Transparent and Dynamic Deployment of Lightweight Transport Protocols

El-Fadel Bonfoh, Djolo Cédric Tape, Christophe Chassot, Samir Medjiah

Transparent and Dynamic Deployment of Lightweight Transport Protocols

El-Fadel Bonfoh
LAAS-CNRS
University of Toulouse
Toulouse, France
efbonfoh@laas.fr

Djolo Cédric Tape
LAAS-CNRS
University of Toulouse
Toulouse, France
dctape@laas.fr

Christophe Chassot
LAAS-CNRS
University of Toulouse
Toulouse, France
chassot@laas.fr

Samir Medjiah
LAAS-CNRS
University of Toulouse
Toulouse, France
medjiah@laas.fr

Abstract—**The adoption of new Transport protocols on the Internet remains a critical challenge and their effective deployment is very slow. Till now, despite its known limitations and the plethora of existing alternatives like QUIC or DCTCP, almost *90%* of applications transmissions are based on TCP. From this observation, we assert that redirecting TCP-connection to another Transport protocol may accelerate the deployment and the adoption of any new Transport protocols in the Internet. The *selected* Transport protocol towards which the redirection is performed may either already exist in the OS or dynamically be deployed on it. Recently introduced in the Linux kernel, eBPF technology provides the abilities to insert at runtime functionalities in the kernel from userspace programs. In this paper, we propose a preliminary design of a eBPF-based framework to perform our approach. Following this design, we implement a prototype that safely (1) perform transparent redirection of TCP connections either to the OS native UDP or to UDP-Lite; the later one is (2) dynamically deployed as eBPF programs. This first prototype, developed on Linux 5.0, has the worth to demonstrate our concept but suffer from performances issues to which we formulate some solutions and open up the associated research questions. Nevertheless, we believe that our approach may lead to innovation at the Internet Transport layer.**

Keywords—***Transport protocols; TCP/UDP; UDP-Lite; eBPF; dynamic deployment; Linux OS***

I. Introduction

The Internet has known rapid and continuous development of Transport protocols not only to satisfy the functional requirements but also to meet applications QoS and, more recently, QoE requirements. Unfortunately, there is a wide gap between the development and the speed of deployment/adoption of those protocols. The Transport layer is only based on TCP and UDP, and nearly 90% of Internet traffic is based on TCP [1]. Deployment and adoption of new Transport protocols are slow and difficult.

Indeed, to support any new protocol, OS developers have to add it to the OS kernel; this is so tedious and error-prone to the point that it can only be motivated by high demand from application programmers. At the same time, for the application programmers, modifying an application to support any new protocol is time-consuming and quite complex [2]; this leads them to definitively use TCP despite its well-known limitations in many modern networks context [3].

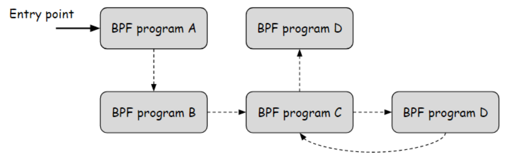

Fig. 1. eBPF programs chain by tail calls [15]

As if it was not already complicated enough, besides these two actors, there are middleboxes vendors which come to “violate” the *end-to-end principle* of the Transport layer; indeed, their devices (NAT, Firewall, etc.) usually block all the Transport protocols that are not supported by mainstream OS developers (Windows, Mac, Linux, etc.). This global context leads to a vicious circle that hampers the deployment and the adoption of any new Transport solutions other than TCP or UDP [4, 5], explaining the lack of innovation within the Internet Transport layer.

Given the huge number of TCP-based applications, we assert that the capability to, transparently for applications and safely for the system, redirect TCP connections to another Transport protocol can definitely accelerate the deployment and encourage the adoption of novel Transport protocols in the Internet. The *selected* Transport protocol towards which the redirection is performed may either already exist in the OS or dynamically be deployed on it. Indeed, every time a new protocol is released, thanks to dynamic deployment features, the OS developers do not have necessarily to upgrade their systems and the applications programmers can leverage the benefits of the new protocol without modifying their applications thanks to transparent redirection capabilities. To be complete and viable, this approach may integrate a systematic fallback to TCP in case of any failure (technical, middleboxes interferences, etc.). It is important to specify at this level that the prototype presented in this paper does not include this complementary mechanism.

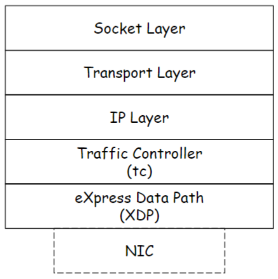

Fig. 2. Traffic Control and XDP

More recently, Linux extended the BPF [6] virtual machine to the know as eBPF. In short, eBPF allow to dynamically add user code in the kernel and so temporary modified its behavior. More about eBPF architecture and functioning is provided in the background section.

To show the feasibility of our approach, we preliminary design and implement a eBPF-based tool on Linux 5.0. This tool, which we call *Hooker* is a part of Virtual Transport Layer described in [4]. The *Hooker* framework leverages eBPF [7] features to transparently and safely redirect TCP connections to the OS native UDP or to the dynamically deployed UDP-Lite. The *selected* protocol could be any other than UDP and UDP-Lite. Through an illustrative data streaming scenario, we test our prototype and successfully perform the transparent redirection of TCP connections and the dynamic deployment of UDP-Lite as eBPF programs. Despite (1) the potential of our approach to make it easier the deployment of new Transport solutions, and then and to stimulate their adoption, and (2) its subsequent benefits in terms of flexibility, customization, and protocol adaptation, the current prototype face performance issues. In this paper, we formulate some preconizations and open up the related research questions.

The rest of the paper is organized as the following. The next section presents the background to this work and an overview of the related work. Section III presents the details of the preliminary design and implementation of the *Hooker* framework. In section IV, functional tests and performances analysis of the Hooker are presented. Finally, section V concludes the paper and enumerates the perspectives of this work and related opened research questions.

II. Background and Related work

A. Background

The *Hooker* framework developed in this work is based on eBPF. eBPF (extended Berkeley Packet Filter), as suggested by its name, is an extension of BPF in-kernel virtual machine allowing to inject bytecode within Linux kernel at runtime. Its main components are: (1) *maps* which are key / value stores used to exchange data either between user-space programs and in-kernel eBPF programs, or between eBPF programs running at different points of the kernel, (2) *helper functions* executed directly inside the kernel and allowing eBPF programs to interact with it, and (3) *tail calls* used to chain up to 32 eBPF programs as shown in Fig. 1. The latter feature can be used to overcome the size limitation of eBPF program, fixed to 4096 bytes.

eBPF is used in several contexts like tracing, monitoring, security, and networking. Each eBPF program may be attached to a *hook* point, a.k.a as *kernel event* (incoming packet, system calls, socket operations, …), and each time the event occurs, the eBPF program is executed. The two main networking hook points are *eXpress Data Path* (XDP) [8] and *Traffic Controller* (tc) [9] which are placed at different levels of the networking stack (see Fig. 2).

eBPF introduces programmability in the Linux kernel by allowing runtime code injection within the kernel and so, as illustrated in Fig. 3, provides the ability to dynamically add functionalities in the kernel from userspace programs compiled by LLVM/Clang compiler. Thanks to JIT compiler and eBPF Verifier, these functionalities are safely added and efficiently executed. Furthermore, eBPF has the advantage of allowing temporary kernel modification as opposed to kernel patches approach.

To dynamically deploy Transport functions in the OS kernel, we first considered the use of Linux kernel modules; permitting seamlessly integration and utilization of the introduced Transport functions. Unfortunately, we faced the most common issue of kernel module utilization: the whole system crashed at the slightest mistake; in other words, there is no verifier to guarantee the safety of the OS. Furthermore, there is a lack of debug tools and it takes time to check the code and to repair the bug.

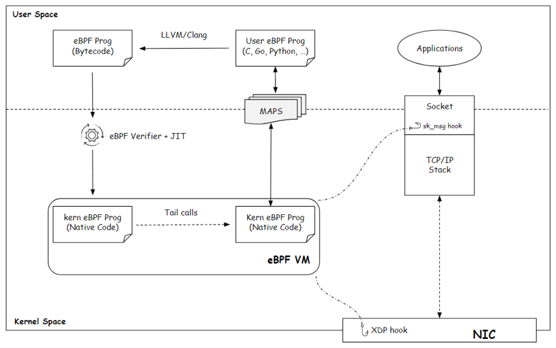

Fig. 3. eBPF Networking Overview

B. Related work

Various works have been done to tackle the issue of deployment and adoption of Transport protocols. [10, 11] propose TCP-SCTP mapping tool for transparent redirection of TCP connection to SCTP in order to gradually deploy SCTP. In [10], the mapping tool acts like a proxy which merges individual TCP connections into SCTP association whereas in [11] the mapping tool is designed to be directly integrated into the OS. However, such approaches address only SCTP deployment and adoption issues; they do not provide a comprehensive way for mapping to other protocols than SCTP.

More recent works [12, 13, 2] propose to rethink the entire Transport layer architecture in order to delegate to the Transport layer the choice of the protocol to be used; let us recall that this choice is currently let to the application developer. In [2], the authors assert that the main cause explaining the lack of new Transport protocols deployment and adoption comes from architectural limitations of the Transport layer, hence their proposal for a new architecture. Although this approach is promising, it requires rewriting existing applications and as we previously mentioned, this can slow down its adoption.

III. Design And Implementation Of The Hooker Framework

The purpose of this section is to present the internal structure of the *Hooker* framework and its interactions both with network stack and the applications. Prior to this presentation, it is worth to recall that the *Hooker*, in its current design, is aimed at demonstrating the concept and hence at opening the way to future performance optimization works.

The *Hooker*'s goals are (1) to redirect TCP connections to another protocol in a transparent manner for the application, and (2) to dynamically deploy the required Transport protocol if it does not already exist in the OS. To do that, the *Hooker* acts in a different manner on outcoming packets and incoming packets as depicted in Fig. 4, i.e. it places at different levels the hook points and uses several types of eBPF programs associated with those hooks.

On a host running the Linux operating system, the *Hooker* is attached to `root cgroupv2` [14]; hence, by leveraging hierarchical model of cgroups, it can process every ingress and egress packets of all processes running on the host. When the *Hooker*'s eBPF programs are loaded, a map of type `SOCKMAP` is created. The map key illustrated in code listing 1 is used by the *Hooker msg_redirector* component to identify the exact socket towards which the packet data must be forwarded to. Each time a connection is established or closed, the map is updated by the *Hooker msg_redirector* running `SOCK_OPS` program attached to `cgroupv2`.

```
struct sock_key {
    __u32 src_ip4;
    __u32 dst_ip4;
    __u32 src_port;
    __u32 dst_port;
}
```

Code listing 1: The structure of the sockmap key

The `SK_MSG` program running by the *Hooker msg_redirector* is executed each time an application process sends a message by invoking `sendmsg()` on a TCP socket. When the *Hooker userspace* invokes `recvmsg()`, it receives only the payload without Transport layer address information. By consequence, in order to allow the *Hooker userspace* to deliver data to the right process when it receives a response from remote pair host, before delivering message to the redirection socket, the *Hooker msg_redirector* rewrites it by adding one field in the header thanks to buffer extension permitted by the helper function `bpf_msg_push_data()`. Finally, to deliver the message either to the redirection socket or to the TCP socket, the *Hooker msg_redirector* leverages `bpf_msg_redirect_map()` helper function.

The redirection socket is created and maintained by the *Hooker userspace* which will use the `recv()` operation to get the redirected data packet and to send it on the right Transport protocol socket chosen between UDP and UDP-Lite. In this first prototype, UDP-Lite is dynamically deployed at the same time as the *Hooker*'s eBPF programs. The right moment and the right protocol to use and deploy

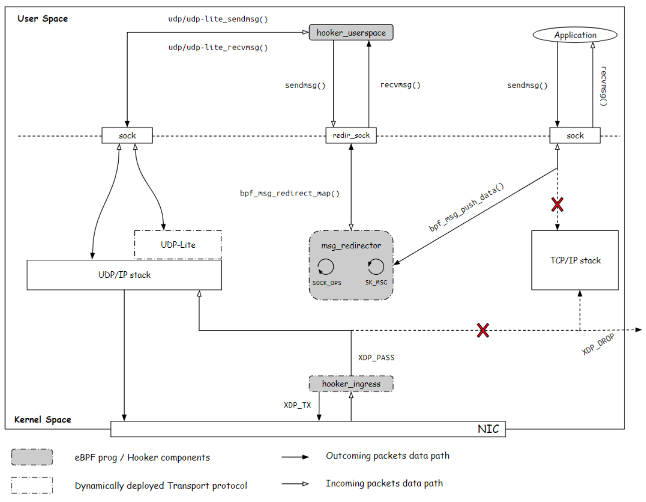

Fig. 4. Hooker Framework Design Overview

are let to future implementations. The UDP-Lite is chosen because it is a lightweight Transport protocol. Indeed, as stated in section II, to ensure that any eBPF program will terminate within quicktime, the maximum size of each program authorized by the eBPF Verifier is 4096 BPF instructions. However, we can overcome this limitation by partitioning the Transport protocol into small functions and using *tail calls* feature to chain them as shown in Fig. 1.

For incoming packets, two main hook points can be used, *XDP* and *tc*. *tc hook* has the advantage to be attachable on incoming datapath as well as outcoming datapath; however, it is triggered only after expensive allocation of `sk_buff` structure by the kernel networking stack. Even if the *XDP hook* avoids the overheads introduced by allocation of `sk_buff structure`, currently, it cannot be triggered on outcoming packet data. This is not crippling because outcoming data are hooked from socket layer, hence we use XDP driver hook in this first prototype to leverage it performance and to offset at the best the overheads generate in socket layer by *Hooker* redirection operations.

Every time, as soon as the host NIC driver receives a packet data, thanks to XDP driver hook, the *Hooker egress* intercepts the packet data and processes it with the associated eBPF program of type `BPF_PROG_TYPE_XDP`. The *Hooker egress* can drop the packet data (`XDP_DROP`), redirect it to the same or another network interface (`XDP_TX`, `XDP_REDIRECT`) or, as currently done in this prototype, pass it to the right network stack (`XDP_PASS`) for further processing.

IV. Preliminary Tests and Results

The proposed *Hooker* prototype is evaluated through experimentations based on data streaming of three files of different types (text, image, and video) having different size, 3KB, 510KB, and 26MB respectively. The main goal of these tests is to demonstrate the feasibility of the proposed approach. We measure the cost (delay and overheads) of the redirection and the dynamic deployment of *Hooker*'s programs and the UDP-Lite.

The testbed illustrated in Fig. 5 is composed of two Linux virtual machines (VMs), one acting as the data streaming server and the other one running the client application. The running Linux version for both VMs is 5.0. Each VM is provided with Intel Core i7-7700 CPU and 6.3GB of RAM. To make (1) UDP and UDP-Lite as reliable as TCP, and (2) TCP as fast as UDP and UDP-Lite, we emulated an IP network tuned to zero rate of loss and error. The RTT average is set to 50 ms. See Table 1 for the specifications of VMs and the emulated network.

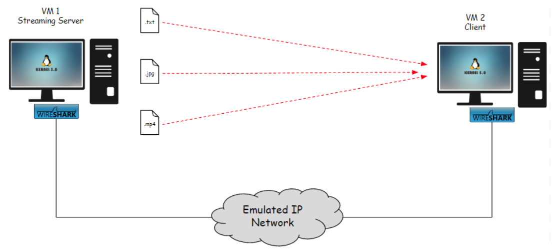

Fig. 5. Experimentations testbed

TABLE I. Testbed network and hosts configurations

VMs specifications		
OS kernel	***CPU***	***RAM***
Linux 5.0	Intel 2x3.6GHz	6.3GB
Emulated Network		
delay	***loss rate***	***error rate***
25 ms	--	--

TABLE II. Redirection and deployment costs

eBPF programs	Compilation	Loading	Redirection ops
Hooker's progs	6 s	20 ms	400 ns
UDP-Lite	200 ms	7 ms	--

For each file, we consider the identical scenario consisting in two steps: the first streaming is performed without running the *Hooker* and the second streaming is realized with the *Hooker* running at both sides of the connection (client and server). For each step, we check that the client correctly received the streamed file and with Wireshark analyzer [16], we validate the redirection by checking the Transport protocol used during streaming.

The performed performance evaluation is depicted in Fig. 6. It shows the impact of the *Hooker* on the transmission latency. Furthermore, we measure the additional overheads introduced by the *Hooker* framework (see Table 2). The *Hooker* cost is explained by (1) the redirection operations (multiple memory copies, etc.) and (2) the deployment of eBPF programs operations (compilation and loading). On average, the *Hooker* introduces 55 ms additional latency, which is not surprising considering the design and implementation choices made in this first prototype.

V. Conclusion and Future Research Directions

We propose to use transparent redirection and dynamic deployment to tackle Internet Transport layer innovation issues. We build a first prototype based on eBPF to perform this approach. This prototype, called *Hooker*, aims to demonstrate the feasibility of our idea and successfully redirect TCP connections to OS native UDP and to UDP-Lite. The UDP-Lite protocol is dynamically deployed by the *Hooker* framework.

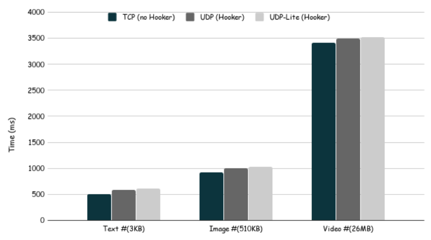

Fig. 6. Latency of the data streaming with and without Hooker

The proposed approach may accelerate the deployment and the adoption of new Transport protocols in the Internet. Furthermore, it can introduce innovation at the Transport layer by allowing the Transport protocol adaptation to the network context and to the application requirements. However, as we show it, the first prototype of the *Hooker* framework presented in this paper generates additional overheads leading to poverty of provided performances.

One way to optimize the performances is to rethink the architecture of the *Hooker*, especially its Hooker userspace component which acts currently as "application" proxy. This component may be back to the kernel to act like kernel proxy and so, allow avoiding multiple userspace - kernel space switching contexts. Furthermore, apart from design optimization, we have to rethink the way the eBPFs programs interact with the kernel, to say new helper functions and maps have to be released. For example, here instead of use `SOCKMAP` to perform the redirection, we can use a shared `DATAMAP` where the *Hooker userspace* may directly come to retrieve or to deliver a data and avoid expensive management of multiples buffers socket. This implies the introduction of permanent patches to the Linux kernel.

By showing (1) the benefits of our approach in terms of effective deployment and adoption of Transport protocols and (2) its feasibility, we believe that it will open the way to further optimization research to fully leverage transparent and dynamic deployment of Transport protocols benefits.

References

[1] D. Murray, T. Koziniec, S. Zander, M. Dixon, P. Koutsakis, “An analysis of changing enterprise network traffic characteristics”, IEEE Asia-Pacific Conference on Communication (APCC), 2014.

[2] M. Oulmahdi, C. Chassot, N. V. Wambeke, “Transport protocols: limitations, evolution obstacles and solutions for an actual deployment in the Internet”, International Journal of Parallel, Emergent and Distributed Systems, Taylor, 2015.

[3] E. Dubois, J. Fasson, C. Donny, E. Chaput, “Enhancing TCP based communications in mobile satellite scenarios: TCP PEPs issues and solutions”, Proc. of the 5th ASMS and 11th SPSC, September 2010.

[4] E.-F. Bonfoh, S. Medjiah, C. Chassot, J. Aguilar, “Towards the Virtualization of Transport-level Functions and Protocols”, 7th IEEE International Conference on Smart Communications in Network Technologies (SACONET’18), Oct 2018.

[5] M. Handley, “Why the Internet only just works”, BT Technology Journal, Vol. 24, No 3, July 2006.

[6] S. McCanne, and V. Jacobson, “The BSD Packet Filter: A New Architecture for User-level Packet Capture”, USENIX winter, Vol. 93, 1993.

[7] M. Fleming, “A thorough introduction to eBPF”, https://lwn.net/Articles/740157/, accessed 2019-05-08.

[8] T. Høiland-Jørgensen and al., “The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel”, 14th ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT ’18), Dec 2018.

[9] “Linux man page”, http://man7.org/linux/man-pages/man8/tc.8.html/, accessed 2019-05-08.

[10] M. Welzl, F. Niederbacher, S. Gjessing, “Beneficial Transparent Deployment of SCTP: the Missing Pieces”, IEEE Global Communications Conference (GlobeCom), December 2011.

[11] R. W. Bickhart and al., “Transparent TCP-to-SCTP translation Shim layer”, Unpublished master’s thesis, Delaware University, 2005.

[12] N. Khademi et al., “NEAT: A Platform- and Protocol-Independent Internet Transport API”, IEEE Com. Magazine, June 2017.

[13] “Transport Services (TAPS)” https://datatracker.ietf.org/wg/taps/about/, accessed 2018-08-05.

[14] “Linux man page”, http://man7.org/linux/man-pages/man7/cgroups.7.html, accessed 2019-05-08.

[15] BPF and XDP Reference Guide, https://docs.cilium.io/en/v1.5/bpf/#, accessed 2019-05-09.

[16] Wireshark, https://www.wireshark.org/, accessed 2019-05-08.