Tracing Network Packets in the Linux Kernel using eBPF

Mark Kovalev
Software Engineering Department
Saint Petersburg State University
Saint Petersburg, Russia
restonich@gmail.com

***Abstract*—During the development and maintenance of complex network infrastructure for a big project, developers face a lot of problems. Although there exist plenty of tools and software that helps to troubleshoot such problems, their functionality is limited by the API that Linux kernel provides. Usually, they are narrowly targeted on solving one problem and cannot show a system-wide network stack view, which could be helpful in finding the source of the malfunction. This situation could be changed with the appearance of a new type of tools powered by the Linux kernel's eBPF technology, which provides a flexible and powerful way to run a userspace code inside the kernel. In this paper, an approach to tracing the path of network packets in the Linux kernel using eBPF is described.**

***Keywords*—Linux, kernel, networking, tracing, eBPF**

I. Introduction

Software and hardware solutions are becoming increasingly complex, which leads to an increasingly complex network infrastructure that lies at the basis of such solutions. Such infrastructures could include multiple physical devices and virtual interfaces, various network namespaces, firewall settings, routing tables, packet filtering, networking protocols, and so on. These technologies are very powerful and feature-rich, but on the other hand, it also makes troubleshooting of such network systems much harder and more time-consuming.

There are a lot of ways to troubleshoot networking problems. One of them is to walk through the OSI stack: go all the way up from the link layer to upper ones checking the system to work correctly on each layer. Check if the network interface is working and is configured right, look at the ARP and routing tables, firewall rules, packet filtering, and move on to check the high-level configurations. Most problems are solved in one of these steps. But complexity of the system configuration will eventually lead to non-obvious relations between different parts of it and more difficult problems will appear. Such problems are solved by excluding possible sources of malfunction one by one with different tools. This is a long and tedious process, and it needs to be repeated for every problem over and over since, for each problem, possible sources of malfunction are new and need to be rechecked. Most of the tools, being narrowly targeted on solving specific issues, do not help either. Though doing their job very well, they cannot provide a system-wide network stack view, which could help us solve non-obvious problems in complex network infrastructures.

In this paper, the "system-wide network stack view" means a path of the network packet through Linux's networking stack. It shows which functions processed the packet and for how long, where it was consumed or dropped, or if it went the wrong way, not intended by the network architecture. With this information, the developer could narrow down the scope of troubleshooting and solve the problem quickly with the use of the appropriate tools.

In the past, information about the packet's path would not be possible to obtain without direct kernel code modification or some serious restrictions. Now it can be easily done with the use of eBPF technology.

II. Technology Overview

A. BPF

Berkley Packet Filter (BPF) is a technology that consists of the register-based virtual machine and the instruction set for that machine. It was designed for a highly optimized and performant network traffic filtering [1]. It is used as the backend for the `libpcap` library and does packet filtering for tools such as `tcpdump`. When `tcpdump` is executed with some filtering rule, it generates BPF bytecode for that rule and sends it to the kernel to attach at the early stages of the network stack processing. That bytecode then gets interpreted on the virtual machine and decides which packet shows up in `tcpdump`'s output [2].

This filtering mechanism is performant and secure by design. BPF programs executed isolated on the in-kernel virtual machine. They are limited to 4096 instructions, cannot have loops, and all memory accesses are checked for a valid range. So, execution of the BPF bytecode is guaranteed to terminate; it cannot cause a kernel fault, a denial of service, or memory damage.

While being a useful concept of securely running userspace code in the kernel, BPF is limited by its design and age. Two registers are not enough to write powerful programs, and the instruction set is outdated as modern processors moved to 64-bit architecture. So, to take advantage of contemporary hardware, BPF had to be improved [3].

B. eBPF

Massive rework of the BPF was initiated in 2014 by Alexei Starovoitov [4][5]:

1. 512 bytes multi-use stack space replaced old spill-fill stack.

2. The number of registers was increased from two to ten, and their width became 64-bit. All of them map one to one to hardware registers.

3. Various old instructions were modified, and new ones added, all of them becoming a close match to the hardware instructions. It greatly improved JIT compilation.

4. Maps were introduced — generic key-value data structures to exchange information between the BPF programs and with the userspace.

5. New attachment points for the programs were implemented: kernel probes (kprobes), tracepoints, perf events, and sockets. These programs are invoked every time an attachment point is passed by, and they have access to the corresponding context.

These improvements significantly increased the programmability and performance of BPF. After some other modifications, API of that rework was frozen and named as extended BPF (eBPF) [6]. Since then, eBPF has been actively developed, providing more usability and flexibility for extending the Linux kernel's functionality without editing its source code. There is an example of the Linux TCP stack extension from the user space with the help of eBPF [7].

Plenty of attachment points makes eBPF a very useful technology for creating tracing and profiling tools. **`bcctools`** and **`bpftrace`** are great examples that make use of this functionality. A thorough explanation of how to use these tools in performance testing can be found in Brendan Gregg's "BPF Performance Tools" [8].

C. *Toolchain*

Classic BPF programs were written directly in VM instructions. This approach would be restricting for the eBPF as it would be harder to use new features and extensions. To simplify programming, the eBPF backend for LLVM was introduced [9]. It allows to write eBPF programs in restricted C language and then compile them to the ELF objects with a `clang`. Restrictions for C language come from the eBPF design and features of ELF parsing:

1. Main program functions and map structures have to be defined with `section()` attribute as loaders need eBPF objects to be self-contained in the ELF sections.

2. Multiple programs can be described inside a single C file in different sections.

3. No global variables, constant strings, or arrays allowed.

4. The stack is limited to 512 bytes.

5. No ability to call library functions (except for those defined with `inline` in included headers or for eBPF helpers).

6. Only bounded loops are available.

These are not all of the limitations and features of writing eBPF programs in C. An up-to-date list with explanations can be found in Cilium's BPF and XDP Reference Guide [10]. Since technology is being actively developed, some restrictions are being fixed. For example, bounded loops were introduced relatively recently, and before that, no loops at all were allowed in the eBPF programs (or they had to be unrolled with `pragma` directive) [11].

The eBPF helpers are special in-kernel functions intended to expand functionality and ease programming. They are used to interact with the eBPF maps, get the time elapsed since system boot, print some information for debugging, edit the network packets, and many more. The exact set of helpers accessible by the eBPF program is determined by its type [12].

Successfully compiled ELF objects with eBPF objects (programs and maps) are passed to the kernel via loader. This is done via `bpf()` syscall, but for ease of use, `bpftool` loader backed by `libbpf` library should be used instead. Though `bpftool` covers overall management of eBPF objects, the attachment eBPF programs to the network path should be done via the `tc` tool from the `iproute2` suite.

When the eBPF program is loaded into the kernel, it is processed by a static verifier. A directed acyclic graph is created from the program to check for the loops and unreachable instructions. Then the verifier simulates the execution of the program for every possible path and observes the state change of registers and stack. If the program passes, a descriptor is created for it that is then used to attach it to an appropriate attachment point.

III. Implementation

A. *Program flow*

The tool has a command-line interface, taking a filter expression as an input and passing network packet path in plain text format as an output. The general sequence is shown in Fig. 1:

1. Filter expression describing network traffic that needs to be observed is passed to the tool.

2. eBPF program for the network path traffic control attachment point (TC program) is generated from that filter and loaded into the kernel along with `skb_map` and `path_map` eBPF maps.

3. eBPF programs for the attachment to kprobes of the main network functions (kprobe programs) are compiled and loaded into the kernel.

4. The TC program matches every packet against the filter. Upon finding a match, the program stores a pointer to the packet's `sk_buff` structure in the `skb_map`.

5. The kprobe program checks arguments passed as the probed function's context, and if it matches the pointer stored in the `skb_map`, program stores current timestamp in the `path_map`. If the program's probed function is marked as the last in packet path, it also fills the `skb_map` with 1's. That serves as a signal for the tool to stop the packet tracing.

6. The tool retrieves information from the `path_map`, sorts it by the timestamp values and passes it as the output.

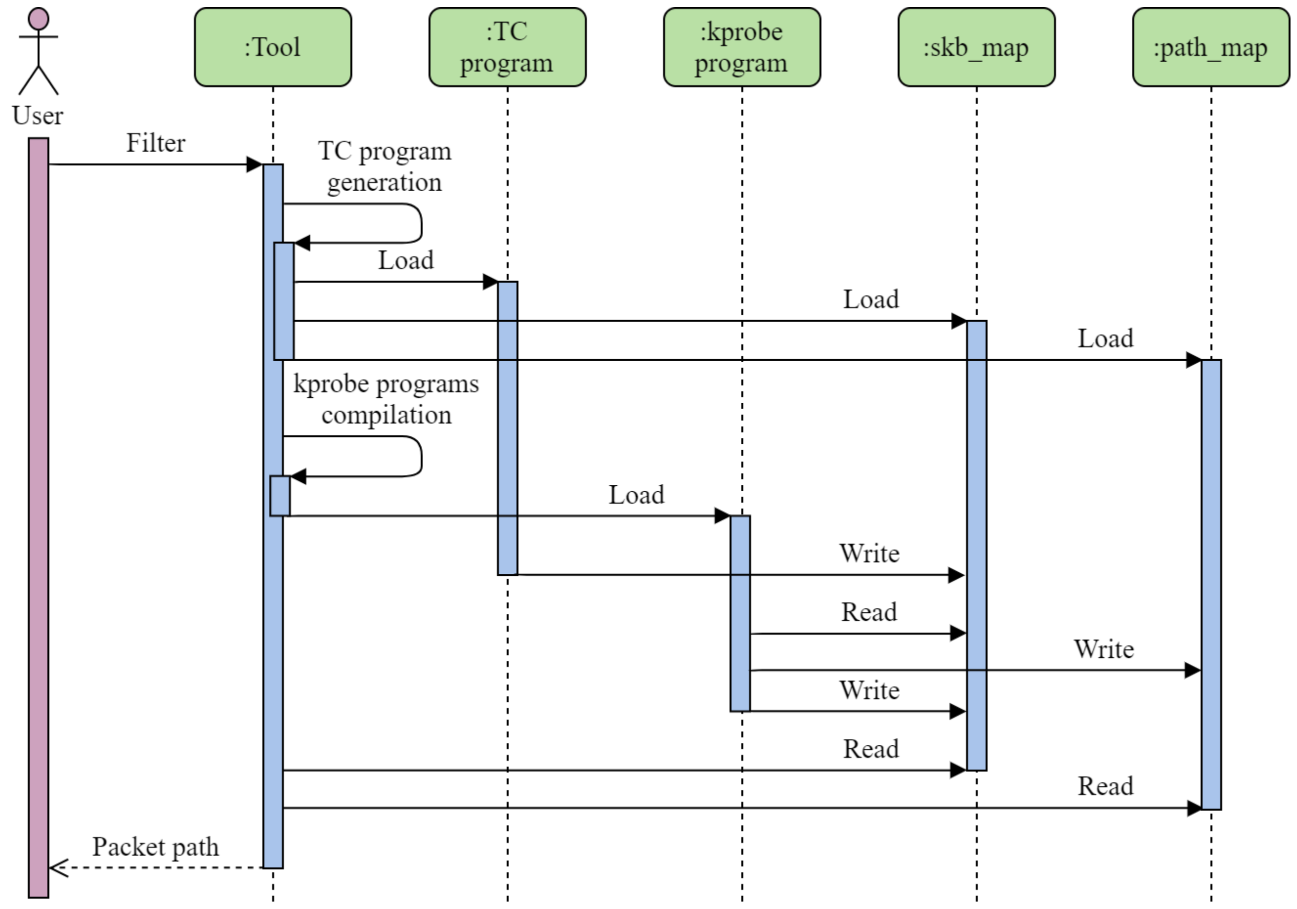

Fig. 1. Program flow

B. *Implementation details*

Every eBPF program has a different context passed to it based on the program type (which depends on the attachment point). For the TC programs, it is a `struct __sk_buff *skb` and for the kprobe programs — `struct pt_regs *ctx`.

`struct __sk_buff` is a user-accessible mirror of the in-kernel `struct sk_buff`. It is not a copy, more like access instructions. Accesses to the fields of this struct are processed in the eBPF verifier and translated to the accesses to the same fields of the real buffer structure. This approach improves security and portability, as programs do not rely on the in-kernel definition of the `sk_buff`.

`struct pt_regs` stores saved registers of the probed function. Arguments of the function are accessed from this structure by architecture-dependent macros, which improves portability of the kprobe functions.

Typically, eBPF programs cannot store their state. Therefore, eBPF maps are used to save the observed packet's pointer and to collect the information about its path. This allows to filter the packet only once in the early stages of its processing and to make the kprobe programs as simple as possible.

The TC program is attached to the appropriate point with the `tc` tool. `clsact` qdisc is added to the network interface, through which the observed traffic will be passing, and the TC program is passed to it as a classifier. Full command reference can be found in the `tc` man page [13].

To load and attach the kprobe programs, I've implemented a program based on the `libbpf` library, as `bpftool` lacks such functionality. My loader also replaces eBPF maps in loaded objects for those already created by the TC program load. This is necessary to establish communication between programs.

The Code of the kprobe programs is basically the same and simple. To attach them to the various kernel functions they are identified by the macros:

- `KP_NUM` stores a number of the probed function;
- `KP_SEC` stores a name of the probed function in the format `"kprobe/<function_name>"`;
- `KP_FIN` stores 1 if probed function marked as the last one and 0 otherwise.

These values are taken from a `kp_funcs.txt` file and are filled on the compilation phase with the use of the `clang`'s `-D<macroname>=<value>` option. This way kprobe programs for all the observed functions are created from the only one source file. This mechanism creates unnecessary overhead and is to be changed for a more suitable solution.

`kp_funcs.txt` file used in the kprobe program's compilation is necessary to provide portability for the tool. It is to be adapted for different Linux versions as names of the in-kernel functions change from time to time. Also, users can easily add new functions to this list to observe if the packets pass through them.

On the Linux systems `/etc/security/limits.conf` file controls limits of the various system values such as maximum file size, stack size, or processes count [14]. For my tool `memlock` value is the most important one. It is a maximum locked-in-memory address space that limits an amount of the memory pages in RAM that are not to be placed in the swap space. So, to be able to load a large amount of kprobe programs, this limit needs to be increased by the user.

IV. Similar approaches

A. *VMware Traceflow*

Traceflow is a part of the VMware NSX Data Center for vSphere platform [15]. It injects packets into the network and traces them as they travel between nodes. It provides a good overview of the whole network, from which an administrator could get information about possible sources of malfunction or performance reduction.

Traceflow operates on a high level of networking and does not tell about internal packet processing. My tool can operate only on one node yet provides a thorough network packet path through the kernel. Additionally, it does not inject special traffic into the system and works on the existing one. Also, usage of the Traceflow is restricted to the vSphere platform, while my solution runs on any system with a recent enough Linux kernel version.

B. *ftrace*

ftrace system [16] also could be used to trace network packets in the Linux. This could be done by restricting all network traffic in the system except for the one that is to be observed. Then `function_graph` tracer can be used on the function such as `__netif_receive_skb_list_core()` to show the path of the incoming packet.

It is an easy and detailed approach, though not so useful on a production system. My tool traces determined network packets, and the presence of another traffic in the system does not interfere with it.

C. *tcpdrop*

`tcpdrop` tool is a part of the BCC (BPF Compiler Collection) project and is built on the eBPF [17]. It provides a stack trace of Linux kernel functions that led to the drop of the TCP packet along with other useful information. This helps answer why such drops are happening.

Though being limited to only TCP traffic and not observing the full path of the packets, `tcpdrop` shows a good example of the usefulness of the approach that lies in the foundation of my tool.

V. Future Work

The current state of the tool is as follows:

1. The TC program is static and can trace incoming ICMP, TCP, or UDP packets with a manual macro modification.

2. The kprobe programs are compiled manually with appropriate values passed.

3. Information of the packet passing through functions is printed by programs via `bpf_trace_printk()` helper and is observed through the `/sys/kernel/debug/tracing/trace_pipe` file.

To reach an MVP (minimum viable product) state for the tool, the following things are to be implemented:

1. The generation of the TC program based on the filter passed.

2. A `kp_func.txt` file composition and a kprobe programs compilation automatization.

3. A `path_map` information retrieval mechanism and a packet path composition.

At the scope of the whole project, there are several points of consideration:

1. Amount of the kprobe programs that is acceptable to sustain a balance between performance and usability of the tracing information.

2. Different use-cases need to be described as well as scenarios of troubleshooting comparison with and without this tool.

3. Kprobes are not a part of the stable Linux API, the name of the functions could change. This should be handled to guarantee the operation of this tool on various kernel versions. Also, it is possible that thorough kprobe tracing is unnecessary in some scenarios, so instead the tool could rely on the tracepoints as a more stable kernel API.

The tool source files can be found in my GitHub repository [18].

References

[1] Steven McCanne, Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", Lawrence Berkeley Laboratory, December 1992. http://www.tcpdump.org/papers/bpf-usenix93.pdf

[2] Marek Majkowski, "BPF - the forgotten bytecode", The Cloudflare Blog, May 2014, https://blog.cloudflare.com/bpf-the-forgotten-bytecode/

[3] Matt Fleming, "A thorough introduction to eBPF", LWN, December 2017. https://lwn.net/Articles/740157/

[4] Alexei Starovoitov, "net: filter: rework/optimize internal BPF interpreter's instruction set", Linux project, commit, March 2014. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8

[5] "Linux Socket Filtering aka Berkeley Packet Filter (BPF)", Linux in-kernel documentation. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/filter.txt

[6] Alexei Starovoitov, "net: filter: split filter.h and expose eBPF to user space", Linux project, commit, September 2014. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=daedfb22451dd02b35c0549566cbb7cc06bdd53b

[7] Viet-Hoang Tran, Olivier Bonaventure, "Making the Linux TCP stack more extensible with eBPF", Netdev 0x13, 2019. https://netdevconf.info/0x13/session.html?talk-tcp-ebpf

[8] Brendan Gregg, "BPF Performance Tools", December 2019, Addison-Wesley Professional, ISBN-13: 9780136554820.

[9] Alexei Starovoitov, "BPF backend", LLVM project, commit, December 2014. https://reviews.llvm.org/D6494

[10] "BPF and XDP Reference Guide", Cilium. https://docs.cilium.io/en/latest/bpf/

[11] Daniel Borkmann, Alexei Starovoitov, "Merge branch 'bpf-bounded-loops'", commit, June 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=94079b64255fe40b9b53fd2e4081f68b9b14f54a

[12] "BPF-HELPERS - list of eBPF helper functions", manual page. http://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[13] Bert Hubert, "tc - show / manipulate traffic control settings", manual page. http://man7.org/linux/man-pages/man8/tc.8.html

[14] Cristian Gafton, "limits.conf - configuration file for the pam_limits module". http://man7.org/linux/man-pages/man5/limits.conf.5.html

[15] VMware Docs, "VMware NSX Data Center for vSphere Documentation" Traceflow documentation, May 2019. https://docs.vmware.com/en/VMware-NSX-Data-Center-for-vSphere/6.4/com.vmware.nsx.admin.doc/GUID-233EB2CE-4B8A-474C-897A-AA1482DBBF3D.html

[16] “ftrace - Function Tracer”, Linux in-kernel documentation. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/trace/ftrace.rst

[17] Brendan Gregg, "Linux bcc/eBPF tcpdrop", Brendan Gregg's Blog, May 2018. http://www.brendangregg.com/blog/2018-05-31/linux-tcpdrop.html

[18] Mark Kovalev, bpfpath, GitHub repository. https://github.com/restonich/bpfpath