POLITECNICO DI TORINO
Repository ISTITUZIONALE

Creating Complex Network Services with eBPF: Experience and Lessons Learned

Original
Creating Complex Network Services with eBPF: Experience and Lessons Learned / Miano, Sebastiano; Bertrone, Matteo; Risso, FULVIO GIOVANNI OTTAVIO; Tumolo, Massimo; VASQUEZ BERNAL, Mauricio. - STAMPA. - (2018). (Intervento presentato al convegno IEEE International Conference on High Performance Switching and Routing (HPSR 2018) tenutosi a Bucarest, Romania nel June 2018).

Availability:
This version is available at: 11583/2712562 since: 2018-09-11T00:19:40Z

Publisher:
IEEE

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

12 October 2024

Creating Complex Network Services with eBPF: Experience and Lessons Learned

Sebastiano Miano*, Matteo Bertrone*, Fulvio Risso*, Massimo Tumolo*, Mauricio Vásquez Bernal*
*Politecnico di Torino, Department of Computer and Control Engineering, Torino, 10129, Italy

***Abstract*—The extended Berkeley Packet Filter (eBPF) is a recent technology available in the Linux kernel that enables flexible data processing. However, so far the eBPF was mainly used for monitoring tasks such as memory, CPU, page faults, traffic, and more, with a few examples of traditional network services, e.g., that modify the data in transit. In fact, the creation of complex network functions that go beyond simple proof-of-concept data plane applications has proven to be challenging due to the several limitations of this technology, but at the same time very promising due to some characteristics (e.g., dynamic recompilation of the source code) that are not available elsewhere. Based on our experience, this paper presents the most promising characteristics of this technology and the main encountered limitations, and we envision some solutions that can mitigate the latter. We also summarize the most important lessons learned while exploiting eBPF to create complex network functions and, finally, we provide a quantitative characterization of the most significant aspects of this technology.**

***Index Terms*—eBPF, XDP, Linux, network functions, NFV, dataplane.**

I. Introduction

The extended Berkeley Packet Filter (eBPF) is a recent technology that enables flexible data processing thanks to the capability to inject new code in the Linux kernel at run-time, which is fired each time a given event occurs, e.g., a packet is received. While its ancestor, the Berkeley Packet Filter (BPF) was used mainly to create packet filtering programs, eBPF has been successfully used primarily in monitoring tasks. Surprisingly, its usage in traditional network applications, such as data plane services, has been less intense.

In fact, the creation of complex network functions that go beyond simple proof-of-concept data plane applications has proven to be challenging, due to the several limitations of this technology, although it is evolving fast, as shown by the significant number of patches and new features added almost daily in the Linux kernel. In addition, eBPF is not (yet) backed by a rich ecosystem of tools and libraries aimed at simplifying the life of potential developers; the BPF Compiler Collection (`bcc`) [1] is more oriented to tracing than packet manipulation.

However, eBPF is very promising due to some characteristics that can hardly be found all together, such as the capability to execute code directly in the vanilla Linux kernel, hence without the necessity to install any additional kernel module; the possibility to compile and inject dynamically the code; the capability to support arbitrary service chains; the integration with the Linux eXpress Data Path (XDP) for early (and efficient) access to incoming network packets. At the same time, eBPF is known for some limitations such as limited program size, limited support for loops, and more, which may impair its capacity to create powerful networking programs.

This paper presents our experience in developing complex network services with eBPF and shows the most promising characteristics of this technology as well as the main encountered limitations, as they appear in everyday life of a typical developer. This paper will discuss the actual importance of the above limitations with respect to the necessity to create complex network applications and the possible solutions (if any). Finally, it will also discuss some of the peculiar advantages of this platform, backed by experimental evidence taken from our services. In the end, this paper ponders advantages and limits of the eBPF technology, analyzing its suitability as a platform for the development of future complex network services targeting mainly virtualized environments.

This paper is structured as follows. Section II presents a high-level overview of the eBPF. Section III represents the central part of the paper, highlighting our experience when coping with different aspects of eBPF, and the main lessons learned. Finally, Section IV provides the necessary evidence to the previous findings and Section V concludes the paper.

II. Background

The Berkeley Packet Filter (BPF) is an in-kernel virtual machine for packet filtering that has been deeply revisited starting from 2013 and is now known as extended BPF (eBPF). In addition to several architectural improvements, eBPF introduces the capability of handling generic event processing in kernel, JIT compiling for increased performance, stateful processing using maps, and libraries (helpers) to handle more complex tasks, available within the kernel.

eBPF allows an user-space application to inject code in the kernel at runtime, i.e., without recompiling the kernel or installing any optional kernel module. eBPF programs can be either written using eBPF assembly instructions and converted to bytecode using `bpf_asm` utility, or in restricted C and compiled using the LLVM Clang compiler. The bytecode can then be loaded using the `bpf()` system call. For this process to succeed, the program has to get through a sanity-check from the eBPF verifier, that walks the control flow graph to ensure termination, simulates the execution to check that memory and registers are always in a valid state, and verifies that the calls to helper functions are allowed.

A loaded eBPF program follows an event-driven architecture and it is therefore hooked to a particular type of event: each occurrence of the event will trigger its execution, and

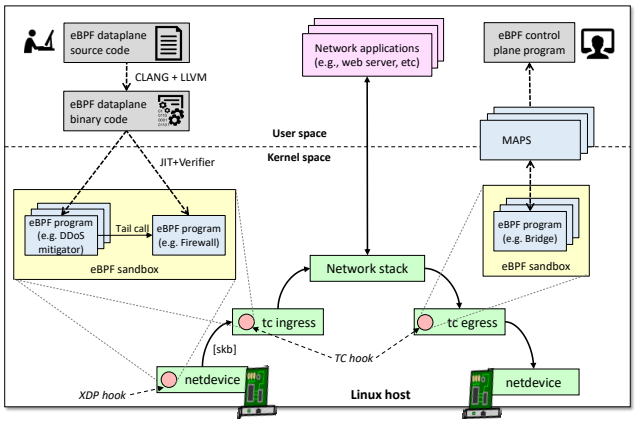

Fig. 1. eBPF overview.

based on the type of the event the program might be able to alter the event context. Furthermore, programs are stateless by their nature, as each run is independent from the others. For this reason, the eBPF provides *maps*, data structures accessible using helper functions, needed to share information between (*i*) different runs of the same program, (*ii*) different programs or (*iii*) a program and the userspace.

For networking purposes, program execution is triggered by the arrival of a packet. Two hooks are available to intercept packets and possibly mangle, forward or drop them: eXpress Data Path (XDP) and Traffic Control (TC). XDP programs intercept RX packets right out of the NIC driver, possibly before the allocation of the Linux socket buffer (`skb`), allowing e.g., early packet drop. TC programs intercept data when it reaches the kernel traffic control function, either in RX or TX mode.

Multiple eBPF programs can be instantiated at the same time, even attached to different hooks. Furthermore, eBPF programs can either operate in isolation (returning the packet to the hook they are attached to) or be chained, e.g., to create a more complex service, using a low-overhead linking primitive called *tail call*. Tail calls are a sort of long jump from one program to another; differently from *function calls*, this primitive does not permit to return to the previous context.

A high-level view of the eBPF architecture, including both code injection and run-time processing, is depicted in Figure 1.

III. Experiences and Insights

This section presents the main challenges encountered while implementing complex network functions with eBPF, together with different insights we adopted (or could be adopted) to accomplish this task.

A. eBPF limitations

eBPF suffers from some well-known limitations due to its restricted virtual machine, which are needed to guarantee the integrity of the system. This Section discusses the impact of the above limitations and highlights some other, less known issues that arise when creating network services.

1) Limited program size: eBPF programs are executed within the kernel context; for this reason, their size is limited to a maximum of 4096 assembly instructions to guarantee that any program will terminate within a bounded amount of time. This restriction may be limiting when creating network functions that perform complex actions in the data plane, considering that the BPF assembly instructions generated after the code compilation could be significantly higher than the number of lines of code of the C source file.

Learning 1: This limitation can be circumvented by partitioning the network function into multiple eBPF programs and jumping from one to another through *tail calls*. This technique enables the creation of network services as a collection of loosely coupled modules, each one implementing a different function (e.g., packet parsing, classification, fields modification), with a very low overhead in jumping from a piece to another, as shown in Section IV-E. This attractive feature also comes with an upper bound limit of 32 nested jumps, which we found being more than enough to implement complex services.

2) Unbounded loops: Since eBPF applications can be loaded at runtime in the kernel, they are checked through an in-kernel verifier that ensures programs cannot harm the system. The verifier looks for multiple possible threats (e.g., state of BPF registers, type of values, etc.), rejecting the code in case backward jumps are detected, thus ensuring that all programs will terminate, but forbidding any loop in the code. We list some cases where this limitation may be a problem.

- **Parsing nested headers**: this is the case of IPv6, which requires to loop through all the *extensions headers* to find the last header indicating the type of the upper-layer protocol in the payload of the packet. A similar issue affects MPLS and VLAN headers, whose number of instances is not known a priori. Creating a network function that performs these actions in eBPF is not possible unless we introduce additional constraints, as described below.
- **Arbitrary processing of packet payloads**: this is required for example to check the presence of a signature in the payload. While there are cases in which this loop can be avoided thanks to the availability of specific helpers that perform the entire job (e.g., to recalculate the L3/L4 checksum), in general the necessity to perform a loop scanning the entire packet cannot be excluded a priori.
- **Linear scanning of data structures**: algorithms that require a linear scan of data structures (e.g., a map) may need to be adapted for the eBPF environment. A possible example is a firewall that looks for the rule that matches a given packet, which is usually performed through a linear scan across all the active policies.

Learning 2: Although the eBPF assembly does not support backward jumps, as far as *bounded* loop are concerned, we can exploit the `pragma unroll` directive of the LLVM compiler to rewrite the loop as a repeated sequence of similar independent statements. This can be achieved by imposing a constant upper bound limit to the loop, such as the maximum

number of IPv6 headers, nested MPLS labels, packet size[1]. This solution presents two limitations: (*i*) the size of the program increases, with the possible consequences (and solutions) shown in III-A1; (*ii*) we may not be able to guarantee that all cases are handled, e.g., in case of an exceptional number of IPv6 headers is present. However, even if the lack of the support for unbounded loops seems to be an important limitation, we found it not so critical in our programs, as it can be often circumvented by creating bounded loops, although this is left to the responsibility (and experience) of the developer.

3) Send the same packet on multiple ports: This is still a rather common operation even in modern local area networks, e.g., to handle broadcast frames (e.g., ARP requests), multicast, or flooding (e.g., in an 802.1D bridge). However, at least three issues can be encountered when implementing this feature. First, we may need to loop through all the interfaces to forward the packet the desired number of times; this can be implemented only if we are able to set an upper limit on the loop and unroll it (as discussed in Section III-A2). Second, the packet must be cloned before sending it on an additional interface; this can be done with the `bpf_skb_clone_redirect()` helper, which simultaneously duplicates and *forwards* the original packet to a target interface. However, this helper is available only when the program is attached to the TC hook, while an equivalent helper is not available for XDP programs[2]. Third, if the service is part of a virtual chain composed by multiple VNFs connected through tail calls such as in [3], the aforementioned approach fails. In fact, the redirect function will be followed by a tail call, which never returns the control to the caller, hence preventing the caller code to send a packet to multiple ports.

4) Packet-driven processing: While the execution of an eBPF program is triggered by an event, the only event that is supported by TC/XDP programs is a frame traversing the selected kernel hook. This prevents the eBPF data plane to react to other events such a timeout that signals the necessity to periodically send a packet (e.g., neighbor greetings in routing protocols), or to refresh an expired entry in a table. The above events have to be handled elsewhere, such as in the slow path (Learning 3) or in the control plane (Section III-A6).

5) Putting packets on hold: In some cases, network functions may need to put the current frame on hold while waiting for another event to happen. This is the case of a router that holds a packet while waiting for an answer to its own ARP request aiming at discovering the MAC address of the next hop; the original packet should be released after receiving the ARP reply. Unfortunately, eBPF does not have a "steal" action such as in Netfilter, hence preventing this technology to take the ownership of the packet. Possible workarounds to this problem can be envisioned, such as copying the entire packet in a temporary memory, but they may not be suited for all cases (e.g., handling retransmissions, as the packet has to be released when a timeout occurs).

Learning 3: Taken together, limitations A3-5 suggest the necessity to introduce a novel data plane component that is no longer limited by the eBPF virtual machine, which executes arbitrary code that can cope with the cases in which the current eBPF technology cannot be used. This brings to the evidence the necessity of a *slow path* module, executed in userspace, that receives packets from the eBPF program and reacts consequently with arbitrary processing defined by the developer; for example by modifying the packet and sending it back in the egress queue of a specific netdevice[3]. The necessity of this module is also highlighted in [4] where the authors use the OvS userspace module to process packets that do not match a flow in the OvS kernel eBPF data path.

6) No support for complex control planes: So far, eBPF has been used mostly for tracing applications, which feature a very simple control plane such as reading data from maps. As a consequence, existing eBPF software frameworks provide a nice set of abstractions that help developers to create data plane code (hook handling, maps, etc), while it is rather primitive with respect to the control plane, enabling userspace programs mainly to read/write data from maps. Networking services are rather different and often require a sophisticated control plane not just to read/write data from maps, but to create/handle special packets (e.g., routing protocols), to cope with special processing that may complicate (and slow down) the data plane if handled here (e.g., ARP handling; Section III-A5), or to react to special events such as timeouts (Section III-A4).

Learning 4: This results in non negligible difficulties when implementing the (complex) control plane of a service, as it forces developers to dedicate a considerable amount of time to write the code that handles common control plane operations from scratch, without any help from existing software frameworks.

B. Enabling more aggressive service optimization

The traditional approach when implementing a network function is to *(i)* create a program that contains all possible use cases and control flows (branches) and *(ii)* make it completely agnostic with respect to its actual configuration, which is pushed in the data plane afterwards. With eBPF, this approach is no longer the only option; in fact, programs can be compiled from their C source code and injected in the kernel at runtime, with the system already up and running. This allows us to take advantage of the runtime service conditions (e.g., traffic pattern, service configuration, interface from which the traffic is received/sent) to empower more aggressive optimizations compared to traditional programs. This section presents three techniques of this type, enabled by the use of eBPF as data plane for our networking services.

1) Moving configuration data from memory to code: A conventional approach for configuring a network function is

[1]A patch [2] that adds support for bounded loops without the necessity to use the `pragma unroll` directive has been recently proposed and it may be integrated in future kernel versions.

[2]For the sake of precision, XDP offers only the `bpf_redirect_map()` helper, which sends the packet to a port but does not clone it.

[3]In Linux, a *netdevice* is a physical or virtual network interface card.

to save data (e.g., the public/private ports in a NAT, the set of rules in a firewall) in memory, which will be accessed by the run-time code each time a packet is received. In the eBPF domain, this corresponds to saving data in *maps*, which provide a bidirectional userspace-kernel communication channel. While this approach is the only viable option for other technologies, eBPF enables the loading of new code dynamically, hence allowing the creation of situational-specific code that also embeds the data needed for the current processing. This technique leverages the superior processing capabilities of modern CPUs (e.g., speculative execution), trading more processing instructions for fewer memory accesses, which are known to introduce a noticeable penalty in particular when random data access patterns are required, which lead to cache ineffectiveness.

Learning 5: Hardcoding parameters in the eBPF code in a way that the service can directly use them without any explicit memory access may lead to significant performance gains (Section IV-C2). However, this requires to handle configuration changes by *dynamically reloading* the program with the updated parameters; more details will be presented in Section III-B3.

2) Code tailoring: A network function can have a different set of features that are not always needed at runtime. For example, our bridge supports both VLANs and Spanning Tree, but they may not be required (hence be turned off) at a given time. The amount of code needed to handle these features is not negligible and can impact the forwarding performance of the eBPF network function.

Learning 6: Our experiments showed that cutting the superfluous code, at runtime, will bring a significant reduction in the number of control flows and branches of the program, hence simplifying the new code and improving the overall performance of the service, as shown in section IV-C1.

3) Dynamic reloading: The previous two techniques can provide substantial performance gains; however, their value would be impacted without the possibility to *reload* the program *at runtime* with a more appropriate version, while maintaining at the same time the state (e.g., maps) and configuration of the old program. Dynamic code reloading is currently supported in eBPF, but the existing software frameworks do not offer any help, leaving this responsibility in the developer's hands and hence requiring additional complexity when writing efficient network services. Our prototypical code that supports this feature is strongly hinged on reducing the service disruption and packet loss (see Section IV-C). While the new service is compiled and injected, the old one still handles the traffic. When the new program is ready, maps of the old instance are attached to it and then atomically *swapped* by substituting the pointer to the old program with the new one. At this point, the new program will start processing the traffic, and the old one is unloaded.

C. Data structures

eBPF does not have the concept of "raw" memory as used by classical computers; data are in fact stored in memory areas structured according to a predefined access model (e.g., hash map, lru map, array). As of this writing, there are seventeen types of map that can be used by an eBPF program. Even though the existing set of maps is very large and allows to fulfill the requirements of the majority of applications, in the next two subsections we present some cases in which it may not be enough.

1) Stack map: We may envision a service that needs to maintain a pool of elements that can be consumed (e.g., through a *pop* action to get the first free element of the pool) or produced (e.g., a *push* operation to insert an item back in the pool) atomically, hence similar to the behavior of a *stack*; this is the case of a NAT service, which needs to keep the list of available TCP/UDP ports. Unfortunately, this type of data structure is not present among the set of maps available in eBPF. Although its behavior can be emulated using an array and a global counter, used as the index of the first element to retrieve, it is subject to concurrency problems when multiple instances of the same program access the same data from different kernel threads, causing race conditions[4].

2) Map with timeout: A typical scenario for networking functions is to have entries in a table with an associated timeout; when an entry is not accessed for a specific time interval, it expires and is removed from the list (e.g., the filtering database of a bridge). Unfortunately, eBPF does not have such a map. This behavior can be emulated by *(i)* inserting an additional field in the entry that corresponds to the current timestamp and *(ii)* check, at every access, that the item has expired; if so, the entry is deleted and the service continues as if the entry was not present. Obviously, entries that are no longer accessed will never be deleted unless an LRU (least recently used) map is used. This approach partly complies the lack of this table in eBPF (indeed, it is the approach used to implement our 802.1D bridge), even if it complicates the data and control plane of the service that must take care of discarding old entries, with possible racing conditions.

3) Concurrent map access: When a hook triggers an eBPF program in the kernel, multiple instances of the same eBPF application can be executed simultaneously on different cores. A normal eBPF map has a single instance across all cores and could be accessed simultaneously by the same eBPF program running on different cores. eBPF maps are native kernel objects that are protected through the kernel Read-Copy-Update (RCU) [6] mechanism, which makes their access thread safe, regardless of whether the interaction occurred from userspace or directly from the eBPF program. The fact that map access is thread-safe does not exclude the presence of data races, given the implicit multi-threading capabilities of eBPF and the impossibility to use locks.

The interaction between the control plane and the data plane is also subject to race conditions since they do not have a standard synchronization mechanism. For example in a bridge, if the cleanup of the filtering database is performed in the

[4]The need for this type of operations in the eBPF maps has been highlighted several times on the IOVisor mailing list [5].

control plane, we could create the following situation: (*i*) the control plane reads an entry from the filtering database and realizes that it is too old, so it must be removed, (*ii*) the data plane receives a packet for that entry and updates the filtering database with the new timestamp, (*iii*) the control plane eliminates the newly inserted entry, producing an unexpected behavior.

Learning 7: We have noticed that map access is thread-safe since these structures are protected by the RCU mechanism. However, race condition can still happen either between control and data plane or from the same eBPF program running on different cores. Unfortunately, we have not yet found a definitive and general solution for all cases. It is, therefore, the developer who has to take care of this problem and find alternative solutions depending on the application logic.

D. High performance processing with XDP

XDP provides a mechanism to run eBPF programs at the lowest level of the Linux networking stack, directly upon receipt of a packet and immediately out of driver receive queues. It has two operating modes; the first one, called *Driver (or Native) mode*, is the primary mode of operation; to load eBPF programs at this level, the driver of the netdevice must support this model. Running network applications in XDP produces significant performance benefits (as shown in Section IV-D) since the application can perform operations on the packet (e.g., redirect, drop or modify) before any allocation of kernel meta-data structures such as the `skb`, spending fewer CPU cycles for processing the packet compared to the conventional stack delivery. The second one, called *Generic (or SKB) mode* allows using XDP within drivers that do not have native support for it, providing a simple way to use and test XDP programs with less dependencies.

In section III-D1 we show the main differences between XDP and the other network hook point, i.e., TC; we will then present the main drawbacks found in the current support of the Linux kernel for both XDP Driver (section III-D2) and XDP Generic mode (section III-D3).

1) Limited helpers: XDP programs are only allowed to call a subset of helpers compared to eBPF services attached to the TC layer. In general, eBPF has a set of base helpers (e.g., map lookup/update, tail calls) available for all types of programs, with some specific helpers for each category of hook; approximately 29 available in TC and only 7 in XDP. We summarize the main differences in the following list:

- *Checksum calculation:* Primitives for checksum computations were not fully available in XDP as they are in TC. It is going to be fixed in the upcoming version 4.16 [7] of the Linux kernel, but this prevents an eBPF program exploiting this feature to be executed on an older kernel. In that case, the solution is to recompute the checksum "by hand", with dedicated code in the XDP program.

- *Push/Pop headers:* XDP does not offer any helper to push and pop a VLAN tag from the packet or to perform tunnel encapsulation or decapsulation. In case this feature is needed, the XDP program can use the more generic `bpf_xdp_adjust_head()` helper, which provides the ability to adjust the starting offset of the packet along with its size, so it is possible to manipulate the packet according to the application logic.

- *Multi-port transmission:* As already highlighted in III-A3, an equivalent of the `bpf_skb_clone_redirect()` helper available in TC is missing in XDP. This does not allow to forward a packet on several ports at the same time, which is required by different network applications (e.g., bridge, router), unless we implement this feature in the *slow path*.

Learning 8: Writing programs with XDP does not have significant differences compared to TC; most of the actions such as direct packet modification, access to maps or the use of tail calls remain identical with the other hook points. However, the limited number of helpers forces the developer to use more generic functions or to implement those functionalities in the slow path, complicating the code and making it less portable, with the consequence that the code must different depending on the kernel hook to which it is attached.

2) XDP Driver mode limitations: Most XDP-enabled drivers today use a specific memory model (e.g., one packet per page) to support XDP on their devices. Among the different actions allowed in XDP, there is the possibility to redirect the packet to another physical interface (`XDP_REDIRECT`). While this action is currently possible within the same driver, in our understanding, it is not possible between interfaces of different drivers. The main problem is the lack of a common layer/API that the drivers can use to allocate and free pages. With this model, when a driver performs a packet transmission, it can communicate the actual sending, back to the Rx driver, which may recycle the page without having to run into costly DMA unmap operations[5]. This lack of generality limits the network applications that can take advantage of the speed gain provided by XDP, that have to entrust the normal stack processing to make the correct forwarding of the packet.

3) Generic XDP limitations: As previously mentioned, XDP generic can be used to run XDP programs even on drivers that do not have native XDP support. Although they are executed right after the `skb` allocation, thus losing the advantages available in the driver mode, it still provides better performance than other hook points such as TC, as shown in section IV-D2. When triggered, XDP generic programs can modify the content of the received packet; however, if the packet data are part of a *cloned skb*, an XDP program cannot be executed on this packet, since cloned skb cannot be modified. This leads to some limitations such as handling TCP traffic; in fact, our network function running on the XDP generic hook will never be able to receive TCP traffic, since most of the packets belonging to TCP sessions are cloned, in order to be later retransmitted, if necessary.

Learning 9: Although writing programs compatible with both XDP and TC is not a significant problem, their use is not interchangeable. Using XDP programs as substitutes of TC

[5]An interesting discussion on this topic is available here [8]. A patch towards that direction is available in [9].

services is not always possible, resulting advantageous only for specific applications. For example, connecting containers with XDP services may not be appropriate since most of its advantages given by the the early stage in which frames are captured would be lost. The XDP hook has been designed to work mainly in ingress, making tricky the modelization of services such as a firewall that would need, for example, to capture packets generated by the host and going outside the network interface; this is instead possible using TC as a hook point in ingress and in egress.

E. Service function chaining

The possibility to connect eBPF programs through *tail calls* in kernel facilitates the combination of network services (e.g., bridge, router, NAT) in a virtual chain, with considerable advantages as shown in [3]. In this way, eBPF services can connect to the external world either *(i)* through a netdevice or *(ii)* through a tail call, directly to another eBPF module. However, this requires the creation of a different source code based on the port the eBPF program is attached to, since the assembly instructions used in the two cases are different, which is an unnecessary complication for a developer.

Learning 10: To make the internal logic of the service independent from the connection type, we can introduce the concept of *virtual port*, which is used by the network function to receive and forward the traffic, and we can dynamically generate the proper source code for any given port. However, the creation of this level of abstraction, so that eBPF programs are independent from the type of interconnection with the outside world, is certainly possible but requires a significant effort of the programmer as it is not explicitly foreseen by any available framework.

IV. Validation

This Section provides experimental evidence about the topics discussed in Section III, showing the impact of the main eBPF limitations and the improvements made in our prototypes. This evaluation leverages some of the network services we have implemented, hence exploiting real applications as a test-bench for our measurements.

A. Test environment and evaluation metrics

Our testbed encompasses two machines (Intel i7-4770 CPU running at 3.40GHz, four cores plus hyper-threading, 8MB of L3 cache and 32GB RAM) physically connected to each other through two direct 10Gbps links terminated in an Intel X540-AT2 Ethernet NIC. Both machines feature an Ubuntu Server 16.04.4 LTS, kernel 4.14.1[6], with the eBPF JIT flag enabled.

Throughput was tested by generating a unidirectional stream of 64B packets through Pktgen-DPDK 3.4.9, with a rate that is dynamically adjusted to achieve no more than 1% packet loss; depending on the test, packets may be looped back to the sender machine. Latency tests were carried out with Moongen [10], which generates the same traffic pattern as before but it exploits the hardware timestamp on the NIC to determine the traveling time of a frame when returns back to the sender; by default, one frame every millisecond is sampled. All tests were repeated ten times and the figures contain error bars representing the standard error calculated from the different runs.

Unless explicitly stated, all tests generate traffic so that only one CPU core is involved in the processing. Consequently, throughput in case of real deployment can be much higher than the reported values thanks to the session-based traffic load balancing provided by the Linux kernel, which automatically exploits multiple CPU cores for the processing.

B. Overcoming eBPF limitations

1) Slow-path forwarding performance: Section III-A showed the most important eBPF limitations that prevent the implementation of all the features required by complex network applications in the data path, which can be delegated to a more flexible userspace component, although with reduced performance.

To validate this module, we used our 802.1D bridge with two ports connected to the physical interfaces of the machine under test. The bridge service uses the slow path when a packet for an unknown MAC destination is received; in that case, it will be sent to the slow path module, which floods that packet to all output ports. To test this feature, we forced our bridge to send each packet to the slow path, from where they are forwarded to the output port.

Figure 2(a) shows the throughput calculated in different networking hook points, i.e., Traffic Control, XDP Generic and XDP Native between the slow path and the fast path. We can notice that, while the fast path forwarding varies depending on the hook point used (XDP performs better than the Traffic Control), the same is not valid for the slow path; the latter, in fact, uses the same mechanism to send the packet to userland regardless the hook point type to which the eBPF program is attached, hence representing the bottleneck for this test. Moreover, we notice that processing a packet in the slow path takes about three times more time than the same processing done in the kernel. In this case, the maximum forwarding rate is about 0.3Mpps regardless of packet size, since most of the time is spent in moving the packet to userspace (and back), significantly reducing the forwarding speed.

Figure 2(b) indicates the latency measurements calculated in the same conditions as before; we can notice that as long as the traffic remains below the maximum throughput calculated for each hook point, the latency values are almost the same regardless the hook type; this is due the batching mechanisms adopted by the driver that introduce a fixed cost on the packet processing. On the other hand, slow path processing incurs in a higher overhead due to the additional copy of the packet between kernel-userspace. Also, we notice how the latency grows when approaching the maximum value calculated in the previous test; in this case, the packet loss increases considerably with a consequent delay in the packet processing.

[6]We noticed that with newer kernels (e.g., 4.15/6) there is a marked performance deterioration, as shown in Figure 2(a), supposedly due to the fixes introduced after the Meltdown and Spectre vulnerability disclosure.

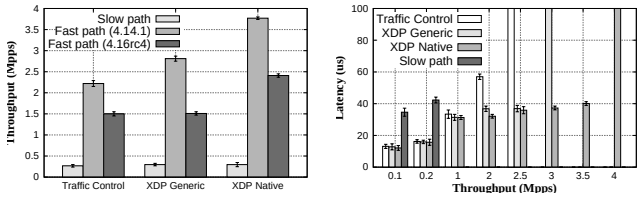

Fig. 2. Throughput (left) and latency (right) for the Bridge service when redirecting packets entirely in the fast path and when using the slow path.

For example, the throughput of the bridge service calculated in Traffic Control is 2.22Mpps, and the peak in latency is seen at 2.5Mpps (in the graph, we do not show the absolute value for readability reasons, but it is, in all three cases, over one millisecond). The same behavior is identified in XDP Generic and XDP Native, where the spike is at 3Mpps and 4Mpps respectively. These numbers indicate the importance of performing most of the actions in the eBPF fast-path, reducing the number of trips to the slow path. It is worth noticing that in real cases we have seen few packets to be raised as exceptions and sent through the slow path, being closer to 100% of the packets handled in the kernel for normal applications.

C. Enabling more aggressive service optimization

1) Code tailoring: To evaluate the potential benefits of this technique we took our Bridge service and we changed its features at runtime, enabling the VLAN and Spanning Tree (STP) and measuring the final throughput using the same traffic, independently from the enabled feature. Figure 3(a) shows how the complexity increases, in terms of BPF instructions, when functionalities requiring more complex actions are enabled. This complexity is also reflected in the overall forwarding performance of the service, where we can see a drop of up to 20% of the throughput between the baseline, in which acts as a simple L2 learning Bridge, to the full version of the bridge that supports VLAN and STP. Note that this performance improvement would not be possible without the *code tailoring* and the *dynamic reloading*, forcing the user to work with the full version of the bridge even if those features were not required at that moment.

2) Moving configuration data from memory to code: To estimate the goodness of this technique, we used our bridge service, comparing the actual throughput in the three different hook points, when the optimization is enabled and when it is not, as shown in Figure3(b). In this specific case, the optimization consists in moving the content of the filtering database from memory (i.e., a map) directly into the code, with the entries statically embedded in it, for example via a switch-case on the destination MAC address to forward the packet on the right port. Obviously, we set a maximum limit to the entries directly written in the code. This feature permits to optimize the most common case, by statically inserting the entry within the code, hence avoiding (costly) memory accesses, with performance benefits ranging from 5% or 10%, with the possibility of obtaining higher gains by combining this technique with the previous one.

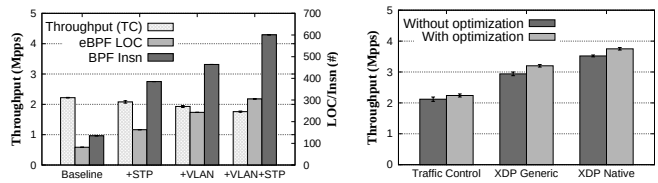

Fig. 3. Effect on the end-to-end throughput, using the code tailoring technique (left) and the moving configuration from memory to code (right).

TABLE I
RELOADING TIME OF VARIOUS EBPF SERVICES

Service	C LOC	BPF Insn	Compilation (ms)		Injection (ms)
			Kernel .h	UAPI .h	
Firewall*	1094	1564	8683	850	50
LBDSR	305	910	889	830	6
LBRP	470	723	885	853	2
Router	331	458	885	799	1
Bridge	243	464	854	236	2
NAT	564	441	847	809	1
DDoS	136	74	806	642	1

* This service uses a chain of eBPF programs; the time shown refers to the one needed to compile and inject the entire programs chain, which comprises several eBPF programs.

3) Reloading: This technique is heavily used in our services (in conjunction with the ones described above) to adjust the code injected in the kernel with the updated runtime service parameters. Table I shows the cost of this technique by comparing the reloading time for different services, which can be split into two pieces. The first one is the time needed to compile the C source code into eBPF assembly instructions while the second is the time required to inject the program in the kernel, which involves a pass in the in-kernel verifier.

We notice how the compilation phase consumes most of the reloading time (we leverage `bcc` [1] to appropriately package eBPF modules in the kernel) while only a small part, around 1%, is spent on the in-kernel injection. Also, we realized that using the Linux standard kernel headers introduces a considerable overhead as they internally include other headers that often are not needed by our eBPF program. By carefully selecting the headers present in the `uapi` directory, which are often smaller and more compact, we noticed a significant reduction in compilation time, as shown in the firewall service in Table I. In fact, this service uses multiple eBPF programs in the data path; optimizing the compilation time of every small program reduces the overall reloading time of the service. It is important to notice that `bcc` provides additional primitives to facilitate the interaction with the eBPF ecosystem; during the compilation phase, the code is then rewritten mapping the `bcc`-provided helpers into the corresponding eBPF functions. The compilation time shown in the Table I is the sum of three phases (of approximately the same duration) during which the code is pre-processed, rewritten and ultimately compiled producing the final BPF assembler code.

Some works such as [11] keep compiled versions of their services and then perform an optimization directly on the

compiled code, without this additional overhead. However, we believe that this mechanism limits the potential of previous techniques, reducing their possible optimizations. Although in this case the overall reloading time would not be negligible, as we explained in Section III-B3, we swap the existing program with its optimized version after the program has been compiled and injected, thus avoiding any service disruption.

D. High performance processing with XDP

1) XDP Native: The performance benefits of Native XDP services are evident from the previous figures. In fact, attaching the same program in XDP Native mode leads to an increase in performance of about 65% (Figures 2(a)-2(b)), allowing to achieve higher throughput. In addition, when comparing XDP programs with the other hook points at the same throughput, it brings to a significant reduction of CPU consumption due to the lower overhead for packets management. This speed comes with the limitation that programs of this type can only be used when the entry points of the chain are physical or virtual interfaces[7], while they must return to the normal stack delivery in case of different workloads (e.g., containers).

2) XDP Generic: Previous figures show that XDP Generic provides about a 25% of performance improvement compared to Traffic Control, allowing us to conclude that it can be used to speed up the performance of services even for drivers that do not have native XDP support. Generic XDP programs may be directly attached to virtual ethernet interfaces (veth) providing services to workloads such as containers, with hopefully a performance increment. However, this feature is hardly usable when containers are involved, due to the problem mentioned in Section III-D3.

E. Service function chaining

The ability to directly connect eBPF services to each other in the kernel (section III-E) is a significant advantage of this technology. This section evaluates the overhead of the tail calls in the three different hook points by using a simple program that forwards a packet between two physical interfaces, but whose code includes a growing number of tail calls. As shown in Figure 4, the overhead of the tail calls is almost negligible up to 8, which is enough for most of the applications, while it increases significantly with 16 and 32; the reason of this additional overhead is still unclear.

V. Conclusion

This paper presents our experience in developing complex network applications with eBPF, an up-and-coming technology that allows executing code at runtime in the Linux kernel, without the need to package custom kernel modules. We described the main limitations of this technology demonstrating how, in most cases, these can be circumvented without affecting the necessities of real network applications. In other cases, we have proposed alternative solutions to overcome these limitations. Thus, we identified and discussed several ideas related to the possibility of injecting code dynamically in the kernel that opens the way to several new optimizations strategies. Finally, we verified the real applicability of the proposed ideas and the consequent performance advantages, exploiting the several applications we created for our experiments. We are confident that this type of practical learnings can positively influence ongoing development and advancements in the field of eBPF-based network services and applications.

[7]Recently, support for receiving frames with XDP has been added to the tuntap driver [12].

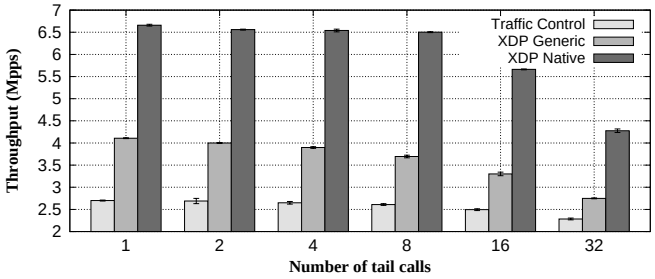

Fig. 4. End-to-end throughput with an increasing number of tail calls.

Acknowledgment

The authors would like to thank VMware and Huawei for their generous support, Y. Lu, J. Pi, A. Shaikh, P. Monclus, B. Blanco, A. Starovoitov, R. Bhagavatula, F. Bonomi, N. Iotti, F. Antonini, D. Siracusa for the inspiring discussions and their scientific help, and the many students and colleagues (I. Cerrato, N. Chiarappa, L. Ferrettino, S. Imperiale, A.P. Malinconico, F. Picciariello, G. Saitta) who collaborated to this project and contributed with ideas, code, comments.

References

[1] BCC authors. BPF Compiler Collection (BCC). [Online]. Available: https://www.iovisor.org/technology/bcc

[2] E. Cree. (2018, feb) Bounded loops for ebpf. [Online]. Available: https://web.archive.org/web/20180308141915/https://lwn.net/Articles/748032/

[3] Z. Ahmed, M. H. Alizai, and A. A. Syed, "Inkev: In-kernel distributed network virtualization for dcn," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 3, 2016.

[4] C.-C. Tu, J. Stringer, and J. Pettit, "Building an extensible open vswitch datapath," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 72–77, 2017.

[5] M. V. Bernal. (2018, feb) [iovisor-dev] handle pool of elements in ebpf maps. [Online]. Available: https://lists.iovisor.org/pipermail/iovisor-dev/2018-February/001241.html

[6] F. What is RCU. (2007, dec) Mckenney, paul e. and walpole, jonathan. [Online]. Available: https://web.archive.org/web/20180125051005/https://lwn.net/Articles/262464/

[7] D. Borkmann, *bpf: add csum_diff helper to xdp as well*, jan 2018, in Linux Kernel, commit 205c380778d0.

[8] J. D. Brouer. (2017, aug) Xdp redirect measurements, gotchas and tracepoints. [Online]. Available: https://web.archive.org/web/20180311113307/https://www.spinics.net/lists/xdp-newbies/msg00269.html

[9] ——. (2018, mar) Xdp redirect memory return api. [Online]. Available: https://web.archive.org/web/20180316133406/https://lwn.net/Articles/748866/

[10] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015, pp. 275–287.

[11] Cilium authors. HTTP, gRPC, and Kafka Aware Network Security and Networking for Containers with BPF and XDP. [Online]. Available: https://cilium.io/

[12] J. Wang and D. S. Miller. (2017, dec) Xdp transmission for tuntap. [Online]. Available: https://web.archive.org/web/20180315083526/https://lwn.net/Articles/742501/