eBPF Survey Check Point 2

Zhaoyuan Su, Renyuan Liu
George Mason University
zsu, rliu23@gmu.edu

Abstract

The objective of this eBPF survey is to give both an overview of the current general features and a detailed exploration for specific features of eBPF tools and applications, and to provide both qualitative and quantitative data support for both of these questions. The purpose of this survey is to give a quick insight to newcomers who are not familiar with eBPF tools and applications, and hopefully to give an inspiration to related researchers.

1 Introduction

Linux divides its memory into two distinct areas: kernel space and user space. Kernel space is where the core of the operating system resides. It has full and unrestricted access to all hardware — memory, storage, CPU, etc. Due to the privileged nature of kernel access, kernel space is protected and allows to run only the most trusted code, which includes the kernel itself and various device drivers.

User space is where anything that is not a kernel process runs, e.g. regular applications. User space code has limited access to hardware and relies on code running in kernel space for privileged operations such as disk or network I/O. For example, to send a network packet, a user space application must talk to the kernel space network card driver via a kernel API referred to as "system calls".

While the system call interface is sufficient in most cases, developers may need more flexibility to add support for new hardware, implement new file systems, or even custom system calls. For this to be possible, there must be a way for programmers to extend the base kernel without adding directly to the kernel source code. Linux Kernel Modules (LKMs) serve this function.

Unlike system calls, whereby requests traverse from user space to kernel space, LKMs are loaded directly into the kernel. Perhaps the most valuable feature of LKMs is that they can be loaded at runtime, removing the need to recompile the entire kernel and reboot the machine each time a new kernel module is required. As helpful as LKMs are, they introduce a lot of risks to the system. Indeed, the separation between kernel and user spaces adds a number of important security measures to the OS. The kernel space is meant to run only a privileged OS kernel, with the intermediate layer, labelled as Kernel Services in the picture above, separating user space programs and preventing them from messing with finely tuned hardware. In other words, LKMs can make the kernel crash. Additionally, and aside from the wide blast radius of security vulnerabilities, modules incur a large maintenance cost in that kernel version upgrades can break them. eBPF is a more recent mechanism for writing code to be executed in the Linux kernel space that has already been used to create programs for networking, debugging, tracing, firewalls, and more. Born out of a need for better Linux tracing tools, eBPF drew inspiration from dtrace, a dynamic tracing tool available primarily for the Solaris and BSD operating systems. Unlike dtrace, Linux could not get a global overview of running systems, since it was limited to specific frameworks for system calls, library calls, and functions. Building on the Berkeley Packet Filter (BPF), a tool for writing packer-filtering code using an in-kernel VM, a small group of engineers began to extend the BPF backend to provide a similar set of features as dtrace. eBPF was born.

eBPF allows regular userspace applications to package the logic to be executed within the Linux kernel as a bytecode. These are called eBPF programs and they are produced by eBPF compiler toolchain called BCC. eBPF programs are invoked by the kernel when certain events, called hooks, happen. Examples of such hooks include system calls, network events, and others. Before being loaded into the kernel, an eBPF program must pass a certain set of checks. Verification involves executing the eBPF program within a virtual machine. Doing so allows the verifier, with 10,000+ lines of code, to perform a series of checks. The verifier will traverse the potential paths the eBPF program may take when executed in the kernel, making sure the program does indeed run to completion without any looping, which would cause a kernel lockup. Other checks, from valid register state and program size to out of bound jumps, are also carried through. From the outset, eBPF sets itself apart from LKMs with important safety controls in place. Only if all checks pass, the eBPFprogram is loaded and compiled into the kernel and starts waiting for the right hook. Once triggered, the bytecode executes. The end result is that eBPF lets programmers safely execute custom bytecode within the Linux kernel without modifying or adding to kernel source code. While still a far cry from replacing LKMs as a whole, eBPF programs introduce custom code to interact with protected hardware resources with minimal risk for the kernel.

2 General Metrics Analysis

We designed a web crawler and downloaded the top 1000 from a total of 1064 (as of April 7, 2022) eBPF software and applications from Github based on search ranking. Then by using CLOC, a tool that counts blank lines, comment lines, and physical lines of source code in many programming languages, we analysed the multiple general feature statistics for each eBPF, including number of files, number comment lines, number of code lines,and the programming languages used for eBPF. The result is showing below.

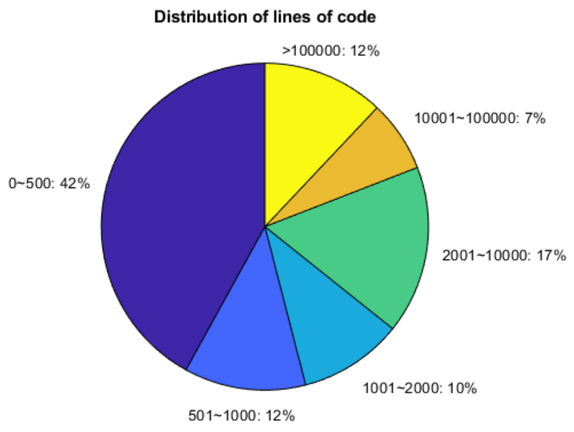

Figure 1: Distribution of lines of code

We can find out that most of the tool consists less than 500 lines codes, while there are still 12 percents of tools consists more than 100k lines.

Assuming that more comments indicates more helpful tool, we can find that more than 20 percents of the tools have more than 1000 lines of comments from Fig. 2.

And From Fig. 3, we can find out most of the tools consist of less than 20 files.

we also analysed the main programming language used by every tools. We can find from Fig. 4 that C/C++ is the most popular language for eBPF tools which matched the expectation.

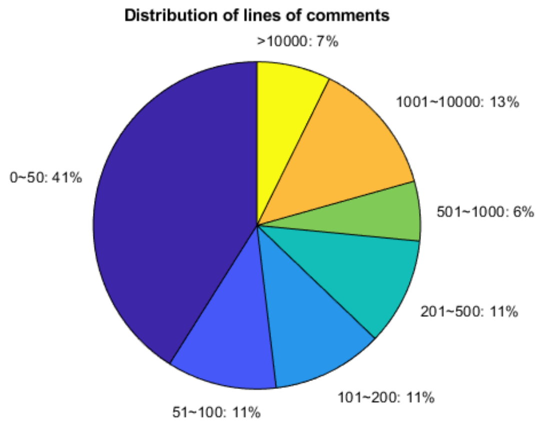

Figure 2: Distribution of lines of comments

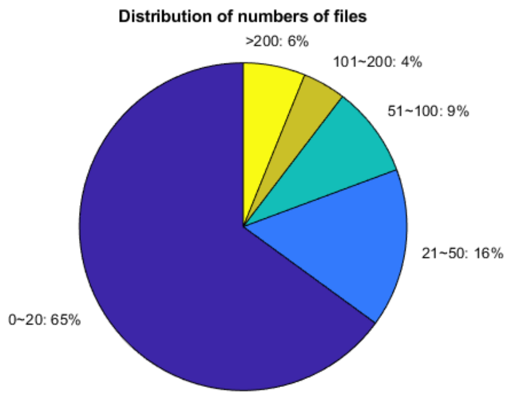

Figure 3: Distribution of numbers of files

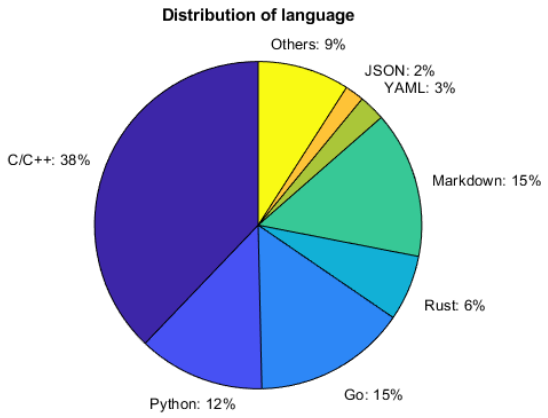

Figure 4: Distribution of languages

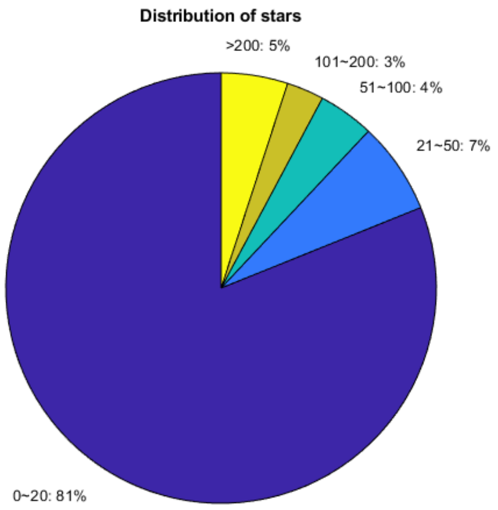

Figure 5: Distribution of stars

The number of stars of an open source project in Github can reflect the degree of attention the project receives from the community. Figure. 5 shows the distribution of the number of stars for 1000 eBPFs.

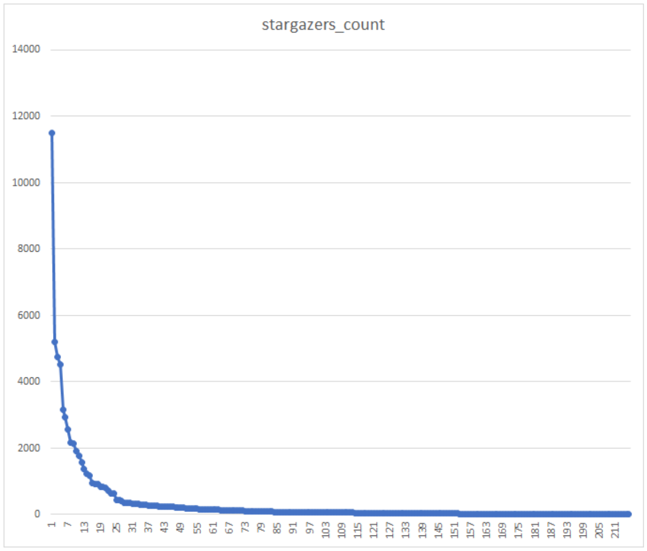

Figure 6: number of stars

From Fig. 5 and Fig. 6 we can see that only greater than 5 percent of the projects have more than 200 star count. And only a very small percentage of eBPF tools have gained more community attention.

3 Related Work

The work from Cyril Cassagnes shows that container engines are strengthening their isolation mechanisms [1]. Therefore, non-intrusive monitoring becomes a must-have for the performance analysis of containerized user-space application in production environments. The work from Vieira presents eBPF covering the main theoretical and fundamental aspects of eBPF and XDP [2]. After a literature review and background of Linux subsystems and container isolation concepts, we present our lessons learned of using the extended Berkeley packet filter to monitor and profile performance. We carry out the profiling and tracing of several Interledger connectors using two full-fledged implementations of the Interledger protocol specifications.

4 Conclusion

We implemented a web crawler and downloaded all open source eBPF tools and applications from Github. We analysed these codes and conducts some general features. We will analyse the function distributions for some top stars projects in terms of some technique aspects like network and security [3] later.

References

[1] Cyril Cassagnes, Lucian Trestioreanu, Clement Joly, and Radu State. The rise of ebpf for non-intrusive performance monitoring. pages 1–7, 04 2020.

[2] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[3] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.