Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges and Applications

MARCOS A. M. VIEIRA*, MATHEUS S. CASTANHO, RACYUS D. G. PACÍFICO, ELERSON R. S. SANTOS, EDUARDO P. M. CÂMARA JÚNIOR, and LUIZ F. M. VIEIRA, Universidade Federal de Minas Gerais, Brazil

Extended Berkeley Packet Filter (eBPF) is an instruction set and an execution environment inside the Linux kernel. It enables modification, interaction and kernel programmability at runtime. eBPF can be used to program the eXpress Data Path (XDP), a kernel network layer that processes packets closer to the NIC for fast packet processing. Developers can write programs in C or P4 languages and then compile to eBPF instructions, which can be processed by the kernel or by programmable devices (e.g. SmartNICs). Since its introduction in 2014, eBPF has been rapidly adopted by major companies such as Facebook, Cloudflare, and Netronome. Use cases include network monitoring, network traffic manipulation, load balancing, and system profiling. This work aims to present eBPF to an inexpert audience, covering the main theoretical and fundamental aspects of eBPF and XDP, as well as introducing the reader to simple examples to give insight into the general operation and use of both technologies.

CCS Concepts: • **Networks → Programming interfaces**; **Middle boxes / network appliances**; End nodes.

Additional Key Words and Phrases: Computer Networking, Packet processing, Network functions

ACM Reference Format:
Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2019. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges and Applications. *ACM Comput. Surv.* X, X, Article X (June 2019), 35 pages.

1 INTRODUCTION

The increase in Internet traffic and the growing complexity of services offered in data center networks have required ever-higher packet processing rates. Also, the dynamicity of service demands requires the network to adapt quickly to maintain adequate levels of quality of service and use available resources efficiently. However, computer networks have been traditionally developed in a static way, embedding the implementation of communication protocols in the hardware of network devices, making it difficult to adapt to current demands.

In recent years several proposals have been made to add more programmability to networks. Among them, we can highlight the SDN [28, 43] and NFV [48] paradigms, new computer systems and languages such as POF [61], P4 [14], and more recently the extended Berkeley Packet Filter

*This is the corresponding author

Authors' address: Marcos A. M. Vieira, mmvieira@dcc.ufmg.br; Matheus S. Castanho, matheus.castanho@dcc.ufmg.br; Racyus D. G. Pacífico, racyus@dcc.ufmg.br; Elerson R. S. Santos, elerson@dcc.ufmg.br; Eduardo P. M. Câmara Júnior, epmcj@dcc.ufmg.br; Luiz F. M. Vieira, lfvieira@dcc.ufmg.br, Universidade Federal de Minas Gerais, Av. Antônio Carlos, 6627 Prédio ICEx, Belo Horizonte, MG, 31270-901, Brazil.

0360-0300/2019/6-ARTX $15.00
https://doi.org/

(eBPF) and the eXpress Data Path (XDP) Linux kernel network layer. This work presents eBPF and XDP assuming little to no background about the subject by the reader.

eBPF provides an instruction set and an execution environment inside the Linux kernel. It is used to modify the processing of packets in the kernel and also allows the programming of network devices. The developer writes an application in restricted C language and then compile the code into eBPF instructions. The resulting eBPF code can be processed in the kernel or by programmable devices such as SmartNICs.

XDP is the lowest layer of Linux network stack [31]. It enables developers to install programs that process packet into the Linux kernel. These programs will be called for every incoming packet. XDP is designed for fast packet processing applications while also improving programmability. In addition, it is possible to add or modify these programs without modifying the kernel source code. eBPF programs modify the (programmable) kernel operation in run-time, not requiring recompilation of the kernel.

The importance of eBPF and XDP is perceived by its fast adoption since its introduction in the Linux kernel in 2014, both by industry and academia. Their use cases have grown rapidly to include tasks such as network monitoring, network traffic handling, load balancing, and operating system insight. Several companies already use eBPF on projects such as Facebook [26], Netronome [10], and Cloudflare [11].

We organized this tutorial as follows: the remainder of this section introduces the reader to the original BPF. In § 1.2, the architecture of the eBPF machine is described. § 2 presents the eBPF system. In § 3, we describe aspects of eBPF programs, such as their structure, the types of programs available, what maps are and how to use them, the types of maps available, what helper functions are, and interaction from user space with libbpf library. In § 4, we explain how eBPF uses hooks and present two of them: the XDP and the TC. § 5 shows examples of eBPF programs and points the reader to extra material. In § 6, some useful tools for developing and debugging eBPF programs are listed. § 7 describes the existing software and hardware platforms that can process eBPF instructions. § 8 discusses some existing industry-led, research and open source projects. § 9 presents the current limitations on eBPF and suggestions on how to overcome them. Finally, § 10 compares eBPF with other similar technologies and § 11 concludes this work.

All code in this paper was tested using kernel version 5.0. A stable version of the extra material is accessible on Zenodo [67]. Step-by-step instructions on how to compile, load and run each example shown throughout this text, including a VM with all tools and dependencies necessary to develop eBPF programs are available on Github [66].

1.1 BPF

Inspired by previous work on in-kernel packet filters [50], the Berkeley Packet Filter (BPF) [46] was proposed by Steven McCanne and Van Jacobson in 1992, as a solution to perform packet filtering on the kernel of Unix BSD systems. It consisted of a set of instructions and a virtual machine (VM) for executing programs written in that language.

Initially, the bytecode of an application was transferred from the user space to the kernel, where it was then checked to assure security and prevent kernel crashes. After passing the verification, the system attached the program to a socket and ran on each arriving packet. The ability to securely run programs provided by the user in the kernel proved to be a good design choice of BPF. Another highlighting factor of BPF was its simple and well-defined set of instructions. Furthermore, there existed a Just-In-Time (JIT) compilation engine for BPF in the kernel. Together, all these factors were fundamental for the good performance of the tool.

Besides the bytecode instructions, BPF also defines a packet-based memory model (load instructions are implicitly made in the processed packet), two registers: accumulator (A) and index register

(X), an implied program counter, and a temporary auxiliary memory. The left side of Figure 1 (Classic BPF machine) illustrates the BPF machine architecture.

The Linux kernel has supported BPF since version 2.5. There were no major changes to the BPF code until 2011, when the BPF interpreter was modified to be a dynamic translator [25]. Instead of interpreting the BPF byte code, the kernel was now able to translate BPF programs directly into x86 instructions.

One of the most prominent tools that use BPF is the `libpcap` library, used by the `tcpdump` tool. When using `tcpdump` to capture packets, a user can set a packet filtering expression so that only packets matching that expression are actually captured. For example, the expression `"ip and tcp"` captures all IPv4 packets that contain the TCP transport layer protocol. This expression can be reduced by a compiler to BPF bytecode. Code 1, based on bpf man page [35], is a BPF program that filters packets to capture only TCP segments. The mnemonic were expanded for clarity.

Code 1 BPF program example based on [35] to only allow IPv4 TCP segments.

```
1 load 2 bytes @ [12]
2 jump equal #0x800 jump true 3 jump false 6
3 load 1 byte @ [23]
4 jump equal #6  jump true 5 jump false 6
5 return #-1
6 return #0
```

Basically, what Code 1 does is:

- Instruction (1): loads two bytes (16 bits) at offset 12 of the frame, into the accumulator. The offset 12 represents the packet type in the Ethernet frame.
- Instruction (2): compares the accumulator value with 0x800, which is the EtherType value for IPv4. If the result is true, the program counter jumps (jumptrue) to instruction (3), otherwise jumps (jumpfalse) to instruction (6).
- Instruction (3): loads the offset 23 of the frame, as a byte, into the accumulator. The offset 23 represents the protocol field of the IPv4 packet. The count is from the beginning of the Ethernet frame.
- Instruction (4): compares the value with the constant 6 (value of the IPv4 packet protocol field for a TCP segment). If true, skip to instruction (5), otherwise go to instruction (6).

The packet filtering program executes until it returns a result, which is usually a boolean. Returning a value other than zero (instruction (5)) means that the packet has matched the filter, whereas returning zero (instruction (6)) indicates the packet does not match the filter and therefore will be discarded.

1.2 Extended BPF (eBPF)

Although BPF was very useful for packet filtering, the community came to realize that other areas could also benefit from its ability to instrument the kernel. In order to transform it into a *universal in-kernel virtual machine* [21], a lot of improvements were introduced to both the BPF machine and its overall architecture. This new version is called eBPF (extended BPF), or simply BPF, while the original iteration became cBPF (classic BPF). eBPF was introduced in version 3.15 of the Linux kernel. The content in this section is based on the eBPF specification [60].

The right side of Figure 1 illustrates the eBPF engine. The number of registers has increased from 2 to 11 (of which 10 are write-registers), the registers width has changed from 32 bits to 64 bits, the instruction set is now 64 bits, and the new engine has a stack of 512 bytes. Global data stores, called maps, were also included, allowing programs to persist data between executions and

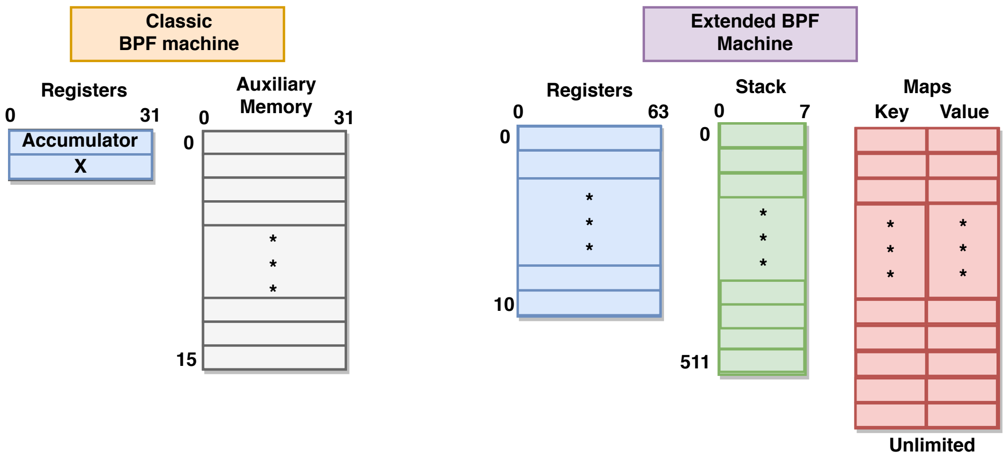

Fig. 1. BPF and eBPF processors.

share information between each other and with user space. It was also added the option to call functions that run inside the kernel, called helper functions [60].

In cBPF, it was necessary to define the jumps for true and false cases in a program. In eBPF, it is only necessary to define the true jumps, and the false jumps follow the execution sequence of the program (called jump-fall-through).

The eBPF's instruction set architecture (ISA) was updated to include function calls. Those calls follow the C calling convention. Parameters are passed to functions through registers, just as it happens in native hardware. This allows mapping an eBPF function call to one hardware instruction, which results in almost no overhead. eBPF uses this feature to enable helper functions, allowing programs to make system calls and manipulate storage (maps). The eBPF virtual machine supports dynamic loading and program reloading. This way, programs can be changed on run-time, modified, or reloaded again if necessary.

Table 1 describes the functionality of each eBPF register. Register `r0` stores the function return value, which indicates, at the end of the computation, what action will be taken in the forwarding of the packet. Register `r10` is the only read-only register, and it stores the address to the BPF stack.

As eBPF follows the C calling convention, arguments are passed as register values to functions. Thus registers `r1`-`r5` are reserved for this purpose, while registers `r6`-`r9` have their values preserved between function calls.

Table 1. Description of the eBPF register set.

Register	Description
`r0`	return value from functions and programs
`r1 - r5`	arguments passed to functions
`r6 - r9`	registers that are preserved during function calls
`r10`	stores frame pointer to access the stack

2 EBPF SYSTEM

The eBPF system is composed of a series of components to compile, verify, and execute the source code of developed applications. This section describes more details on each of them.

2.1 Overview

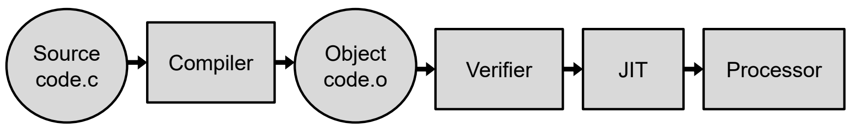

Fig. 2. eBPF Workflow.

The typical workflow of the eBPF system is illustrated in Figure 2. An eBPF program is written in a high-level language (mainly restricted C). The clang compiler transforms it into an ELF/object code. An ELF eBPF loader can then insert it into the kernel using a special system call. During this process, the verifier analyzes the program and upon approval the kernel performs the dynamic translation (JIT). The program can be offloaded to hardware, otherwise it is executed by the processor itself.

2.2 Compiler

Starting in version 3.7, the LLVM compiler collection has a backend for the eBPF platform. It allows the development of eBPF programs in a subset of C and generation of executable code in eBPF format through the clang compiler.

This subset of C excludes some syscalls and libraries, but it provides helper functions to manipulate eBPF maps and to perform other common tasks. Partial solutions to these restrictions are presented later in § 9.

The main restrictions are:

- eBPF can only use a subset of C language libraries. For example, the *printf()* function is not available for use;
- Non-static global variables are not allowed;
- Only bounded loops are allowed.
- Stack space is limited to 512 bytes.

There are also efforts from open source projects such as IOVisor [17, 32] and from VMWare [68] to implement a P4 [14] compiler for eBPF. Early versions already exist but are still not ready for production. Also, the BPF Compiler Collection (BCC) project [7] enables extra abstractions over the standard C code to facilitate writing and interacting with eBPF programs, as well as `libbpf` (§ 3.6), the main upstream library for user space interaction with eBPF.

2.3 Verifier

To ensure the integrity and security of the operating system, the kernel uses a verifier that performs static program analysis of eBPF instructions being loaded into the system. Its implementation is available at `kernel/bpf/verifier.c` in the kernel source code.

Among other things, the verifier checks if a program is larger than allowed (current limit is 10^6 instructions), whether or not the program ends, if the memory addresses are within the memory range allowed for the program, and how deep the execution path is. It is called after the code has been compiled and during the process of loading the program into the data plane. A good overview is presented by Miller [49].

The verifier uses two passes to decide whether to reject a program or not. In the first pass, it uses a depth-first search to check if the program instructions can be parsed into a Directed Acyclic Graph (DAG). eBPF programs that do not have backward jumps or that have only predefined size loops (which can thus have loop unroll) can be synthesized into a DAG, guaranteeing their termination. Moreover, the DAG is useful to check for unreachable instructions (the graph must have only one connected component) and to compute the worst-case execution time.

The second pass explores all possible paths from the program's first instruction. It does so by creating a state machine, where it verifies if states present correct behaviors and also keeps records of the ones it has already checked [59]. The verifier uses the states already checked for pruning and so be able to reduce its amount of work to do. It also limits the maximum length of paths to analyze. This limit was initially 64k instructions, but currently equals to the maximum program size allowed.

It is also worth mentioning two more points about the eBPF verifier. The first one is related to the fact that some eBPF functions can only be called by programs with GPL compatible licenses. Because of that, the verifier checks if the licenses of the functions used by a program and the program's license are compatible, and rejects the program if they are not.

Lastly, the verifier does not allow memory accesses beyond the local variables and packet boundaries to ensure the integrity and security of the kernel. To access any bytes in the packet, it is always necessary to perform a border check (as shown later in § 5.2.1). However, each byte only needs to be checked once, unless the storage space of the packet gets modified. This way, during the analysis of the program, the verifier guarantees that all memory accesses made to the packet are in checked addresses. If the eBPF program does not do this type of check, the verifier rejects it, and so it can not be loaded in the kernel [31].

3 EBPF PROGRAMS

Several types of applications can be implemented with eBPF, e.g., performance analysis, packet filtering, and traffic classification, to name a few. The main advantage of the eBPF system as a whole is offering a flexible and safe programmable environment inside the Linux kernel. For example, eBPF programs can be loaded and modified during runtime and are capable of interacting with kernel elements such as kprobes, perf events, sockets, and routing tables [44].

However, the subsystems and functionalities available to an eBPF program depend on where it is loaded in the kernel, i.e., which layer or subsystem it is attached to, which is defined by a program's type. In this section, we discuss different eBPF program types, present key-value store data structures called maps, and also show some helper functions available to eBPF programs.

3.1 How and when are eBPF programs executed?

To execute an eBPF program, it is first necessary to attach it to an interface that allows custom programming. This interface is called a hook. Hooks allow the registration of programs for certain events. In § 4, we describe two Linux kernel hooks to which eBPF programs can be attached, XDP and TC.

eBPF programs execute whenever there is an event for which they were registered. In Computer Networking, common events are sending or receiving a packet.

3.2 Program types

Each eBPF program has a type, which determines three important aspects: what is the input passed to it (its context), which helper functions it is allowed to use, and to which kernel hook it will be attached. For example, two of the many types of eBPF programs are socket filter and tracing. The input parameter for a socket filter program is a socket buffer, containing packet metadata generated by the kernel but stripped of L2 and L3 information. A tracing program, on the other hand, receives a set of register values. Also, the subsets of helper functions available for these two types are not the same, although there is an overlap of common general-purpose functions.

Supported program types are defined on the header file `linux/bpf.h` by the *enum* `bpf_prog_type`. On version 5.3-rc6, the kernel offers a total of 25 valid different program types, some of which are listed below:

- **`BPF_PROG_TYPE_SOCKET_FILTER`**: program to perform socket filtering;
- **`BPF_PROG_TYPE_SCHED_CLS`**: program to perform traffic classification at the TC layer;
- **`BPF_PROG_TYPE_SCHED_ACT`**: program to add actions to the TC layer;
- **`BPF_PROG_TYPE_XDP`**: program to be attached to the eXpress Data Path hook;
- **`BPF_PROG_TYPE_LWT_{IN, OUT or XMIT}`**: programs for Layer-3 tunnels;
- **`BPF_PROG_TYPE_SOCKET_OPS`**: program to catch and set socket operations such as retransmission timeouts, passive/active connection establishment etc;
- **`BPF_PROG_TYPE_SK_SKB`**: program to access socket buffers and socket parameters (IP addresses, ports, etc) and to perform packet redirection between sockets;
- **`BPF_PROG_TYPE_FLOW_DISSECTOR`**: program to do flow dissection, i.e., to find important data in network packet headers.

These program types are related to networking, which is the focus of this work. However, there are other program types for kernel tracing/monitoring (e.g., `BPF_PROG_TYPE_PERF_EVENT`, `BPF_PROG_TYPE_KPROBE` and `BPF_PROG_TYPE_TRACEPOINT`), cgroups (e.g., `BPF_PROG_TYPE_CGROUP_SKB` and `BPF_PROG_TYPE_CGROUP_SOCK`) and others [44, 51]. The full list of supported program types can be obtained directly from the kernel source code with the following command:

```
$ git grep -W 'bpf_prog_type {' include/uapi/linux/bpf.h
```

3.3 Maps

Maps are generic key-value stores available to eBPF programs. Keys and values are treated as binary blobs, allowing the storage of user-defined data structures and types, whose sizes must be informed during map definition.

Maps are created using the `bpf` system call, allowing map manipulation through the map's file descriptor. This is done by passing the command `BPF_MAP_CREATE` (defined by `enum bpf_cmd`) and the `bpf_attr` union with extra parameters to the `bpf` system call:

```
bpf( BPF_MAP_CREATE, &bpf_attr, sizeof(bpf_attr) ).
```

In this case, the following attributes should be set on `bpf_attr`:

(1) *map_type*: type of the map to be created;
(2) *key_size*: number of bytes to store the key;
(3) *value_size*: number of bytes to store the value;
(4) *max_entries*: number of rows in the map.

A user-space process can create multiple maps, and they can be accessed by both user-space processes and eBPF programs loaded in the kernel, enabling data exchange between the two environments. To access a map, an eBPF program needs to declare a special global variable, of type `struct bpf_map_def` (defined by `libbpf`), in the *maps* ELF section. During the load process, the file loader uses the syscall above to create any declared maps and pass their file descriptors to the program, which are later converted into actual pointers by the verifier for use at run time. Code 2 shows an example where a `BPF_PROG_TYPE_ARRAY` map named `mapname` is declared.

Code 2 Map declaration example.

```
struct bpf_map_def SEC("maps") mapname = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof( uint32_t ),
    .value_size = sizeof( long ),
    .max_entries = 256,
};
```

3.3.1 Map Types. There are many different map types available for eBPF programs, and they are defined in the *enum* `bpf_map_type`, from `linux/bpf.h`. Each map type provides a different behavior, some of them being used generically, while others have specific use cases.

Examples of eBPF map codes are provided by the kernel. Version 5.3-rc6 of the kernel lists a total of 24 valid different map types. Some of them are:

- **BPF_MAP_TYPE_ARRAY**: a map where entries are indexed by a number, as in a high-level programming language array. It follows the RAM model, where the input to query an item is an address.
- **BPF_MAP_TYPE_PROG_ARRAY**: a map that stores references to eBPF programs. Its use allows, for example, the call of subprograms to deal with specific situations.
- **BPF_MAP_TYPE_HASH**: stores entries using a hash function.
- **BPF_MAP_TYPE_PERCPU_HASH**: a map that is similar to the **BPF_MAP_TYPE_HASH**. Allows the creation of a hash table for each processor core.
- **BPF_MAP_TYPE_LRU_HASH**: a map that stores entries using hash function. When the table is full, the policy to remove element LRU, i.e., the elements to be removed are the ones that were last used the longest.
- **BPF_MAP_TYPE_LRU_PERCPU_HASH**: allows the creation of a hash table for each processor core with LRU remove policy.
- **BPF_MAP_TYPE_PERCPU_ARRAY**: a map that is similar to the **BPF_MAP_TYPE_ARRAY**. Allows the creation of an array for each processor core.
- **BPF_MAP_TYPE_LPM_TRIE**: longest-prefix match (LPM) trie.
- **BPF_MAP_TYPE_ARRAY_OF_MAPS**: an array to store references to eBPF maps.
- **BPF_MAP_TYPE_HASH_OF_MAPS**: a hash table to store references to eBPF maps.
- **BPF_MAP_TYPE_DEVMAP**: stores reading references of network devices.
- **BPF_MAP_TYPE_SOCKMAP**: stores socket references. It can be used to implement socket redirection, for example.
- **BPF_MAP_TYPE_QUEUE**: a map with behavior similar to that of a queue.
- **BPF_MAP_TYPE_STACK**: a map with behavior similar to that of a stack.

The list of all supported map types can be obtained directly from the kernel source code with the following command:

```
$ git grep -W 'bpf_map_type {' include/uapi/linux/bpf.h
```

3.3.2 Lifetime of maps and map pinning. Every eBPF object (programs, maps, and debug info) has a reference counter (`refcnt`) that is maintained by the kernel [63]. When a user-space process creates a map with the call `bpf_create_map()`, the kernel initializes the map `refcnt` to 1. The kernel then increments the map `refcnt` whenever a new eBPF program that uses the map is loaded and decrements it whenever one of them is closed. The map `refcnt` will also be decremented when the process that created it exits (or crashes). When a `refcnt` reaches zero, a memory free is trigged, destroying the eBPF object related to the counter. This flows represents, in a simple way, the lifetime of an eBPF map.

The scheme just described allows sharing the same eBPF map between several programs at once. It keeps the map alive as long as its parent process or some eBPF program that uses it is alive. However, there is also another way to keep eBPF maps alive: by doing map pinning.

A user-space process can pin a map (or any other eBPF object) to the BPF file system, a minimal kernel space file system located at `/sys/fs/bpf/`. When a map is pinned to this file system, the kernel increments its `refcnt`, which allows it to stay alive even though no program is using it. Similarly, when a map is unpinned, its `refcnt` gets decremented and again it may be destroyed if it is not being used.

The map pinning can be done in several ways: by using the `bpf()` system call from user space (with command `BPF_OBJ_PIN`), through `libbpf` (§ 3.6), by using `bpftool` (§ 6.2) or by using a special map structure provided by `iproute2` (§ 6.1). This alternative structure, called `bpf_elf_map`, is compatible with the one provided by the kernel, and can be used by eBPF programs instead of `bpf_map_def`. It exposes extra members, such as `pinning` which can be used to define the map scope. This field can receive three distinct values: `PIN_GLOBAL_NS`, `PIN_OBJECT_NS`, and `PIN_NONE`.

Maps created with `PIN_OBJECT_NS` have local scope, being unique to the program that declared them. As a consequence, maps with the same declaration can co-exist in different programs. In this case, a specific directory will be created in the BPF file system to store the nodes corresponding to those maps. If the value `PIN_OBJECT_GLOBAL` is used, the map is created with a global scope, enabling it to be shared by multiple programs. This map will receive an entry in the directory `globals` in the pseudo-file system. `PIN_NONE` indicates that the map should not be fixed in the file system, disabling sharing it with other applications. Finally, a map can be unpinned by removing its file from the BPF file system. This removal can be done using the syscall `unlink()`.

3.3.3 Locked memory. eBPF maps use locked memory, which is a resource that is usually limited by many systems. Default limits may be too low, which may cause programs to be rejected at load time. To overcome this restriction, increase the locked memory limit to a sufficient one or even remove it entirely. This limit can be changed with `ulimit -l <size>`.

3.4 Helper functions

eBPF differs from cBPF in several ways, one being the ability to allow programs to call the so-called helper functions. These are special functions offered by the kernel infrastructure to enable interaction with the context of each hook and other kernel facilities and structures, such as maps, routing tables, tunneling mechanisms, etc.

Tasks performed by helper functions include interacting with maps, modifying packets and printing messages to the kernel trace. Since there are many program types, and each has a specific execution context, the list of functions callable by a specific function represents a subset of all helper functions implemented by the kernel, which varies depending on the hook the program is attached to. For example, function `bpf_xdp_adjust_tail()` is used to remove the last bytes of a packet, effectively decreasing the packet's size. However, as the name indicates, it is only available at the XDP hook. The BCC project maintains a list of helper functions for each program type [8].

The helper functions available to eBPF programs are restricted to the list provided and implemented by the kernel. The addition of new helper functions can only be done through extensions to the kernel source code, since extensions through kernel modules are not allowed. New functions should follow a calling convention shared by all functions on eBPF programs, limiting the maximum number of input parameters to 5. Parameter passing is done through the use of registers `r1-r5`, requiring no interaction with the stack.

The number of helper functions available is large and increases constantly with new kernel versions. Version 5.3-rc6 offers a total of 109 such functions. Some of these are highlighted below:

- **`bpf_map_delete_elem`**, **`bpf_map_update_elem`**, **`bpf_map_lookup_elem`**: used to remove, install or update, and search elements from maps, respectively;
- **`bpf_get_prandom_u32`**: returns a 32-bit pseudo-random value;
- **`bpf_l4_csum_replace`**, **`bpf_l3_csum_replace`**: used to recalculate Layer-4 and Layer-3 checksums, respectively;
- **`bpf_ktime_get_ns`**: returns time since system boot, in nanoseconds;
- **`bpf_redirect`**, **`bpf_redirect_map`**: functions to redirect packets to other network devices. The second allows specifying the device dynamically through a special redirection map;
- **`bpf_skb_vlan_pop`**, **`bpf_skb_vlan_push`**: remove/add, respectively, VLAN tags from a packet;
- **`bpf_getsockopt`**, **`bpf_setsockopt`**: similar in functionality to user-space calls to **`getsockopt()`** and **`setsockopt()`** to get/set socket options.
- **`bpf_get_local_storage`**: returns a pointer to a local storage area. Depending on the program type, this area can be shared between multiple program instances running in parallel.

The declarations of helper functions are spread across several header files included in the directory `tools/testing/selftests/bpf` in the kernel source code. However, most of them are in `bpf_helpers.h`. Some common operations to perform endianness conversion are declared by `bpf_endian.h`, placed in the same folder. This file offers BPF-compatible versions of well-known functions like `ntohs()` and `htons()`, in the form of `bpf_ntohs()` and `bpf_htons()`, for example.

As explained earlier, the kernel provides the implementation of these functions, and eBPF programs only need to be compiled against the header files containing their signature, with no need for their `.c` counterpart. This can be done by passing the path to these files in the kernel source code to `clang` using the `-I` flag. However, it is also possible to make a local copy of the header files needed and avoid compiling against the kernel tree, making the code easier to compile and to distribute.

3.4.1 Tail calls. eBPF programs can call other program to run next, never returning to the caller, via tail calls. They can be used, for example, to simplify complex programs and build dynamic chains of programs [62]. Tail calls are implemented as long jumps, and they reuse the current stack frame to avoid creating a new one, leading to minimal overhead when compared to function calls. The use of tail calls involves the use of (i) a specialized map, called program array (**`BPF_MAP_TYPE_PROG_ARRAY`**), to store references of eBPF programs, and (ii) a helper function (**`bpf_tail_call`**) to execute the tail calls. The program array can be filled by user space with key-value pairs, where the values are the file descriptors of the eBPF programs. The helper function receives three arguments: the context, a reference to the program array map, and a lookup key.

Tail calls have some limitations, however. As the chain of tail calls can form loops, the maximum number of tail calls is currently limited to 32 to avoid infinite loops. Furthermore, eBPF only allows programs of the same type to be tail called. The same is true for the translation type, which should match the caller's (JITed or interpreted).

3.5 Return codes

The codes returned by eBPF programs vary in meaning and value depending on the program type. For example, an XDP program (§ 4.2), returns a verdict about what should be done with the packet after processing (pass along, drop, redirect, etc), which is defined by `enum xdp_action` in `bpf.h`. TC return codes (§ 4.3) have a similar meaning, but use a different enumeration type. Socket filters, on the other hand, use the return code to indicate the packet length to be passed to the stack, being able to trim or even discard the packet entirely.

3.6 Interaction from user space with libbpf

Although the kernel exposes the bpf() syscall to interact with the eBPF framework from user space, it is a single tool serving many purposes, making it rather complex. A more user-friendly API is offered by libbpf [39], which is a user-space library developed by the kernel community for that matter. It is available under tools/lib/bpf on the kernel source code and is also distributed in a stand-alone version on GitHub [40], which mirrors the corresponding files from the kernel. To include this library, follow the steps on the README to compile the library and link it to your code:

```
$ LIBBPF_DIR=<path-to-libbpf>/src
$ clang -I${LIBBPF_DIR}/root/usr/include/ -L${LIBBPF_DIR} myprog.c -lbpf
```

The root directory can be different based on the DESTDIR used during libbpf compilation. Finally, include the library in the C code:

```c
#include <bpf/libbpf.h>
```

Several examples available in the directory tools/testing/selftests/bpf demonstrate use cases of this library, and can serve as a good starting point. Also, the stand-alone version on GitHub has detailed instructions on how to build a integrate libbpf into projects without requiring compiling against the kernel source.

This API includes a few direct wrappers of the bpf() system call and exposes several structures to help the interaction with the eBPF system. For example, the user can handle information about maps, programs, and object files using struct bpf_map, struct bpf_program, and struct bpf_object, respectively. Each of these object-like types has specific getters and setters, whose names start with the name of the structure, followed by a double underscore and a declarative name of the action to be performed. The following paragraphs list some of the most common functions for each of these object types.

After compiling a .c file containing eBPF programs with clang, the object file generated will contain several ELF sections corresponding to each program. A user-space program can interact with such file using the bpf_object__* family of functions. Some examples include:

- **bpf_object__open**, and **bpf_object__open_xattr**: read an object file and returns a pointer to a struct bpf_object. The _xattr version allows specifying the program type;
- **bpf_object__load**, and **bpf_object__load_xattr**: load the programs from a struct bpf_object into the kernel. The _xattr version allows specifying the desired log level;
- **bpf_object__pin_maps**: allows handling pinning of all maps from an object file;
- **bpf_object__for_each_program**: macro to iterate over each program from an object file;
- **bpf_object__find_program_by_title**: returns the handle to a BPF program based on its section name.

Some of the functions above also have their respective counterparts (*unload*, *close*, *unpin*).

Another useful set of functions included in the API allows handling programs separately, and can be used with the program iterator showed above, for example, to apply specific actions to each program in a file. These functions have a bpf_program__* signature, and some are shown below:

- **bpf_program__is_<type>**, and **bpf_program__set_<type>**: getters and setters, respectively, for a program's type. Each type has its own pair of functions, in which <type> is replaced by the corresponding name (e.g. sched_cls, sched_xdp, etc).
- **bpf_program__load**: loads a given program to the kernel;
- **bpf_program__fd**: returns the file descriptor for a BPF program;
- **bpf_program__pin**: pins a given program to an specific file path;
- **bpf_program__set_ifindex**: sets a device's *ifindex* to offload maps and programs to.

At last, functions starting with `bpf_map__*` support actions on map objects, ranging from creation, information retrieval, reuse, pinning, etc.

- **`bpf_map__def`**: returns basic map information (type, size, etc);
- **`bpf_map__reuse_fd`**: allows the reuse of an existing map when loading a new program;
- **`bpf_map__resize`**: used to change the number of maximum entries allowed in a map;
- **`bpf_map__fd`**: returns the file descriptor for a given map;
- **`bpf_map__for_each`**: macro to iterate over all maps in an object file;

This section is not supposed to give an extensive discussion of all available functions, but rather give the reader a glimpse of the facilities offered by `libbpf`. The parts of the API that were left out include interaction with perf buffers, preprocessor helpers, and more. For the complete list of all available calls, as well as the full signature of the functions shown, please check the `libbpf.h` header file in the source code. Example code using `libbpf` will be shown later in § 5.

3.7 Basic program structure

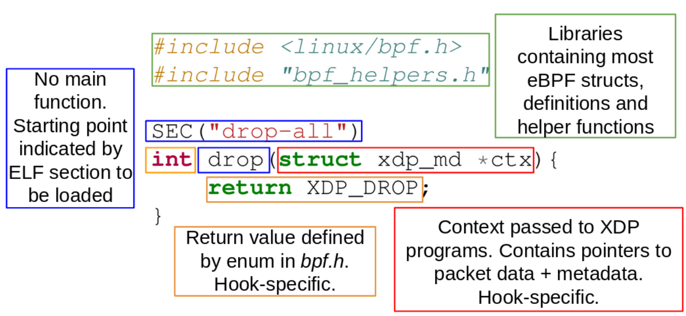

Fig. 3. Dropworld example illustrating the structure of an eBPF program.

Figure 3 illustrates the basic structure of an eBPF program. It presents a simple XDP program that drops all received packets. More details about it will be given in § 4. The library `linux/bpf.h` has all *struct* and constants definitions used by the eBPF programs, except for specific subsystems such as Traffic Control (TC) and *perf*, which need extra header files. As a rule of thumb, all eBPF programs should include this file.

Return values and the input parameter received by programs depend on the hook they will be attached. As shown in Figure 3, XDP programs receive a pointer to a `struct xdp_md`, which is explained in detail in § 4.2.1. Programs on other hooks receive different context structures. See § 5 for example programs on other hooks.

Note that the program shown does not contain a `main` function, usual on standard C programs. The program's starting point is indicated by its section in the ELF object file. When compiled, the program shown will be placed on the default `.text` section. § 5 shows the definition of custom sections.

Below, we show the object file and disassembler output of the program example from Figure 3.

```
0:      b7 00 00 00 01 00 00 00        r0 = 1
1:      95 00 00 00 00 00 00 00        exit
```

The first instruction writes 1 (XDP_DROP) to register `r0`. Remember from Table 1 that `r0` contains the return value of an eBPF program. The second instruction just exits. Now that the reader understands the basic program structure of an eBPF program, the next section covers XDP.

4 NETWORK HOOKS

In Computer Networking, hooks are used for intercepting packets before the call or during execution in the operating system. The Linux kernel exposes several hooks to which eBPF programs can be attached, enabling data collection and custom event handling. Although there are many hook points in the Linux kernel, we will focus on two present in the networking subsystem: eXpress Data Path (XDP) and Traffic Control (TC). Together, they can be used to process packets close to NIC on both RX and TX, enabling the development of many network applications. This section explains how eBPF can be used to program these two hooks and how programs can be loaded to each one.

4.1 Kernel's networking layers

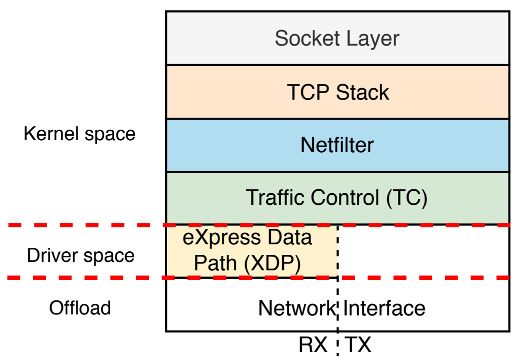

Fig. 4. Linux kernel network stack.

Packets entering the OS are processed by several layers in the kernel, as shown in Figure 4. These layers are: socket layer, TCP stack, Netfilter, Traffic Control (TC), the eXpress Data Path (XDP), and the NIC.

Packets destined to a userspace application go through all these layers and can be intercepted and modified during this process by modules such as `iptables`, which resides in the Netfilter layer. As explained before, eBPF programs can be attached to several places inside the kernel, enabling packet mangling and filtering.

4.2 eXpress Data Path (XDP)

XDP is the lowest layer of the Linux kernel network stack. It is present only on the RX path, inside a device's network driver, allowing packet processing at the earliest point in the network stack, even before memory allocation is done by the OS. It exposes a hook to which eBPF programs can be attached [31].

In this hook, programs are capable of taking quick decisions about incoming packets and also performing arbitrary modifications on them, avoiding additional overhead imposed by processing inside the kernel. This renders the XDP as the best hook in terms of performance speed for applications such as mitigation of DDoS attacks.

After processing a packet, an XDP program returns an action, which represents the final verdict regarding what should be done to the packet after program exit.

4.2.1 ***XDP Input Context.*** The context seen by an XDP program is defined by the single input parameter passed to it by the kernel. It is of type `struct xdp_md`, defined by `bpf.h`, and is reproduced here as Code 3. Upon program execution, the `data` and `data_end` fields contain the pointers to the beginning and the end of packet data, respectively. These values must be used to guide packet access, as further explained in § 5. The third value inside the structure is the `data_meta` pointer, which holds the address of a memory area free to be used by XDP programs to exchange packet metadata with other layers. The last two fields hold the indexes of the interface that received the packet and the corresponding RX queue, respectively. When accessing these two values, the BPF code is rewritten inside the kernel to access the kernel structure `struct xdp_rxq_info` that actually holds those values.

Code 3 Declaration of `struct xdp_md` as-is from `bpf.h`

```
struct xdp_md {
        __u32 data;
        __u32 data_end;
        __u32 data_meta;
        /* Below access go through struct xdp_rxq_info */
        __u32 ingress_ifindex; /* rxq->dev->ifindex */
        __u32 rx_queue_index;  /* rxq->queue_index  */
};
```

Although the first three fields hold pointer values, their C data type is a regular 32-byte unsigned integer. To properly use the memory addresses, a program must first cast them, which is usually done through the following code snippet, present at the beginning of almost all XDP programs:

```
void *data_end = (void *)(long)ctx->data_end;
void *data     = (void *)(long)ctx->data;
```

Here, `ctx` is the input of an XDP program, of type `struct xdp_md` shown above.

4.2.2 ***XDP Actions.*** Table 2 lists all possible XDP actions, their values, and their description. The action is specified as a program return code, which is stored at register `r0` right before the eBPF program exits.

Table 2. Description of XDP action set

Value	Action	Description
0	XDP_ABORTED	Error. Drop packet.
1	XDP_DROP	Drop packet.
2	XDP_PASS	Allow further processing by the kernel stack.
3	XDP_TX	Transmit from the interface it came from.
4	XDP_REDIRECT	Transmit packet from another interface.

The first four actions are a simple return value (no parameters), which indicate the packet should be dropped while raising an exception (`XDP_ABORTED`), dropped silently (`XDP_DROP`), passed along to the kernel stack (`XDP_PASS`) or immediately retransmitted through the same interface (`XDP_TX`).

The `XDP_REDIRECT` action allows an XDP program to redirect packets to (i) another NIC (physical or virtual), (ii) another CPU for further processing, or (iii) an AF_XDP socket for userspace processing. Different from the others, this action requires a parameter to specify the redirection target. This is done through one of two helper functions: `bpf_redirect()` or `bpf_redirect_map()`. The former receives the target's interface index and is focused on network devices. The latter is a more generic alternative, which performs lookups on an auxiliary map to retrieve the final target, which can be both net devices or CPUs. The second option is recommended, since it provides much better performance if compared to `bpf_redirect()` by batching packet transmits, and it also offers better flexibility, as map entries can be modified dynamically from user and kernel spaces.

4.2.3 ***XDP modes of operation.*** For increased performance, eBPF programs attached to the XDP alter the packet processing pipeline at the device driver level, which requires explicit support by the associated network driver. Some drivers for high-speed devices such as `i40e`, `nfp`, `mlx*` and the `ixgbe` family already have such functionality. On devices compatible with these drivers, XDP programs are executed directly by the driver, even before these are handled by the operating system. This is called *XDP Native* mode. BCC Project [9] maintains an up-to-date list of XDP-enabled drivers.

However, the kernel offers a compatibility mode called *XDP Generic*, which enables XDP program execution for devices without native support at the driver level. On this mode, XDP execution is done by the operating system itself, emulating native execution. This way even devices without explicit XDP support can have programs attached to them, at the cost of reduced performance due to socket buffer allocation extra steps required to perform the emulation [47].

The system automatically chooses between these two modes when loading the eBPF program. Once loaded, it is possible to check the mode of operation using the `ip` tool, as shown in the following section.

There is yet another mode of operation, *XDP Offload*. As the name suggests, the eBPF program is offloaded to compatible programmable NICs (§ 7.2.1), achieving even greater performance if compared to the other two modes. This mode should be indicated explicitly when loading the program.

4.2.4 ***XDP and XDP offload example.*** To demonstrate how to compile and load XDP programs, we will use the example from Figure 3. It is a simple XDP program, which drops every packet as soon as they arrive at the network interface.

After saving the example in a `dropworld.c` file, the code can be compiled into an ELF object file using the `clang` compiler:

```
$ clang -target bpf -O2 -c dropworld.c -o dropworld.o
```

The `ip` tool can load the object file into the kernel. In the example code, the program does not have any section tag, so the generated bytecode resides inside the default section (`.text`) in the ELF object file. This section should be specified when loading the program. The `-force` parameter indicates that the program should be loaded even if there is another program loaded on that interface, which will get replaced. The [DEV] parameter should be changed to the corresponding interface name.

```
# ip -force link set dev [DEV] xdp obj dropworld.o sec .text
```

After loading the program, the `ip` tool can also be used to verify that it is attached to the interface on the XDP hook.

```
$ ip link show dev [DEV]
 DEV: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc
mq state UP mode DEFAULT group default qlen 1000
    link/ether 00:16:3d:13:08:80 brd ff:ff:ff:ff:ff:ff
    prog/xdp id 27 tag f95672269956c10d jited
```

The keyword `xdp` on the first line of output indicates that an XDP program is attached to that interface in XDP Native mode. Other possible outputs could be `xdpgeneric` and `xdpoffload` for the other two modes of operation. The program can also be removed from the interface by passing the parameter `off`:

```
# ip link set dev [DEV] xdp off
```

In this last example, the eBPF program was executed at the XDP hook by the driver, using the CPU. To offload programs, the same method as before can be used, but passing the `xdpoffload` parameter to the `ip link set` command:

```
# ip -force link set dev [DEV] xdpoffload obj dropworld.o sec .text
```

As before, to remove the program just execute the following command:

```
# ip link set dev [DEV] xdpoffload off
```

4.3 Traffic Control (TC) Hook

Currently, although the XDP layer is well suited for many applications, it can only process ingress traffic (packets being received). To process egress traffic (transmitting packets), the closest layer to the NIC that has access to the entire Ethernet frame is the Traffic Control (TC) layer.

This layer is responsible for executing traffic control policies on Linux. In it, the network administrator can configure different queuing disciplines (*qdisc*) for the various packet queues present in the system, as well as add filters to deny or modify packets.

The TC has a special queuing discipline type called `clsact`. It exposes a hook that allows queue processing actions to be defined by eBPF programs. Pointers to the packet to be processed are delivered to the configured eBPF program as part of its input context: a `struct __sk_buff`. This structure is a UAPI for certain fields that the program is allowed to access from the kernel's socket buffer internal data structure. It has the same `data` and `data_end` pointers as `struct xdp_md` but also has much more information if compared to the XDP case. This is explained by the fact that at the TC level, the kernel has already parsed the packet to extract protocol metadata, hence the richer context information passed to the eBPF program. The entire declaration of `struct __sk_buff` is omitted for brevity but can be seen on `include/uapi/linux/bpf.h`.

During program execution, the input packet can be modified, and the return value indicates to TC what action should be taken for it. The library `linux/pkt_cls.h` defines the available return values. The most common ones are listed in Table 3.

Table 3. Description of TC set of actions

Value	Action	Description
0	TC_ACT_OK	Delivers the packet in the TC queue.
2	TC_ACT_SHOT	Drop packet.
-1	TC_ACT_UNSPEC	Uses standard TC action.
3	TC_ACT_PIPE	Performs the next action, if it exists.
1	TC_ACT_RECLASSIFY	Restarts the classification from the beginning.

The loading of programs on the TC hook is done using the `tc` tool, available in the `iproute2` package. The following command illustrates how to create the `clsact` *qdisc* and load an eBPF program to process the packets on interface `eth0`:

```
# tc qdisc add dev eth0 clsact
# tc filter add dev eth0 <direction> bpf da obj <ebpf-obj> sec <section>
```

The `<direction>` parameter indicates which direction the program should be associated with, which can be `ingress` or `egress`. `<ebpf-obj>` and `<section>` should be the names of the file containing the compiled eBPF code and the section to load the program, respectively.

To check if there is any program already loaded on `eth0`, use the following command:

```
# tc filter show dev eth0 <direction>
```

For an example of a functional eBPF program for the TC layer and its interaction with the XDP layer, please check our repository on GitHub [66].

4.4 Comparison between XDP and TC

Both hooks can be used for similar applications, such as DDoS mitigation, tunneling and handling link layer information. However, since XDP runs before any socket buffer allocation takes place, it can reach higher throughput values than programs on TC. The latter, on the other hand, can benefit from extra parsed data available through `struct __sk_buff` and execute eBPF programs for both ingress and egress traffic, being the lowest layer on TX.

5 EXAMPLES

This section presents a few examples with in-depth code explanations to help the reader become familiar with eBPF programs. The first example describes a program that allows only IPv4 TCP segments, similar to the BPF example in Code 1. In the second example, we show the interaction between user and kernel spaces through `libbpf`, while the third one shows how programs on the XDP and TC layers can work together to collect statistics. Finally, we point the reader to a few more external examples.

5.1 TCP filter

The first example is a program to only accept packets with TCP segments (Code 4). This is similar to the example in Code 1, but it uses eBPF to drop packets with no TCP segment. Here we present it in two perspectives: the higher level C code, and the the actual eBPF assembly-like code generated after compilation.

5.1.1 ***C code:*** This program was designed to be loaded into the XDP hook, so the input parameter of the function must be of type `struct xdp_md`, as discussed before (§ 4.2.1). The bytes of the packet being processed are delimited by the `data` and `data_end` pointers, which must be used throughout the program to access the packet. Type conversions for these two values are standard, so Lines 9 and 10 should be used at the beginning of every eBPF program that accesses packet data. By using `data`, the parsing of headers can be done with the standard header files provided by Linux.

The main difference, however, to other common packet parsing Linux programs is that bound checks are necessary before actually accessing protocol header data. Since the kernel verifier performs strict memory bound checks (§ 2.3), every access to packet data needs to be covered by an if statement with a border check (Lines 14 and 21). Each byte only needs to be checked once, unless helper functions that modify the storage space of the packet are used (e.g. `bpf_xdp_adjust_head()`). On that case, it is necessary to redo all checking after calling such functions. If an eBPF program does not perform this type of check, it gets rejected during load time by the verifier and is not loaded into the kernel.

After extracting the packet's protocol number, the program checks if it corresponds to the TCP protocol and allows it to go through the stack (Line 26). If not, the packet is just dropped (Line 28).

5.1.2 ***eBPF bytecode:*** Upon compilation, *clang* generates an object file with the eBPF instructions, which can then be loaded into the kernel. As explained in § 2.3, the verifier creates a DAG based on the program. Figure 5 shows the respective DAG the this example. Each DAG node contains one or more eBPF instructions. The conditional jump nodes are the nodes that contain two output lines

Code 4 Example of a C code that checks if the packet contains an IPv4 TCP segment.

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/in.h>
#include "bpf_endian.h"

int isTCP( struct xdp_md *ctx ) {
    void *data_end = (void *)(long) ctx->data_end;
    void *data_begin = (void *)(long) ctx->data;
    struct ethhdr* eth = data_begin;

    // Check packet's size
    if(eth + 1 > data_end)
      return XDP_PASS;

    // Check if Ethernet frame has IPv4 packet
    if (eth->h_proto == bpf_htons( ETH_P_IP )) {
        struct iphdr *ipv4 = (struct iphdr *)( ((void*)eth) + ETH_HLEN );

        if(ipv4 + 1 > data_end)
          return XDP_PASS;

        // Check if IPv4 packet contains a TCP segment
        if (ipv4->protocol == IPPROTO_TCP)
          return XDP_PASS;
    }
    return XDP_DROP;
}
```

and a light gray background. The solid line indicates the next Acyclic Control Flow Graph (ACFG) node. Dotted lines indicate jumps to another ACFG node. In our example, there are three different kinds of conditional jump instructions: *jgt* (jump if greater), *jne* (jump if not equal), and *jeq* (jump if equal). The last number on each of this instructions indicates how many instructions to jump when the condition is valid.

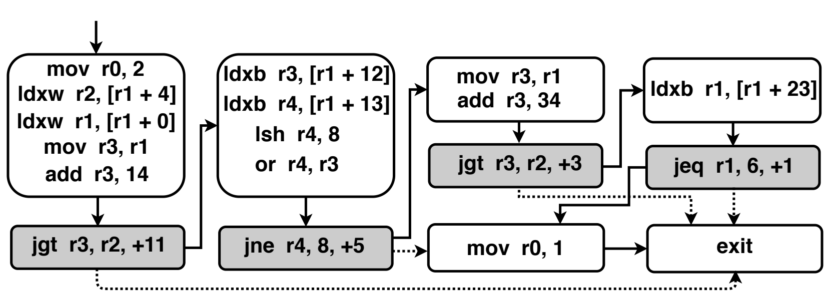

Fig. 5. Directed Acyclic Graph example of Code 4.

To better understand this eBPF program, remember that register `r1` starts with a pointer to the input context stored in main memory and register `r0` stores the return value (Table 1).

In the first node, the first assembler instruction sets `r0` (from Table 1) to 2 (XDP_PASS as seen in Table 2. Moreover, load instructions compute the references to the packet's beginning and ending from the input passed (Lines 10-12). Then the bounds of the Ethernet header are checked to guarantee valid memory access later (required by the verifier) (Lines 14-16). If the check fails, the first jump instruction switches the flow to the *exit* instruction, ending the program. Otherwise, bytes 12 and 13 of the Ethernet header are loaded (counting from 0), which is the Ethernet type field. Then, a byte swap is done to set the endianness (Line 19).

The second jump instruction compares if the Ethernet type is `0x0800` (Line 19). Then the bounds of the IP header are checked to guarantee valid memory access later (again, required by the verifier) (Lines 21-23). Next, the IP protocol field is extracted from the IPv4 packet, and then the program checks if it is the TCP protocol (value 6) (Line 26) (last gray background node in the DAG). Finally, the packet is passed along to the kernel since register (`r0`) was previously loaded with the value 2 (`XDP_PASS`), which indicates acceptance. If the IP packet does not contain a TCP header, the value 1 (`XDP_DROP`) is loaded to `r0`. The last instruction tells that the code terminates.

5.2 User and kernel space interaction

Following, we present the `xdp1` example, extracted directly from the kernel source code, present in the `samples/bpf` directory. It is divided into two parts: `xdp1_kern.c` is the actual eBPF program to be compiled and loaded into the kernel, while `xdp1_user.c` is a user-space counterpart to load the eBPF program into the kernel and interact with it through maps. We show the code for each one and discuss them separately below.

5.2.1 ***Kernel space.*** The file `xdp1_kern.c` [54] contains an eBPF program that processes each packet on the XDP hook, extracts the corresponding IP protocol number, counts the number of packets received per protocol using a per-CPU array map called `rxcnt` and finally drops all incoming traffic.

```c
/* Copyright (c) 2016 PLUMgrid
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of version 2 of the GNU General Public
 * License as published by the Free Software Foundation.
 */
#include <uapi/linux/bpf.h>
#include <linux/in.h>
#include <linux/if_ether.h>
#include <linux/if_packet.h>
#include <linux/if_vlan.h>
#include <linux/ip.h>
#include <linux/ipv6.h>
#include "bpf_helpers.h"

struct bpf_map_def SEC("maps") rxcnt = {
    .type = BPF_MAP_TYPE_PERCPU_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(long),
    .max_entries = 256,
};

static int parse_ipv4(void *data, u64 nh_off, void *data_end) {
    struct iphdr *iph = data + nh_off;

    if (iph + 1 > data_end)
```

```
        return 0;
    return iph->protocol;
}

static int parse_ipv6(void *data, u64 nh_off, void *data_end) {
    struct ipv6hdr *ip6h = data + nh_off;

    if (ip6h + 1 > data_end)
        return 0;
    return ip6h->nexthdr;
}

SEC("xdp1")
int xdp_prog1(struct xdp_md *ctx) {
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct ethhdr *eth = data;
    int rc = XDP_DROP;
    long *value;
    u16 h_proto;
    u64 nh_off;
    u32 ipproto;

    nh_off = sizeof(*eth);
    if (data + nh_off > data_end)
        return rc;

    h_proto = eth->h_proto;

    if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
        struct vlan_hdr *vhdr;

        vhdr = data + nh_off;
        nh_off += sizeof(struct vlan_hdr);
        if (data + nh_off > data_end)
            return rc;
        h_proto = vhdr->h_vlan_encapsulated_proto;
    }
    if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
        struct vlan_hdr *vhdr;

        vhdr = data + nh_off;
        nh_off += sizeof(struct vlan_hdr);
        if (data + nh_off > data_end)
            return rc;
        h_proto = vhdr->h_vlan_encapsulated_proto;
    }

    if (h_proto == htons(ETH_P_IP))
        ipproto = parse_ipv4(data, nh_off, data_end);
    else if (h_proto == htons(ETH_P_IPV6))
        ipproto = parse_ipv6(data, nh_off, data_end);
    else
        ipproto = 0;

    value = bpf_map_lookup_elem(&rxcnt, &ipproto);
    if (value)
```

```
85          *value += 1;
86
87      return rc;
88  }
89  char _license[] SEC("license") = "GPL";
```

The first thing to point out is that a C source file can contain many eBPF programs. They are separated into unique sections in the ELF file generated by the compiler. The section label above the corresponding function (Line 40) indicates to the compiler the name of the ELF section that will contain the program in the generated object file. This information is required while loading the code into the kernel so that the system knows which ELF section to load. Section labels are also used during map (Line 17) and program license (Line 89) declarations, both of which have fixed values. The verifier uses the license section to determine which helper functions will be available to the user, as some of them are restricted to programs declaring GPL compatible licenses.

Similarly to the previous example, the program parses the packet headers up to the IPv4 or IPv6 headers. After determining the Layer-4 protocol type (Lines 77 and 79), the program retrieves the counter for the corresponding protocol using the lookup helper function (Line 83). This function returns a pointer to the current value stored in the map if it exists, or `NULL` otherwise. This address can be used to change the stored data directly, without the need for a map update operation. Finally, the program returns the action that must be taken by the XDP hook for the current packet, which in this case is always `XDP_DROP`, indicating that the packet should be discarded.

5.2.2 ***User space.*** The statistics collected by the kernel and stored on the map `rxcnt` are then queried by a user-space application, implemented by the file `xdp1_user.c` [55]. For brevity, we highlight below only the meaningful parts of this program.

Firstly, instead of using the facilities of `iproute2` as shown in § 4.2.4 to load the program to the kernel, this example implements a custom loader based on `libbpf` (§ 3.6). This method yields a higher degree of control over how the program in `xdp1_kern.c` is loaded to the kernel, which can be modified programmatically from the user-space application. For such, `libbpf.h` and `bpf.h` are included to allow interaction with the eBPF system from user space:

```
// SPDX-License-Identifier: GPL-2.0-only
/* Copyright (c) 2016 PLUMgrid
 */
#include "bpf/bpf.h"
#include "bpf/libbpf.h"
```

The program information to be loaded is passed through the `bpf_prog_load_attr` structure, including the program type, the object file containing the program, and the interface identifier to which it should be associated.

```
struct bpf_prog_load_attr prog_load_attr = {
    .prog_type        = BPF_PROG_TYPE_XDP,
};
```

```
snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);
prog_load_attr.file = filename;
```

This structure is then used to load the program into the XDP hook. In the case of success, after the call, the variables `obj` and `prog_fd` contain the detailed information of the code already loaded and its file descriptor, respectively. The descriptor is used to identify the program from the others currently loaded in the kernel, which is necessary for future interactions with this program.

```
if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd))
    return 1;
```

After loading the eBPF program into the kernel, the reference to the `rxcnt` map is obtained. The function `bpf_map__next` returns an iterator for the list of maps declared in the program. Since in this case there is only one declared map, the value of that iterator can be used to obtain the file descriptor referring to it. The `libbpf.h` library also offers other functions to get map descriptors by name or by index in the map list.

```
map = bpf_map__next(NULL, obj);
if (!map) {
    printf("finding a map in obj file failed\n");
    return 1;
}
map_fd = bpf_map__fd(map);
```

Finally, the eBPF program is ready to be attached to an interface, and the userspace application can enter the infinite loop inside the `poll_stats()` function.

```
if (bpf_set_link_xdp_fd(ifindex, prog_fd, xdp_flags) < 0) {
    printf("link set xdp fd failed\n");
    return 1;
}

poll_stats(map_fd, 2);
```

The poll function `poll_stats()`, using the file descriptor of map `rxcnt`, performs periodic lookups to it and lists all the existing entries along with the statistics calculated so far.

```
35 static void poll_stats(int map_fd, int interval)
36 {
37     unsigned int nr_cpus = bpf_num_possible_cpus();
38     __u64 values[nr_cpus], prev[UINT8_MAX] = { 0 };
39     int i;
40 
41     while (1) {
42         __u32 key = UINT32_MAX;
43 
44         sleep(interval);
45 
46         while (bpf_map_get_next_key(map_fd, &key, &key) != -1) {
47             __u64 sum = 0;
48 
49             assert(bpf_map_lookup_elem(map_fd, &key, values) == 0);
50             for (i = 0; i < nr_cpus; i++)
51                 sum += values[i];
52             if (sum > prev[key])
53                 printf("proto %u: %10llu pkt/s\n",
54                        key, (sum - prev[key]) / interval);
55             prev[key] = sum;
56         }
57     }
58 }
```

Note that since `rxcnt` is a per-cpu array map, stored data is actually spread across multiple CPUs. The helper `bpf_num_possible_cpus` (Line 37) retrieves the number of CPUs used by the program, which is used to set the size of the array which will hold to data from each CPU (`values`). Then, an infinite loop queries the entire map from time to time to retrieve recently collected statistics. This is done with the `bpf_map_get_next_key` iterator function, which yields one map key at a time, enabling iterating through all entries in the map in order. Remember from the kernel-side program that the keys are IP protocols numbers, so given the map size declared (256), iterating through all keys corresponds to iterating through all IP protocol numbers possible.

The user-space version of `bpf_map_lookup_elem` is used (Line 49) to actually read the map values associated with the corresponding key from all CPUs at the same time, which are then stored by the helper function in the `values` array. These values are then added to get the overall statistic (Line 51) and if the value obtained is greater then what was seen the last time, the difference is printed to standard output, showing the user how many packets with that specific protocol number were received during the sampling interval.

This example shows how user and kernel spaces can interact through eBPF programs and maps. All data collection and packet handling on the fast path (kernel) is executed by a minimal, optimized eBPF program, while an agent retrieves the data periodically on the slow path (user space) and can take actions based on it, for example, display to the user. This is a very powerful and useful approach that can be applied and extended to many different scenarios.

5.3 Cooperation between XDP and TC

The following example consists of two separate eBPF programs, one to be attached to the XDP layer and another to TC. Together, they track the number of packets and bytes exchanged between two different IPv4 addresses and store these pieces of information on a map, tracking both RX and TX. This example demonstrates some of the unique facilities offered by `iproute2` (§ 6.1), how programs can be loaded without the need of a custom user-space program, and how programs in different layers can interact through maps.

```c
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <linux/in.h>
#include <linux/bpf.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/if_ether.h>
#include <linux/pkt_cls.h>
#include <iproute2/bpf_elf.h>

#include "bpf_endian.h"
#include "bpf_helpers.h"

struct pair {
    uint32_t lip; // local IP
    uint32_t rip; // remote IP
};

struct stats {
    uint64_t tx_cnt;
    uint64_t rx_cnt;
    uint64_t tx_bytes;
    uint64_t rx_bytes;
};

struct bpf_elf_map SEC("maps") trackers = {
    .type = BPF_MAP_TYPE_HASH,
    .size_key = sizeof(struct pair),
    .size_value = sizeof(struct stats),
    .max_elem = 2048,
    .pinning = 2, // PIN_GLOBAL_NS
```

```
};

static bool parse_ipv4(bool is_rx, void* data, void* data_end, struct pair *pair){
    struct ethhdr *eth = data;
    struct iphdr *ip;

    if(data + sizeof(struct ethhdr) > data_end)
        return false;

    if(bpf_ntohs(eth->h_proto) != ETH_P_IP)
        return false;

    ip = data + sizeof(struct ethhdr);

    if ((void*) ip + sizeof(struct iphdr) > data_end)
        return false;

    pair->lip = is_rx ? ip->daddr : ip->saddr;
    pair->rip = is_rx ? ip->saddr : ip->daddr;

    return true;
}

static void update_stats(bool is_rx, struct pair *key, long long bytes){
    struct stats *stats, newstats = {0,0,0,0};

    stats = bpf_map_lookup_elem(&trackers, key);
    if(stats){
        if(is_rx){
            stats->rx_cnt++;
            stats->rx_bytes += bytes;
        }else{
            stats->tx_cnt++;
            stats->tx_bytes += bytes;
        }
    }else{
        if(is_rx){
            newstats.rx_cnt = 1;
            newstats.rx_bytes = bytes;
        }else{
            newstats.tx_cnt = 1;
            newstats.tx_bytes = bytes;
        }

        bpf_map_update_elem(&trackers, key, &newstats, BPF_NOEXIST);
    }
}

SEC("rx")
int track_rx(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct pair pair;

    if(!parse_ipv4(true,data,data_end,&pair))
        return XDP_PASS;
```

```
90
91      // Update RX statistics
92      update_stats(true,&pair,data_end-data);
93
94      return XDP_PASS;
95  }
96
97  SEC("tx")
98  int track_tx(struct __sk_buff *skb)
99  {
100     void *data_end = (void *)(long)skb->data_end;
101     void *data = (void *)(long)skb->data;
102     struct pair pair;
103
104     if(!parse_ipv4(false,data,data_end,&pair))
105         return TC_ACT_OK;
106
107     // Update TX statistics
108     update_stats(false,&pair,data_end-data);
109
110     return TC_ACT_OK;
111 }
```

Besides standard C types, maps can also handle user-defined structures. For example, `struct pair` and `struct stats` (Lines 15 and 20) are used here as key and value, respectively, for the `trackers` map (Line 27), where communication statistics will be stored. The statistics consist on the number of RX and TX packets and bytes, which are tracked for each unique pair of IPv4 addresses seen by an interface.

Note that the map definition is different from the one in the previous example. The `bpf_elf_map` structure is used by `iproute2`, hence the inclusion of the header file `iproute2/bpf_elf.h`, which is added to the system when `iproute2` is installed from source. Besides slightly different field names, it also contains an extra `pinning` field, which can be used to define the map's scope, as discussed in § 3.3.2. The value set for this field determines there should be a single `trackers` map, which will be shared by both programs, instead of a separate copy for each.

This same source file holds the two programs use, in two different ELF sections. Section `rx` (Line 81) has an XDP program which will process packets as soon as they arrive on the system. The function `parse_ipv4` (Line 35) executes all parsing necessary to extract source and IP addresses from the packet. Besides the pointers to packet data and a buffer to place the address information in, it also receives a boolean value as an input parameter. It indicates if this is an incoming or outgoing packet. Since we want to track each pair of communicating IPs, packets flowing in both directions should be tracked by the same entry in the map. Thus, the program interprets source and destination IP addresses as local or remote. On RX, the destination IP is local (the host's or possibly a guest), whereas the source IP is remote. On TX, it is the other way around.

Once the IP pair is extracted from the packet, it is passed to the function `update_stats` to update the `trackers` map accordingly. It also receives a parameter `is_rx`, which will indicate if the packet seen is incoming or outgoing. The function retrieves the previous statistics for the corresponding address pair (Line 59) and checks if an entry already exists. If not, a new entry is created with a call to `bpf_map_update_elem` with the `BPF_NOEXIST` flag (Line 77). In this case, the return value of `bpf_map_update_elem` is ignored since there is not much the program can do if the call fails. If `stats` is not null, then the values are updated directly using the that pointer (Lines 61-67). The program terminates returning a `XDP_PASS` code, to just pass the packet along to the stack normally.

Since there is no XDP layer on TX, the closest we can get to handling the packet just before sending it to the NIC is by attaching an eBPF program to the TC layer. The program on section `tx` does just that. It is in essence the same as the XDP version, loaded to RX, with just small changes to handle packets going on the other direction. Note that besides the arguments passed to the auxiliary functions, the input context and the return codes are different (as explained in § 4.3) since this program will be loaded on TC.

In this example, two programs on different layers work independently to fill the same map. This same idea can be extended to enable sharing information between programs inside the kernel, in addition to user space.

5.4 Other examples

The examples shown above demonstrate some basic aspects that should be considered during the development of eBPF programs. They also show how to interact with user space programs.

Linux kernel: other useful examples are offered by the source code of the Linux kernel, located in two separate directories: `samples/bpf` and `tools/testing/selftests/bpf`. Both contain programs that demonstrate the use of various features and hooks on kernel stack, with new programs being added to each new version of the kernel. Most of the examples in the first directory are divided into two separate files, one with user space code to load the program (files ending in `user.c`) and the other with the implementation of the eBPF program to be loaded in the kernel (files ending in `kern.c`). In general, examples in this folder represent stand-alone projects that may be useful for various tasks. The examples on the second directory are used as the basis for running functional tests during the development of the kernel. The actual eBPF programs that go into the kernel are placed inside the `progs/` sub-directory, while the remaining files on the root correspond to user-space programs and scripts used to load them.

XDP Project: beyond the examples in the kernel, the official XDP tutorial [70] also presents examples with step-by-step instructions, explaining different features of the XDP hook in detail.

L4 load-balancer: Netronome provides code for an XDP program called `l4lb` that implements an L4 load balancer [53]. The program processes incoming network packets and calculates a hash value based on the source IP address along with TCP or UDP ports, to ensure that all packets from the same flow are processed in the same server. The generated hash is used as a key in an eBPF map. This eBPF map is populated with the address of the available servers for which the program can redirect the packets. This program extends and inserts an external IP header with data from the map. Then, the packet is forwarded to the corresponding server. This program could also be offloaded to a SmartNIC, saving CPU cycles.

Programmable Receive Side Scaling (RSS): RSS allows to map a receiving packet to be processed by a CPU core. This is important to distribute network traffic across multiple CPUs in multiprocessor systems. RSS techniques are used by many network adapters to distribute the computation of packets to a set of distinct CPUs through the use of multiple queues. However, the implementations are usually proprietary and hardware-based, allowing little or no programmability. Using an eBPF program, packet distribution can be modified on demand through map values or full replacement of the loaded eBPF program [53].

6 TOOLS

This section introduces some tools that can be useful for developing and debugging eBPF programs.

6.1 `iproute2`

`iproute2` is a set of user space tools to control, configure, and monitor the kernel's network. `ip` and `tc` are example of these tools. Both offer alternative ways of loading eBPF codes into the kernel without the need for a user space program making use of the libbpf library or bpf system call. Examples on how to use both ip and tc tools were shown previously in this text.

In addition, iproute2 has its own interface to interact with the eBPF system, offering additional functionalities such as the ability to specify the scope of eBPF maps allocation. Unfortunately, its extra features makes it incompatible with libbpf loader. For example, when using iproute2, maps are declared using an alternative structure (bpf_elf_map), defined by iproute2/bpf_elf.h. An example usage is as follows:

```
#include <linux/bpf.h>
#include <iproute2/bpf_elf.h>

struct bpf_elf_map SEC("maps") src_mac = {
    .type = BPF_MAP_TYPE_HASH,
    .size_key = 1,
    .size_value = 6,
    .max_elem = 1,
    .pinning = PIN_GLOBAL_NS,
};
```

This structure is similar to bpf_map_def of libbpf but has extra fields, such as the pinning, used to define the map's scope (which was described in § 3.3.2). So, iproute2 can load libbpf programs, but the opposite is not true as libbpf does not know how to handle the extra functionalities [2].

6.2 bpftool

The bpftool [15] is a user-space debug utility that can also load eBPF programs, create and manipulate maps, and collect information about eBPF programs and maps. It is part of the Linux kernel tree and is available at tools/bpf/bpftool. Here we present just some of its functionalities.

- It can list loaded programs with the command:
  ```
  # bpftool prog show
  ```
- To print the instructions of an specific program, use:
  ```
  # bpftool prog dump xlated id <id>
  ```
- It is possible to list and print the contents of maps at run-time:
  ```
  # bpftool map
  # bpftool map dump id <map id>
  ```
- It can also perform some management operations, including loading programs, performing searches, or updating map values. An example for the last item is:
  ```
  # bpftool map update id 1234 key 0x01 0x00 0x00 0x00 value 0x12 0x34
  ↪ 0x56 0x67
  ```

6.3 llvm-objdump

The llvm-objdump tool (version 4.0 or higher) provides a disassembler to transform the compiled byte code into a format readable for humans before the user attempts to inject it into the kernel. Also, it is useful for inspecting the ELF sections of the compiled eBPF file. The command below shows how to use the disassembler:

```
$ llvm-objdump -S dropworld.o
```

6.4 BPF Compiler Collection (BCC)

The open-source project BPF Compiler Collection (BCC) [7] aims to facilitate the development of eBPF programs. It provides a set of frontends that can be used to interact with the eBPF system using high-level languages such as Python and Go.

The project also has a series of example tools built using BCC capable of performing various tasks in the operating system. These tools can perform tasks such as analyzing the number of system calls by an application, and the time elapsed during a disc read. As they are based on eBPF programs, they can be used to analyze actual production systems with low additional overhead.

7 PLATFORMS

There are several platforms that make use of the eBPF instruction set to add programmability to different environments beyond the Linux kernel. In this section we present the most prominent, divided by their nature: software or hardware.

7.1 Software

7.1.1 ***Linux kernel.*** As discussed previously, the Linux kernel was the origin of eBPF and is the most popular platform, with the most active development, being the main focus of this text. The eBPF applications inside the kernel are not only restricted to network processing, but can also be applied to kernel instrumentation and monitoring. Thus, eBPF programs today represent an import toolset to perform Linux introspection and performance analysis, with important open source projects such as IOVisor [32] offering many solutions on this front.

7.1.2 ***Userspace BPF (uBPF).*** The *uBPF* [65] is an open-source project that adapts the eBPF processor to run in userspace. *uBPF* project has a copy of the eBPF interpreter and JIT compiler stripped of all kernel-specific data structures, allowing users to embed an eBPF machine into other projects running in userspace. By leveraging the *uBPF* source code, one can easily add programmability with eBPF to other tools and environments. Similarly to *uBPF*, project *rbpf* offers an alternative implementation of the eBPF VM in Rust [52]. Since *uBPF* runs on userspace, it does not have the restrictions imposed by the eBPF verifier like requiring unrolling loop.

However, differently from the eBPF system in the kernel, *uBPF* does not offer native support to maps and does not have any helper functions implemented. Nonetheless, it can be easily extended to support these features, as was done by [34], who used it as part of the BPFabric programmable virtual switch, discussed next.

7.1.3 ***BPFabric.*** To deal with the limitations of OpenFlow, [34] propose a new SDN architecture called BPFabric. BPFabric is a switch architecture that allows the data plane to be programmed with eBPF instructions by using a modified version of *uBPF*.

Since eBPF programs can be dynamically modified, it allows the parsing of arbitrary protocols and the use of eBPF maps to store states. New helper functions can be developed to add support to new features and to provide services such as telemetry, statistics collection, and packet tracking.

The control plane hosts an agent that communicates with the data plane through the Southbound API. The agent is responsible for (i) changing the behavior of the switch, (ii) receiving packets, (iii) reporting events, and (iv) reading and updating table entries. When the agent receives the compiled code from the controller, it uses the eBPF ELF Loader to modify the switch pipeline. The eBPF Loader calls the verifier to check the code. On success, it must perform the allocation of the required eBPF tables and convert the received byte code into a switch-specific format.

When receiving a packet, the eBPF processor executes the previously loaded programmed eBPF instructions onto the packet. At the end of the pipeline, it returns a routing decision to drop the packet, sent it to the controller, forward it to some output port, or flood it.

7.2 Hardware

7.2.1 ***Netronome SmartNICs.*** Smart network interface cards (SmartNICs) are network devices capable of having their functionality modified on runtime to implement different modes of operation.

Instead of just providing connectivity and basic Layer-2 and physical layer processing, these devices can execute user-defined packet processing making use of dedicated cores and memory, freeing many cycles from the computer's CPU. To the best of our knowledge, the only company currently offering this kind of product (with support for eBPF) in the market is Netronome.

Network administrators can define packet processing routines in P4 or eBPF, for example, and load these programs to Netronome cards, which will execute the code provided upon packet reception and transmission. Different languages are supported by different firmware versions, which can be loaded to the SmartNIC without a reboot or removal of the device from the server.

With the corresponding firmware version, eBPF programs running on the TC and XDP layers can be seamlessly offloaded to these devices for increased performance. Most drivers and code necessary to perform the offloading are already part of the upstream kernel.

Given their great processing power, SmartNICs have become a good alternative for low-latency and high speed workloads. Since some functionality can be implemented directly in the hardware, packets can be modified and sent back to the network without even having to go up the operating system's network stack. Moreover, programs can be modified and updated on-the-fly. Typical use cases for such devices are early packet filtering, rate limiters, DDoS mitigation tasks, load balancing, RSS jobs, packet switching, etc.

8 PROJECTS WITH EBPF

Although fairly recent, eBPF has already been used by many groups to power interest research- and industry-led projects. These range from tasks to support production systems operations to techniques to provide new network services. This section discusses some of these to demonstrate the wide spectrum of possible applications to eBPF.

The recent demand for greater network programmability has led to the emergence of technologies such as segment routing [29]. This technique allows network administrators to specify different actions to be performed on packets at specific points in the network. Using MPLS labels or the IPv6 protocol with a special field called Source Routing Header (SRH), each packet is encapsulated with an ordered list of routing and processing actions, called segments. As the packet is moved over the network, enabled devices process the list of segments and perform the actions specified by it. This allows the implementation, for example, of different network functions [4].

The Linux kernel already supports segment routing over IPv6 since version 4.10, but with only a few processing options, such as routing and sending and receiving labeled packets. [24] used the eBPF framework to make the creation and specification of new segments more generic and flexible. Through the addition of new helper functions, the authors extended the Linux kernel to allow the implementation of segments in the form of eBPF programs. Thus, new network functions can be easily developed and associated with Linux routing rules, making it easier to integrate with segment-based routing over IPv6 environments.

Several other projects that make use of the eBPF framework to add programmability to the data plane in different contexts. InKeV [5] is a network virtualization platform that uses eBPF programs to modify the data path of virtual data center networks. To solve the OpenFlow fixed matching problem, [33] propose to utilize eBPF programs to provide varied matching fields. [64] use eBPF to create an extensible datapath architecture to the Open vSwitch virtual switch. [12] propose a new version of the *iptables* tool using this technology.

In IoT scenarios and the edge computing paradigm, the data collected from sensors can be requested by several entities. In this case, the information is duplicated. In [6], an eBPF program controls the packet duplication operation.

Some companies already use eBPF in production environments, such as Cloudflare, which uses XDP for denial-of-service attack mitigation, load balancing, and eBPF on upper layers for socket filtering and dispatching [45]. Another example comes from Facebook, which developed an L4 load balancer based on eBPF, called Katran [26]. Also, Netflix has been using eBPF for performance monitoring and system profiling [36].

Moreover, eBPF technology has been playing an important role in container networking. The `veth` interface type, which is commonly used on Linux for communication between containers received support for Native XDP on kernel 4.14 [41]. The Cilium [20] open-source project uses eBPF extensively to provide networking and security for microservice applications, being aware of higher-level details than just network headers. By loading eBPF programs to containers, Cilium can apply per-container security and networking policies. Weave Scope is another project that has been leveraging eBPF for a while to track TCP connections on Kubernetes clusters [69]. Project Calico has also announced earlier this year that a new data plane for container networking based on eBPF is being developed [57].

Finally, eBPF-based open source projects have also emerged in recent years. Cilium [18] is a project to provide security in container networks and microservice applications. The IOVisor [32] project maintains multiple eBPF-based subprojects such as *gobpf* [30], which allows interaction with the eBPF system using the Go language, *ply* [56] and *bpftrace* [16] for kernel introspection, in addition to BCC [7] and *uBPF* [65], discussed previously.

9 LIMITATIONS AND WORKAROUNDS

eBPF is a powerful technology for fast packet processing and kernel programmability. However, to execute inside the kernel, some restrictions are applied to eBPF programs to guarantee system stability and security. This section discusses some eBPF limitations and workarounds to overcome them. Some of the workarounds described here can be found on the XDP project development repository available at [3].

9.1 Subset of C language libraries

eBPF uses a restricted number of C language libraries and does not support operations with external libraries. An alternative to overcome this limitation is to define and use auxiliary functions. For example, eBPF programs can not use the `printf` function, because they run inside the kernel and this function is not implemented in the eBPF. However, they can use the `bpf_trace_printk()` helper function, which saves log messages generated by eBPF programs according to user-defined output in the kernel trace folder (`/sys/kernel/debug/tracing/trace`). By using the log generated, the user can analyze and find possible errors in the execution of the eBPF program.

9.2 Non-static global variables

Currently, eBPF programs only support static global variables [13]. An alternative is to use the `BPF_MAP_TYPE_PERCPU_ARRAY` map. This map reserves a user-defined size non-shared memory space that can be used to store temporary data with a single entry during program execution. [19].

9.3 Loops

The eBPF verifier did not allow loops to make sure all programs finish. The first technique adopted to bypass this limitation was to use loop unrolling directives from the `clang` compiler to rewrite a loop as a repeated sequence of independent instructions. This technique partially solves this limitation, as it can only be used when the number of repetitions can be determined at compile time. Besides that, it has a side effect of increasing the number of instructions in the final program. The code snippet below demonstrates how to tell the compiler to unroll a `for` loop:

```
#pragma clang loop unroll (full)
for (int i=0; i<8; i++) { ... }
```

9.3.1 *Bounded loops.*
In kernel version 5.3 the use of unroll loop directives was replaced by bounded loops. Since this release, the verifier does not reject all eBPF programs that contain loops without first checking if loops finish within a timeout. Bounded loops were proposed by John Fastabend and presented during the BPF Microconference at the 2018 Linux Plumbers Conference. This technique allows using simple loops modeled through the verifier. It analyzes the behavior of a loop through its induction variable and checks if memory accesses using the induction variable belong to the range of memory addresses. Implementation details about bounded loops on the verifier are available at [27].

9.4 Limited stack space

Local variables of a C program are stored in the stack after its translation to an eBPF program. As the stack space is limited to only 512 bytes, it may be insufficient to store all the local variables of a program after its translation. The workaround adopted is the same used for global variables: to use the `BPF_MAP_TYPE_PERCPU_ARRAY` as an auxiliary buffer to store some local variables when the stack space is not enough [19].

9.5 Complex applications

Miano et al. [47] provide an in-depth discussion about the challenges faced when implementing complex network functions with eBPF. For example, applications that need to send the same packet to multiple interfaces (e.g., flood operation on a switch) are hard to implement, since the program would have to loop through all interfaces and copy the packet to each one of them. These are difficult tasks given the current available eBPF mechanisms.

In addition, since eBPF programs are associated with hooks, they follow a passive event-based model. This makes it difficult to perform asynchronous tasks, such as active network measurements, which (currently) would have to be performed by a userspace application, for example.

Moreover, the existing tools to implement a control plane for eBPF programs (e.g., `libbpf` and the `bpf` *syscall*) are very basic, mainly relying on map structures for data exchange. However, projects like Cilium and BCC represent progress in that respect.

Finally, the maximum number of instructions of eBPF programs used to be limited to 4096, and this difficulted the development of complex network functions. One way to get around this limitation was to split a program into multiple subprograms and jump from one subprogram to another using the `bpf_tail_call()` helper function. This technique enables the development of network services as a collection of loosely coupled modules, where each module performs a different function (analysis, classification, or field modification), with low overhead when jumping from one module to another. However, the maximum number of nested tail calls allowed is 32. In April 2019, the maximum number of instructions was increased from 4096 to 1 million, allowing the execution of larger eBPF programs without requiring tail calls [1].

10 COMPARISON WITH SIMILAR TECHNOLOGIES

As shown throughout this work, eBPF is a powerful tool to add programmability to networking platforms and an alternative to achieve fast packet processing on Linux environments through the XDP hook. In this section, we discuss how eBPF fits into programmable and high-speed data planes landscapes by comparing it with some widely used technologies: P4, Click, Netmap, and DPDK.

10.1 Programmable dataplanes

The P4 language [14] has been proposed to enable the definition of custom data planes for programmable switches. With widespread adoption, P4-enabled devices could compose a fully programmable core network, defining the functionality of both physical and virtual switches. On the same front, BPFabric uses eBPF as its language to define the behavior of a programmable virtual switch, applicable to virtualized environments or even server-centric networks.

However, the main strength of eBPF resides in its application to the edges of communication. In the Linux kernel, it is capable of defining data plane functionalities at the communication endpoints, a different use case than the one P4 is designed for. Switching and routing are done by the core network, covered by P4, but a significant portion of the network protocol stack is implemented at the endpoints. With eBPF, this remaining part can be monitored, modified and reconfigured on demand, making it an essential tool to provide full network programmability.

Click [37] also enables the creation of custom network elements to process packets inside the Linux kernel. Applications can be created through the composition of many components, called elements, which could also be implemented as C++ code using Click-specific function calls. The definition of a Click application is done through a configuration file specifying the kinds of elements used and the interconnection between them, representing a packet processing graph. This specification can then be compiled to userspace or kernel. Click kernel modules capture packets close to the network device, and can pass them to the kernel stack using *ToHost* elements, similar to what can be done in the XDP hook with eBPF. Li et al. [38] build upon Click to create the ClickNP framework that enables the creation and execution of network functions on FPGAs, achieving 40 Gbps throughput for any packet size.

Although Click and ClickNP provide a broader set of pre-built primitives to create routers and network functions, they do not offer the same level of integration with the kernel stack as eBPF. The latter allows attaching programs to interact with several layers of the kernel stack, providing a higher degree of control over the kernel's packet processing mechanisms. However, Click could be modified to take advantage of the eBPF/XDP infrastructure to be its basis to interact with packet processing facilities, and possibly combine its expressiveness with eBPF's performance and native kernel integration.

10.2 High-speed packet processing

On high-speed networks with 10 Gbps links and beyond, inter-packet arrival times can get as low as tens of nanoseconds, leaving very little time to process each packet. Due to this stringent requirement, common system operations become too costly, such as context switches and interrupt handling. Well-known technologies like DPDK (Data Plane Development Kit) [42] and NetMap [58] handle this problem by operating in poll mode and bypassing the kernel altogether, performing all packet processing in user space. By the use of specialized drivers, packets received by the NIC are sent directly to a user space application, which will process them. In addition, the DPDK library and applications built with it are usually optimized for aligned memory access, multi-core processing, non-uniform memory access (NUMA) and other optimizations aimed to save precious CPU cycles.

The eBPF system in the kernel provides good performance in a different manner: by allowing custom packet processing at the XDP hook, which is the lowest kernel layer. Through this hook, eBPF applications can parse, modify, collect statistics, and take action on incoming packets possibly without going through the OS' network stack, forwarding packets directly back to the network. This way, context switches are avoided by embedding all network processing in the kernel.

If compared to DPDK, one of the advantages of the XDP hook is being able to use existing network facilities present in the kernel (ex: routing tables), which have to be re-implemented from the ground up when processing is done in user space, which is the case of DPDK. Also, since XDP is handled by the kernel, it can benefit from kernel API stability guarantees, from the separation mechanisms already in place to enforce security, integration with the existing stack without requiring re-injecting packets through an exception path. Device sharing is also facilitated, as the program does not need full-control over the NIC, which can remain visible to the OS. At the same time, programs are transparent to other applications on the host, since processing is hidden under the OS' abstraction layers. In terms of resource usage, it avoids investing precious CPU cycles with busy-polling due to its event-based nature. Recent performance comparisons between the two technologies show that DPDK still reaches higher bandwidths (115 Mpps against 100 Mpps for XDP when dropping packets with five cores), but at the cost of a much higher CPU usage than XDP [31]. Finally, programs can be replaced atomically, providing greater flexibility for on-demand changes to packet processing.

Thus, eBPF and DPDK provide different approaches to high-speed packet processing, both with their own set of restrictions and advantages, as discussed in this section and in § 9. The best choice of technology may depend on the actual use case, although integration of both could be achieved by using AF_XDP sockets and the corresponding poll mode driver on DPDK [22] or the *librte_bpf* library provided by DPDK to add an eBPF VM directly to DPDK applications [23].

11 CONCLUSION

In this work, we presented a vision of the theoretical and practical aspects related to fast packet processing with eBPF and XDP. In the theoretical part, we discussed the BPF and eBPF machines, an overview of the eBPF system provided by the Linux kernel, the available hooks and some results of recent research. In the practical part, we focused on eBPF and the XDP hook, providing examples and showing existing tools.

Given their power for fast packet processing, we consider that there is great potential in the development of new research projects with eBPF and XDP, either as a tool for the development of new network functions, or allowing to provide new functionalities in the data plane, such as the creation of new communication standards and protocols, or in the development of new research prototypes and network solutions. Together, eBPF and XDP will help develop new interesting research with great potential in the area of Computer Networking.

REFERENCES

[1] 2019. bpf: increase complexity limit and maximum program size. https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/commit/?id=c04c0d2b968ac45d6ef020316808ef6c82325a82.

[2] 2019. libbpf unification and golang bindings. Discussion Summary, Linux Kernel Developers' bpfconf 2019. http://vger.kernel.org/bpfconf2019.html#session-4

[3] 2019. XDP project repository. https://github.com/xdp-project/xdp-project.

[4] Ahmed Abdelsalam, Francois Clad, Clarence Filsfils, Stefano Salsano, Giuseppe Siracusano, and Luca Veltri. 2017. Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure. In *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017*. IEEE, Bologna, Italy, 1–5. https://doi.org/10.1109/NETSOFT.2017.8004208 arXiv:1702.05157

[5] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. 2018. InKeV: In-kernel Distributed Network Virtualization for DCN. *SIGCOMM Comput. Commun. Rev.* 46, 3, Article 4 (July 2018), 6 pages. https://doi.org/10.1145/3243157.3243161

[6] S. Baidya, Y. Chen, and M. Levorato. 2018. eBPF-based content and computation-aware communication for real-time edge computing. In *INFOCOM - IEEE Conference on Computer Communications Workshops (INFOCOM WORKSHOPS)*. IEEE, Honolulu, HI, USA, 865–870. https://doi.org/10.1109/INFCOMW.2018.8407006

[7] BCC. 2019. BPF Compiler Collection. https://github.com/iovisor/bcc.

[8] BCC. 2019. BPF program types. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#program-types.

[9] BCC. 2019. XDP compatible drivers. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp.

[10] David Beckett, Jaco Joubert, and Simon Horman. 2018. Host Dataplane Acceleration (HDA).

[11] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Netdev 2.1 - Technical Conference on Linux Networking*. 1–5.

[12] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. 2018. Accelerating Linux Security with eBPF Iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos (SIGCOMM '18)*. ACM, New York, NY, USA, 108–110. https://doi.org/10.1145/3234200.3234228

[13] Daniel Borkmann. 2019. bpf, libbpf: support global data/bss/rodata sections. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d859900c4c56.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[15] Autores bpftool. 2018. Manual Bpftool. https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/bpf/bpftool/Documentation/bpftool.rst.

[16] bpftrace. 2019. High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace.

[17] Mihai Budiu. 2015. Compiling p4 to ebpf. https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4

[18] Cilium. 2018. Cilium 1.0: Bringing the BPF Revolution to Kubernetes Networking and Security. https://cilium.io/blog/2018/04/24/cilium-10/.

[19] Cilium. 2019. BPF and XDP Reference Guide. https://cilium.readthedocs.io/en/latest/bpf/. Accessed on 09/09/2019.

[20] Cilium. 2019. Cilium: API-aware Networking and Security. https://cilium.io/. Accessed on 09/09/2019.

[21] Jonathan Corbet. 2014. BPF: the universal in-kernel virtual machine. https://lwn.net/Articles/599755/.

[22] DPDK. 2019. AF_XDP Poll Mode Driver . https://doc.dpdk.org/guides/nics/af_xdp.html.

[23] DPDK. 2019. Berkeley Packet Filter Library . https://doc.dpdk.org/guides/prog_guide/bpf_lib.html.

[24] Fabien Duchene, Mathieu Jadin, and Olivier Bonaventure. 2018. Exploring Various Use Cases for IPv6 Segment Routing. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos (SIGCOMM '18)*. ACM, New York, NY, USA, 129–131. https://doi.org/10.1145/3234200.3234213

[25] Eric Dumazet. 2011. A JIT for packet filters. https://lwn.net/Articles/437981/.

[26] Facebook. 2018. Katran source code repository. https://github.com/facebookincubator/katran.

[27] John Fastabend. 2018. [RFC PATCH 00/16] bpf, bounded loop support work in progress. https://lwn.net/ml/netdev/20180601092646.15353.28269.stgit@john-Precision-Tower-5810/.

[28] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review* 44, 2 (2014), 87–98.

[29] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. 2018. *Segment Routing Architecture*. RFC 8402. RFC Editor.

[30] gobpf. 2019. Go bindings for creating BPF programs. https://github.com/iovisor/gobpf.

[31] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. 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 (CoNEXT '18)*. ACM, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[32] IOvisor. 2019. Iovisor Project. www.iovisor.org. Accessed on 29/03/2019.

[33] S. Jouet, R. Cziva, and D. P. Pezaros. 2015. Arbitrary packet matching in OpenFlow. In *16th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, Budapeste, Hungary, 1–6. https://doi.org/10.1109/HPSR.2015.7483106

[34] Simon Jouet and Dimitrios P. Pezaros. 2017. BPFabric: Data Plane Programmability for Software Defined Networks. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems (ANCS '17)*. IEEE Press, Piscataway, NJ, USA, 38–48. https://doi.org/10.1109/ANCS.2017.14

[35] Michael Kerrisk. 2013. BFPC 8 Linux Manual Page. http://man7.org/linux/man-pages/man8/bpfc.8.html. Accessed on 08/06/2019.

[36] Jason Koch, Martin Spier, Brendan Gregg, and Ed Hunter. 2019. Extending Vector with eBPF to inspect host and container performance. https://medium.com/netflix-techblog/extending-vector-with-ebpf-to-inspect-host-and-container-performance-5da3af4c584b

[37] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[38] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/2934872.2934897

[39] libbpf. 2018. libbpf source code. https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/lib/bpf.

[40] libbpf. 2019. stand-alone libbpf. https://github.com/libbpf/libbpf.

[41] Linux. 2017. net: xdp: support xdp generic on virtual devices. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d445516966dcb2924741b13b27738b54df2af01a.

[42] Linux Foundation. 2015. Data Plane Development Kit (DPDK). http://www.dpdk.org

[43] D. F. Macedo, D. Guedes, L. F. M. Vieira, M. A. M. Vieira, and M. Nogueira. 2015. Programmable Networks: From Software-Defined Radio to Software-Defined Networking. *IEEE Communications Surveys Tutorials* 17, 2 (2015), 1102–1125. https://doi.org/10.1109/COMST.2015.2402617

[44] Alan Maguire. 2019. Notes on BPF (1) - A Tour of Program Types. https://blogs.oracle.com/linux/notes-on-bpf-1.

[45] Marek Majkowski. 2019. Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/

[46] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. USENIX Association, Berkeley, CA, USA, 1–11. http://dl.acm.org/citation.cfm?id=1267303.1267305

[47] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating complex network service with ebpf: Experience and lessons learned. In *High Performance Switching and Routing (HPSR)*. IEEE, Budapeste, Hungary, 1–8.

[48] Rashid Mijumbi, Joan Serrat, Juan Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. 2016. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys and Tutorials* 18, 1 (2016), 236–262. arXiv:1509.07675 http://ieeexplore.ieee.org/document/7243304/

[49] David Miller. 2017. BPF Verifier Overview. https://lwn.net/Articles/794934/. Accessed on 09/04/2019.

[50] J. Mogul, R. Rashid, and M. Accetta. 1987. The Packer Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. ACM, New York, NY, USA, 39–51. https://doi.org/10.1145/41457.37505

[51] Quentin Monnet. 2019. All-Out Programmability in Linux: An Introduction to BPF as a Monitoring Tool. https://qmo.fr/docs/talk_20190516_allout_programmability_bpf.pdf. Accessed on 09/04/2019.

[52] Quentin Monnet. 2019. Rust virtual machine and JIT compiler for eBPF programs. https://github.com/qmonnet/rbpf.

[53] Netronome. 2019. Sample BPF offload apps. https://github.com/Netronome/bpf-samples.

[54] PLUMgrid. 2016. Linux kernel Source Code. https://github.com/torvalds/linux/blob/v5.3/samples/bpf/xdp1_kern.c. Accessed on 07/06/2019.

[55] PLUMgrid. 2016. Linux kernel Source Code. https://github.com/torvalds/linux/blob/v5.3/samples/bpf/xdp1_user.c. Accessed on 07/06/2019.

[56] ply. 2019. Dynamic Tracing in Linux. https://github.com/iovisor/ply.

[57] Alex Pollitt. 2019. Tigera adds eBPF support to Calico . https://www.projectcalico.org/tigera-adds-ebpf-support-to-calico/. Accessed on 09/09/2019.

[58] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 101–112. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo

[59] Marta Rybczyńska. 2019. Bounded loops in BPF for the 5.3 kernel. https://www.spinics.net/lists/xdp-newbies/msg00185.html. Accessed on 09/04/2019.

[60] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. 2019. Linux Socket Filtering aka Berkeley Packet Filter (BPF). www.kernel.org/doc/Documentation/networking/filter.txt. Accessed on 03/17/2019.

[61] Haoyu Song. 2013. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN '13)*. ACM, New York, NY, USA, 127–132. https://doi.org/10.1145/2491185.2491190

[62] Alexei Starovoitov. 2015. bpf: introduce bpf_tail_call() helper. https://lwn.net/Articles/645169/.

[63] Alexei Starovoitov. 2018. Lifetime of BPF objects. https://facebookmicrosites.github.io/bpf/blog/2018/08/31/object-lifetime.html

[64] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. 2017. Building an Extensible Open vSwitch Datapath. *SIGOPS Oper. Syst. Rev.* 51, 1 (Sept. 2017), 72–77. https://doi.org/10.1145/3139645.3139657

[65] uBPF. 2019. Userspace eBPF VM. https://github.com/iovisor/ubpf.

[66] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2019. eBPF Tutorial. https://github.com/racyusdelanoo/bpf-tutorial.

[67] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2019. Zenodo - eBPF Tutorial. https://zenodo.org/record/3519347#.XbMxR6zMNhE.

[68] VMWare. 2018. p4c-xdp. https://github.com/vmware/p4c-xdp.

[69] WeaveWorks. 2017. Improving performance and reliability in Weave Scope with eBPF . https://www.weave.works/blog/improving-performance-reliability-weave-scope-ebpf/.

[70] XDP-Project. 2019. AXDP Hands-On Tutorial. https://github.com/xdp-project/xdp-tutorial Accessed on 08/30/2019.