cut the action execution to the deparser. If a matching flow has more than 32 actions, then the eBPF datapath delegates the execution to userspace datapath, i.e., the slow path. For many use cases, this is sufficient; there is room to further optimize this path in the future if common cases require more than 32 actions.

Each action also requires certain action-specific metadata to execute. For example, an output action would require an *ifindex* to indicate the output interface, while a push_vlan action needs the *VLAN ID* to be inserted into the VLAN tag. To accommodate this, the array element not only contains the action to execute, but also the metadata required by the particular action. The size of each element must be big enough to hold the metadata for any action, as it is a fixed sized array. Future work may relax this requirement.

Each eBPF program executes in an isolated environment. As a consequence of breaking the action list into individual eBPF action programs, some state needs to be transferred between the current eBPF program and the next. The state to transfer from one eBPF action program to another includes (1) The flow key, so that the currently executing eBPF action program can lookup the flow table, and (2) The action index in the actions list, so that the eBPF action program knows it is executing the Nth element in the list, and at the end of its processing, to tail-call the $n+1$th action program. Currently eBPF could use either per-CPU maps as scratch buffers or the context's packet control block (cb) to pass data between programs. In our design, we use one 32-bit value in the cb to keep the action index, and per-cpu maps to save the flow key.

Figure 4 also demonstrates an example eBPF execution of a packet forwarded to port 2 as well as port mirroring

to a VLAN port with VLAN ID 100 at port 3. Once the packet is parsed, the flow table lookup returns an action list of output:2, push_vlan:100, output:3. At the end of the parser+lookup program, it tail calls the eBPF output program as the first step to kick start action execution. The execution of this output program *overwrites* the caller's stack, so it has to look up the flow table map to retrieve the flow key, and execute the output action. Once done, the output action increments the action index from 0 to 1, saves it in cb, and tail-calls the next action program, which is push_vlan. The push_vlan eBPF program again looks up the flow table, and fetches the action metadata at index 1 and executes the push VLAN action using a BPF helper function. The third output action follows the same procedure and finally sends the packet out to port 3.

3.4 Flow Miss Upcall and Installation

One of the important tasks of the OVS datapath is to pass any packet that misses its flow table to the slow path to get instructions for further processing. In the existing OVS *kernel* datapath implementation, the missed packet is sent to ovs-vswitchd, which processes the packet, inserts a flow into the flow table, and re-injects the packet into the kernel datapath using the Linux generic netlink interface. For the eBPF datapath, this design implies two requirements: (1) a way for the eBPF program to communicate with userspace, and (2) a mechanism for the userspace to re-insert packet into the eBPF program.

To address the first requirement, we use the support for Linux perf ring buffers and the skb_event_output() eBPF helper function [5] which allows the eBPF program to pass data to the userspace through the Linux perf ring buffer [5]. During miss upcall processing, the eBPF program will insert the full packet and the current flow key into the perf ring buffer. To receive the data from userspace, ovs-vswitchd runs a thread which maps the ring buffer to its virtual memory using mmap system call, and polls the buffer using the poll system call. If there is incoming data from the ring buffer, the thread is woken up, it reads from the buffer and processes the packet and metadata. The result of this processing will be inserted into the flow table map. To address the second requirement, we construct a dedicated

Linux Bridge	OVS Kernel	Linux TC	OVS eBPF
1.6 Mpps	1.4 Mpps	1.9 Mpps	1.12 Mpps

Table 1: Baseline port-to-port forwarding rate using existing Linux subsystems and OVS kernel and eBPF.

Linux TAP device which also has the OVS eBPF datapath program attached to it. `ovs-vswitchd` sends the missed packet using an AF_PACKET socket, triggering the underlying eBPF program for further processing. This program is very similar to the previously-used parser+lookup program, with minor changes. Specifically, this packet was originally received on one device, however when `ovs-vswitchd` sends the packet on the TAP device, the eBPF program is triggered for the TAP device instead. So, the platform metadata for incoming port misidentifies the source as the TAP device. To ensure that the packet lookup occurs correctly, `ovs-vswitchd` prepends the port number to the beginning of the packet data, then when the eBPF program for the dedicated TAP device processes the packet, it will read this port number into the metadata, then strip this port from the packet. The resulting packet is identical to the originally-received packet, and now the metadata will match the metadata originally generated by the parser+lookup the first time the packet was received. The rest of the lookup, actions execution, and deparser is then executed as per the description in the previous sections.

3.5 Evaluation

To quantify the performance of the OVS eBPF datapath, we measure the packet forwarding rate in millions of packets per second (Mpps), using 64-byte packets under different forwarding scenarios. The hardware testbed consists of two Intel Xeon E5 2650 servers, each with an Intel 10GbE X540-AT2 dual port NIC, with the two ports of the Intel NIC on one server connected to the two ports on the identical NIC on the other server. The OVS eBPF datapath is installed on one server, acting as a bridge to forward packets from one port on the NIC to the other, and vice-versa. The other server acts as packet generating host, which runs DPDK packetgen sending at the maximum packet rate of 14.88Mpps. This server sends to one port to the target server, and receives the forwarded packets on the other port. All experiments use only one CPU core running Linux kernel 4.9-rc6.

Baseline Forwarding Performance. We start by conducting a simple port-to-port packet forwarding experiment, i.e., receiving packets from one port and outputting to the other, using the Linux native bridge, Linux TC, and OVS kernel datapath, as shown in Table 1. The native Linux bridge is a simple L2 mac-learning switch with no programmability, showing the forwarding rate of 1.6 Mpps. The OVS kernel module, which does additional flow key extraction, shows a slower forwarding rate of 1.4 Mpps. For forwarding packets using TC, we have an eBPF program loaded into TC, with the program only having one BPF helper func-

tion call, the bpf_skb_redirect, that redirects the incoming packet from one interface to another. Since TC accesses the incoming packets closest to the driver layer, it shows the highest performance of 1.9 Mpps. Finally, we measure our proposed OVS eBPF forwarding speed, with incoming packets traversing through an eBPF parser+lookup program, an output action program, and the deparser program. Since the OVS eBPF is implemented based on tail calling these additional eBPF programs at TC, we observe the overhead of 0.78 Mpps, a reduction from 1.9 Mpps to 1.12 Mpps.

Forwarding with action execution. To further investigate the overhead, we program one additional action executed before the packet is forwarded to the other port. Since the Linux bridge has no programmability, we only compare the OVS kernel datapath with the OVS eBPF datapath. Table 2 shows forwarding packets while executing the additional action type: hash, push_vlan, set_dst_mac, and set_gre_tunnel, with the OVS kernel and eBPF datapaths. The forwarding performance of these results is bounded by the CPU cost. In the case of the hash and vlan actions, the NIC allows the processing to be offloaded from the CPU, resulting in minimal performance overhead for executing these actions. As a result, the forwarding rate exhibits little to no overhead above the baseline. Moreover, for the hash and push_vlan actions, due to the NIC offloading of actions processing, the datapath does not need to execute any modification to the packet contents, so the deparser program execution may be optimized out.

To ensure invoking the deparser, the third experiment, set_dst_mac, involves altering packet data, specifically to modify the destination MAC address. In this case, the deparser writes stored metadata from actions processing back to the packet. The performance in this experiment drops to 0.84 Mpps. The drop of 0.28 Mpps from baseline is the result of the deparser writing all known packet header metadata back to the packet, and due to extraneous memory copies introduced by the P4-to-eBPF compiler. With a more intelligent deparser implementation, and by using recent improvements to the eBPF API [4, 19], the performance gap compared with the native kernel implementation is expected to shrink.

Finally, the set_get_tunnel action experiment yielded 0.48 Mpps, which represents the additional cost of tunnelling traffic; such traffic must traverse the Linux network stack twice, once for overlay traffic and once for underlay traffic. The eBPF result is also more expensive than the existing Linux implementation, but with a narrower performance gap than the earlier experiments. The majority of processing in this case occurs outside of OVS, so the overhead of the eBPF datapath has less effect.

Action	eBPF DP	Kernel DP	Overhead
hash	1.12	1.34	16%
push_vlan	1.11	1.32	15%
set_dst_mac	0.84	1.28	34%
set_gre_tunnel	0.48	0.57	8%

Table 2: Comparison of single core forwarding rate in Mpps with eBPF and Kernel Datapath, with the additional action executed before forwarding the packet to another port.
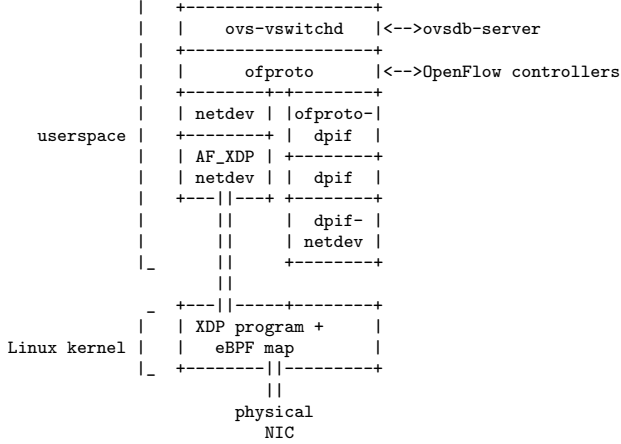
```
|  +------------------+
|  |   ovs-vswitchd   |<-->ovsdb-server
|  +------------------+
|  |     ofproto      |<-->OpenFlow controllers
|  +--------+-+--------+
|  | netdev | |ofproto-|
userspace |  +--------+ | dpif   |
|  | AF_XDP | +--------+
|  | netdev | | dpif   |
|  +---||---+ +--------+
|       ||   | dpif-  |
|       ||   | netdev |
|_      ||   +--------+
        ||
_  +---||-----+--------+
|  | XDP program +     |
Linux kernel |  |   eBPF map       |
|_ +--------||---------+
             ||
           physical
             NIC
```

Figure 5: OVS Architecture AF_XDP

4. Userspace OVS with AF_XDP

4.1 Datapath and Netdev Interface

OVS provides a userspace datapath interface (dpif) implementation, called dpif-netdev. The dpif-netdev userspace datapath receives and transmits packets from its userspace interface. One major use case of dpif-netdev is OVS-DPDK, where the packet reception and transmission are all conducted in DPDK's userspace library. The dpif-netdev is designed to be agnostic to how the network device accessing the packets, by an abstraction layer called netdev. Therefore, packets can come from DPDK packet I/O library, a Linux AF_PACKET socket API, or in our case, the AF_XDP socket interface, as long as each mechanism implements its own netdev interface. Once dpif-netdev receives a packet, it follows the same mechanism performing parse, lookup the flow table, and apply actions to the packet.

Figure 5 shows the architecture of userspace OVS with AF_XDP. We implement a new netdev type for AF_XDP, which receives and transmits packets using the XSK. We insert a XDP program and a eBPF map which interacts with XDP program to forward packets to the AF_XDP socket. Once the AF_XDP netdev receives a packet, it passes the packet to the dpif-netdev for packet processing.

4.2 AF_XDP netdev configuration

When users attach a AF_XDP netdev to an OVS bridge, for example, by issuing the following commands:

```
ovs-vsctl add-br br0
ovs-vsctl add-port br0 eth0 -- \
    set int eth0 type="afxdp"
```

ovs-vswitchd does the following steps to bring up the AF_XDP netdev:

1. Attach a XDP program to the netdev's queue: OVS attaches a simple and fixed XDP program to each netdev's queue. The program only consists of a few lines of code, which receives the packets and redirects them to XSK by calling the bpf_redirect_map() helper function.

2. Create a AF_XDP socket: Call socket() syscall to create a XSK, set up its Rx and Tx ring buffer, allocate a umem region, and set up Fill/Completion ring of the umem.

3. Load and configure the XSK eBPF map: The XSK eBPF maps consists of key value pairs, where key is an u32 index and value is the file descriptor of the XSK. ovs-vswitchd programs an entry to the map with the key as queue id and the file descriptor, fd, of the XSK as its value. Therefore, the XDP program calling bpf_redirect_map will derive the corresponding XSK with the queue id.

4. Populate the umem Fill ring: Get a couple of umem elements and place into Fill ring.

Finally, when a AF_XDP netdev is detached or closed by user, ovs-vswitchd closes the XSK socket, free the umem memory region, and unload the eBPF program and map.

4.3 umem memory management

In order to properly program Rx/Tx/Fill/Completion rings, we implement a umem memory management layer, call umempool. Basically, it's a data structure maintaining the unused/available elements in umem with GET and PUT access APIs. We will demonstrate the use case of umempool with packet reception and transmission in the following subsections.

4.3.1 Packet Reception

Figure 6 shows how ovs-vswitchd sets up the Fill ring and the Rx ring for receiving packets from XSK. For simplicity, in this example, we assume that there are only eight umem buffers, and each buffer's size is 2KB. Initially, at step 1, ovs-vswitchd pushes four available umem elements into the Fill ring and waits for incoming packets. When there are incoming packets, at step 2, the Fill ring's four buffer elements have been consumed and moved to the Rx ring. In order to keep receiving packets, ovs-vswitchd gets another four available umem elements from the umempool, and fills into Fill ring (step 3). Then, ovs-vswitchd creates the metadata needed for the four packet buffer {1, 2, 3, 4}, i.e., struct dp_packet and struct dp_packet_batch, and passes to the dpif-netdev layer for parse, lookup and action executions. Finally, when ovs-vswitchd finishes processing the umem buffer, a recycle mechanism is triggered to place this buffer back to
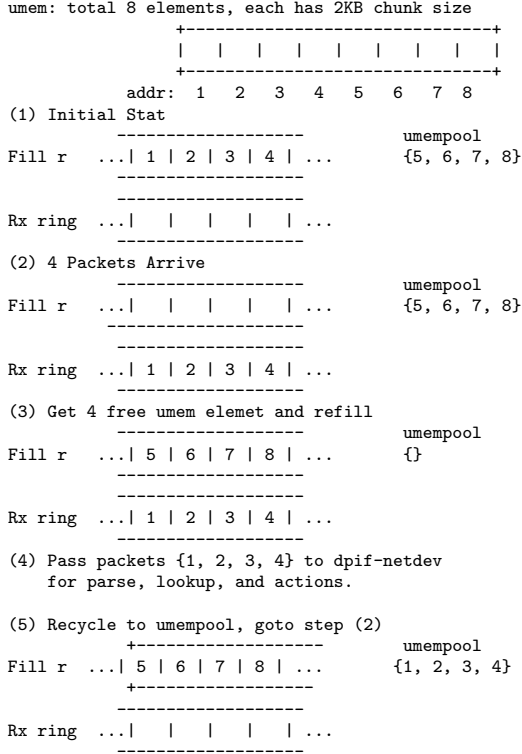
```
umem: total 8 elements, each has 2KB chunk size
              +-----------------------------+
              | | | | | | | | |
              +-----------------------------+
         addr: 1  2  3  4  5  6  7  8
(1) Initial Stat
         -------------------         umempool
Fill r  ...| 1 | 2 | 3 | 4 | ...     {5, 6, 7, 8}
         -------------------
         -------------------
Rx ring ...|   |   |   |   | ...
         -------------------
(2) 4 Packets Arrive
         -------------------         umempool
Fill r  ...|   |   |   |   | ...     {5, 6, 7, 8}
         -------------------
         -------------------
Rx ring ...| 1 | 2 | 3 | 4 | ...
         -------------------
(3) Get 4 free umem elemet and refill
         -------------------         umempool
Fill r  ...| 5 | 6 | 7 | 8 | ...     {}
         -------------------
         -------------------
Rx ring ...| 1 | 2 | 3 | 4 | ...
         -------------------
(4) Pass packets {1, 2, 3, 4} to dpif-netdev
    for parse, lookup, and actions.

(5) Recycle to umempool, goto step (2)
         +-------------------        umempool
Fill r  ...| 5 | 6 | 7 | 8 | ...     {1, 2, 3, 4}
         +-------------------
         -------------------
Rx ring ...|   |   |   |   | ...
         -------------------
```

Figure 6: An example of how OVS programs the Fill ring and Rx ring when processing incoming AF_XDP packets.

umempool (step 5). Step 5 makes sure that there are always available elements in Fill ring, so that the underlying XDP program in kernel can keep processing packets while the userspace ovs-vswitchd is processing the previous received packets on Rx ring. When step 5 finishes, ovs-vswitchd goes back to step 2, waiting for new packets.

4.3.2 Packet Transmission

Figure 7 shows the process for sending packets to the XSK. In this example, there is one bridge and two ports, eth1 and eth2, in OVS. Both ports are configured with AF_XDP support, and there is a flow, in_port=eth1, actions=output:eth2 that forwards packets received from eth1 to eth2.

Initially, assume ovs-vswitchd receives four packets from eth1's XSK. To send packets to eth2 using eth2's XSK, ovs-vswitchd first gets four packet buffers, {4, 5, 6, 7} from eth2's umempool and copies the packet data from eth1's umem to eth2's umem at {4, 5, 6, 7}. Later on, TX descriptors for the four packets are created and placed into eth2's Tx ring (step 2). At step 3, sendmsg syscall is issued to signal the kernel to start the transmission. As the send msg syscall in XSK is asynchronous, ovs-vswitchd needs to poll the COMPLETION ring to make sure that these four packets have been sent out (step 4). Once the four packets' descriptors show up at the COMPLETION ring, at step 5, ovs-vswitchd recycles their umem elements back to eth2's
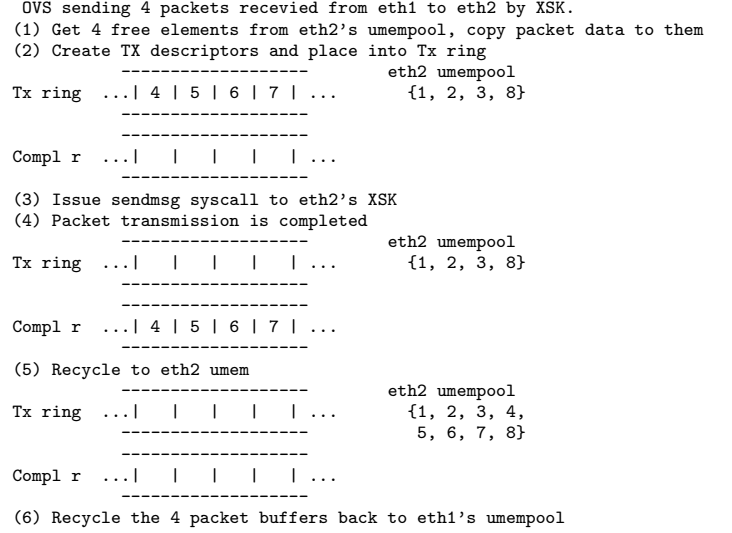
```
OVS sending 4 packets recevied from eth1 to eth2 by XSK.
(1) Get 4 free elements from eth2's umempool, copy packet data to them
(2) Create TX descriptors and place into Tx ring
         -------------------         eth2 umempool
Tx ring ...| 4 | 5 | 6 | 7 | ...     {1, 2, 3, 8}
         -------------------
         -------------------
Compl r ...|   |   |   |   | ...
         -------------------
(3) Issue sendmsg syscall to eth2's XSK
(4) Packet transmission is completed
         -------------------         eth2 umempool
Tx ring ...|   |   |   |   | ...     {1, 2, 3, 8}
         -------------------
         -------------------
Compl r ...| 4 | 5 | 6 | 7 | ...
         -------------------
(5) Recycle to eth2 umem
         -------------------         eth2 umempool
Tx ring ...|   |   |   |   | ...     {1, 2, 3, 4,
         -------------------          5, 6, 7, 8}
         -------------------
Compl r ...|   |   |   |   | ...
         -------------------
(6) Recycle the 4 packet buffers back to eth1's umempool
```

Figure 7: An example of how OVS programs the COMPLETION ring and Tx ring when processing incoming AF_XDP packets from one netdev and sending them to another netdev.

umempool. The original four packet buffers from eth1 are recycled back to eth1's umempool as well.

4.4 Performance Evaluation and Optimization

All of our performance evaluation are conducted on a test bed consisting of two Intel Xeon E5 2440 v2 1.9GHz servers, each with 1 CPU socket and 8 physical cores with hyperthreading enabled. The first server, the source server, is served as a traffic generator. It is equipped with a Netronome NFP-4000 40GbE device, and runs a customized traffic generator that generates 64-byte single UDP flow packets at the rate of $19Mpps$ using the DPDK library. The second server, the target server, is equipped with an Intel 40GbE XL710 single port NIC, and it runs Linux kernel 4.19-rc4 with i40e driver supporting the AF_XDP zero copy mode [11, 12]. We install OVS-AFXDP on the target server, and we disable the Intel Spectre and Meltdown fixes [9, 21] for demonstrating the best performance. In addition, we enable 1GB huge memory page support to reduce the page fault overhead.

For all our experiments, we use a microbenchmark program, called xdpsock, as the baseline to compare with OVS-AFXDP implementation. xdpsock is an AF_XDP sample program that can be configured into two modes. In the rx-drop mode, xdpsock simply drops all incoming packets without any packet processing. In the l2fwd mode, xdpsock forwards the packet to the same port from where it is received. In our testbed, we measure $19Mpps$ for xdpsock-rxdrop and $17Mpps$ for xdpsock-l2fwd. For OVS-AFXDP, we conduct two similar experiments, with OVS's OpenFlow rule installed as below:

- OVS-AFXDP rxdrop: Install a single OpenFlow rule to drop every packets, e.g., in_port=eth1, actions=drop.

- OVS-AFXDP l2fwd: Install a single OpenFlow rule to forward packet to the same port as it receives, e.g., `in_port=eth1, actions=set_fields:eth2->in_port, output:eth1`.

AF_XDP can provide the following three operation modes depending on the device and driver support. We list the three mode from the slowest to the fastest throughput. 1) skb mode: works on devices using generic XDP [16], 2) driver mode: works on devices with XDP support, and 3) zero-copy driver mode: works on devices with XDP_XDP zerocopy support. For all our experiments, we use the zero-copy driver mode since it provides the best performance.

4.4.1 PMD netdev

In the beginning, our initial prototype of OVS-AFXDP only provides $0.5Mpps$ throughput for rxdrop. We observe that when dropping/forwarding packets under OVS-AFXDP, two processes below can easily reach 100% CPU utilization:

- ovs-vswitchd: This is the process that keeps doing the send and receive steps in Figure 6 and 7.

- ksoftirqd/core_id: This is the kernel software interrupt thread handling the incoming packets, triggering XDP program to pass packets to the XSK, and also processing transmission.

With further investigation using Linux perf tools, our first improvement is to to enable OVS's Poll-Mode-Driver (PMD) mode to the afxdp netdev.

In OVS's non-PMD mode, ovs-vswitchd does packet reception by putting all the receiving netdev's file descriptors (fd)s together and invokes the poll system call to determine if any one of the fd is ready to perform I/O. As a result, the polling of XSK's fd is *shared* with other fds, and we observed that the poll system call incurs high overhead.

Applying OVS's PMD netdev avoids these problems and improves the rxdrop performance from $0.5Mpss$ to $3Mpps$. This performance gain is because of reducing userspace and kernel space context switch overhead by omitting poll system call. That is instead of using poll system call, we implement packet reception by using PMD netdev for AFXDP, so that ovs-vswitchd allocate a dedicated thread for each XSK's receive queue and the thread keeps polling the RX ring for new packets. Currently, each round of receive polling processes a batch up to 32 packets.

4.4.2 Memory Management and Locks

For every stage of optimizations, we use Linux perf extensively, e.g., perf stat and perf record/report. With the above PMD netdev optimization, we observe the new bottleneck is the umempool APIs that we introduced. We introduce two major umempool APIs, umem_elem_get(), which gets N free element from the umempool, and umem_elem_put(), which places back the free umem buffer to the umempool. We im-

plement three different data structures for umempool management as below.

- LIFO-list_head: We reverse the first 8-byte of a umem chunk as a next pointer that pointing to the next available element in umempool, and implement LIFO (last in, first out) access get/put APIs. In this design, we maintain the list_head pointer that always points to the first available umem element, and we get and put elements from list_head.

- FIFO-ptr_ring: This design maintains an extra FIFO (first in, first out) pointer ring that is similar to Linux kernel's ptr_ring. The ring is an array of elements that allocated in a continuous memory region, where each element points to an available umem chunk. We keep track of head and tail pointer for the FIFO-ptr_ring. The consumer gets elements from the tail pointer, and then the producer puts elements back to where head pointer points to.

- LIFO-ptr_array: This design is similar to Linux kernel's skb_array [20] where an array of pointers are allocated in contiguous memory region. Each element in the array points to an available memory chunk in umem. We keep track of a head pointer to the array, and both consumer and producer get/put elements from the head pointer.

Initially, we allocate one umem per netdev. Since there might be multiple queues per netdev sharing the same umem, the above three data structures require a mutex lock to protect umempool accessing. In this design, Linux perf reports pthread mutex lock related APIs as one of the top CPU utilization function. We then change our design by 1) allocating per-queue umem region and 2) allocating one PMD thread per queue. As a result, no lock is needed because each queue has only one thread and its own set of umem elements.

4.4.3 Metadata Allocation

Moving forward, Linux perf shows that the packet metadata allocation takes a lot of CPU cycles, i.e., dp_packet_init(), dp_packet_set_data(). So instead of allocating packet metadata at packet reception time, we pre-allocate the packet metadata and implement two data structures to compare their performance:

- Embedding in packet buffer: As we already allocate 2KB chunk for each umem packet buffer, we reserve the first 256-byte in each umem chunk as struct dp_packet and initialize the dp_packet's packet independent fields at allocation time. This is similar to the DPDK mbuf design [10], where a single memory buffer is used for both packet data and metadata.

- Separate from packet buffer: This design allocates a contiguous memory region storing an array of packet metadata, and initialize their packet independent fields.

With the above design change, we find that using the LIFO-ptr_array in Section 4.4.2 with separated packet meta-

Projects	xdpsock	OVS-AFXDP
rxdrop	19Mpps	19Mpps
l2fwd	17Mpps	14Mpps

Table 3: Performance comparison of the xdpsock and OVS-AFXDP.

data allocation yields the best performance. It is because both data structures have better spatial locality and there are more batching friendly. For example, accessing 32 packet metadata in an array incurs less cache misses than accessing the packet metadata 32 umem chunks. The ptr_array for umempool management outperforms the other two designs for the similar reason. With the above design decisions, the OVS rxdrop can achieve similar performance as the baseline xdpsock rxdrop at around $19Mpps$.

4.4.4 Batching Send Syscall

With all the aforementioned optimizations, We continue measuring the performance of OVS-AFXDP l2fwd and observe only $4Mpps$, compared to $17Mpps$ baseline xdpsock-l2fwd. We find that the OVS PMD thread under rxdrop has much fewer context switches compared to the l2fwd, indicating that the PMD process spends much more time in kernel space than in userspace. By using strace, we find that the OVS-AFXDP l2fwd experiment calls sendto system call at very high rate. It is because we design to check the completion of send immediately after issuing send as shown in step 3 and 4 in Figure 7. We modify the design by calling sendmsg syscall (step 3) only when TX ring is close to full, e.g., when 3/4 ring elements have been used. For example, instead of issuing sendmsg syscall for a batch of 32 packets and making sure they are finished, we only issue send when there are 512 outstanding packets. With this change, the OVS-AFXDP l2fwd experiment can achieve around $14Mpps$.

4.4.5 Summary

We summarize the OVS-AFXDP in Table 3. Through the step-by-step analysis, the keys to performance improvement are to keep packet processing in userspace to avoid kernel userspace context switch, and allocating dedicated userspace processes for packet processing. Moreover, standard optimization techniques such as batching is critical to performance. We observe performance boost when apply batching to a couple of places such as issuing sendmsg syscall, packet reception and transmission.

Although there are still rooms for improvement, we are now working on making the patch upstream to the OVS code base for more people to use.

5. Conclusion

This paper describes two eBPF projects related to OVS: OVS-eBPF and OVS-AFXDP.

References

[1] XDP: eXpress Data Path. URL https://www.iovisor.org/technology/xdp, 2018.

[2] A.; Borkmann, D.; Starovoitov and H. F. Sowa. bpf: add support for persistent maps/progs. In *Linux kernel, commit b2197755b263.*, 2015.

[3] D. Borkmann. bpf: avoid stack copy and use skb ctx for event output. In *Linux kernel, commit 555c8a8623a3.*, 2016.

[4] D. Borkmann. bpf: direct packet write and access for helpers for clsact progs. In *Linux kernel, commit 36bbef52c7eb.*, 2016.

[5] D. Borkmann and A. Starovoitov. bpf: add event output helper for notifications/sampling/logging. In *Linux kernel, commit bd570ff970a5.*, 2016.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[7] Jesper Dangaard Brouer. Xdp express data path, intro and future use-cases. *NetDev 1.2*, 2016.

[8] Jonathan Corbet. Accelerating networking with AF_XDP. URL https://lwn.net/Articles/750845/, 2018.

[9] Jesper Dangaard Brouer. XDP performance regression due to CONFIG_RETPOLINE Spectre V2. URL https://lkml.org/lkml/2018/4/12/285, 2018.

[10] DPDK Guide. Mbuf Library. https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html, April 2017.

[11] Magnus Karlsson and Björn Töpel. AF_XDP zero-copy support for I40E. URL https://patchwork.ozlabs.org/cover/962906/, 2018.

[12] Magnus Karlsson and Björn Töpel. Zero-copy support for AF_XDP. URL https://lwn.net/Articles/756549/, 2018.

[13] Magnus Karlsson, Björn Töpel, and John Fastabend. AF_PACKET V4 and PACKET_ZEROCOPY. In *Netdev Conference 2.2*, 2017.

[14] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, volume 46, 1993.

[15] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium Conference Proceedings*, page 175. Ottawa Linux Symposium, 2001.

[16] David Miller. Generic XDP. https://lwn.net/Articles/720072/, April 2017.

[17] Pavel Odintsov. Capturing packets in linux at a speed of millions of packets per second without using third party libraries. URL https://kukuruku.co/post/capturing-packets-in-linux-at-a-speed-of-millions-of-packe

[18] A. Starovoitov. bpf: allow bpf programs to tail-call other bpf programs. In *Linux kernel, commit 04fd61ab36ec.*, 2015a.

[19] A. Starovoitov. bpf: direct packet access. In *Linux kernel, commit 969bf05eb3ce.*, 2016.

[20] Michael S. Tsirkin. skb_array: array based fifo for skbs. URL `https://lwn.net/Articles/689537/`, 2016.

[21] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. URL `https://support.google.com/faqs/answer/7625886`, 2018.