A Study of Linux eBPF

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

Abstract

The objective of this eBPF study 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. We crawled 1000 eBPF applications from the open source website Github based on search rankings, and then conducted a general automated analysis of all these applications as well as a detailed analysis of some of the popular ones. The purpose of this study 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 Methodology

In this section, we first give a brief description of our analysis metrics on eBPF applications. Then we present the workflow we followed to analyze these dimensions. Finally, we discuss the limitations of our methodology.

2.1 Analysis Metrics

Our goal 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. In order to answer these questions, we design features to analyze eBPF applications.

General Features	
Category	Research Question
Function classification	Which functional category does each eBPF tool or application belong to?
size	What is the software size of each eBPF tool or application? how many files and lines of code are there in total?
programming language	What programming language is used for each eBPF tool? Which programming language is the mainstream of eBPF tool development? what is the distribution of each programming language?
Open Source Community Reviews	How many stars / forks / issues / requests each tool has on github?
hooks and calls	Does each eBPF tool or application use hooks? Which hooks are used? What is the probability distribution like?
Running Platform	What is the distribution of the system platforms targeted by each eBPF tool or application, such as Windows, Linux, Mac, etc.?
bugs	What are the bugs of each eBPF tool or application? How are the types of bugs distributed? How many bugs has been fixed and how many bugs has still been open?

Figure 1: general features

First we will discuss the general Features of eBPF(Fig. 1), which are described for each feature as follows. Functional classification can be used to describe which specific functional category an eBPF belongs to, and generally speaking the commonly used functions are developed by more software. size is used to describe the software size of an eBPF. Open Source Community Reviews are used to measure the popularity of an eBPF software on open source platforms, and generally the better the quality of the software the more popular it is. In this study, we mainly focus on the eBPF software running on Linux.

Specific Features	
Category	Research Question
Security	What system security features are eBPF developers concerned about? What methods do current eBPF tools use to improve system security? What are the obvious advantages and disadvantages of these methods compared to the system default methods?
Tracing	What system runtime behaviors are of concern to eBPF developers in tracking the flow of users running applications? Instead of requiring the export of vast amounts of sampling data as typically done by default system, what data structures and algorithms are used by the advanced eBPF tools and applications to provide tracing system performance insight?
Networking	In which situations is eBPF best suited for programming networks? which properties of eBPF give it this advantage? Which compiler do most eBPF developers choose to use for networking and why?
Monitoring	Which system metrics are most concerned by eBPF developers? How does dBPF reduce the overall system overhead for monitoring?

Figure 2: specific features

Then we divided the eBPF software into four main categories according to their functional characteristics(Fig. 2), which are Security, Tracing, Networking and Monitoring. For each category we analyze a certain number of metrics in a segmented way, such as that what system security features eBPF developers are concerned about and what metrics eBPF applications mainly trace on.

2.2 Workflow

In this subsection, we briefly describe the research workflow we follow and give an overview of the experimental design and conduct.

Figure 3: workflow

After identifying the scientific problems, we implement a web crawler to crawl the eBPF software on an open source website Github. On the one hand, we at the same time crawl the measurement information of the software in the open source community to support the general feature analysis. On the other hand, We implement an eBPF software analysis tool

to automate the analysis of general features of software such as programming language, number of code lines. Then we sort the eBPF by certain features and select a certain number of software for analysis of specific features. The results of all the analyses will be aggregated to obtain a summary of study.

2.3 Limitations

Our study focuses on eBPF software on the Linux platform, which may not reflect other operating systems, such as other windowing systems and Mac systems. Our analysis of the general features of all crawled eBPF software is done automatically using software analysis tools, and this process may not guarantee complete accuracy. We are only able to perform a detailed manual analysis of some of the features of some of the software because the process is too time consuming and labor intensive.

3 Results and Analysis

3.1 Data Crawling

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. We employed GitHub's official API to get information about each repository.

```
def download_git(urls, OUTPUT_FOLDER):
    count = 1
    for pageUrl in urls:
        data = json.loads(json.dumps(getUrl(pageUrl))
            )
        for item in data['items']:
            user = item['owner']['login']
            url = item['clone_url']
            fileToDownload = url[0:len(url) - 4] + "/
                archive/refs/heads/master.zip"
            fileName = item['full_name'].replace("/",
                "#") + ".zip"
            try:
                time.sleep(
                    DELAY_BETWEEN_QUERIES)
                wget.download(fileToDownload, out=
                    OUTPUT_FOLDER + fileName)
                count += 1
            except Exception as e:
                print("Could not download file {}".
                    format(fileToDownload))
                print(e)
```

We also crawled open source information for each application, including but not limited to the number of stars, forks, and open issues. The total size of all crawled eBPF applications is 4.4G.

3.2 General analysis

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.

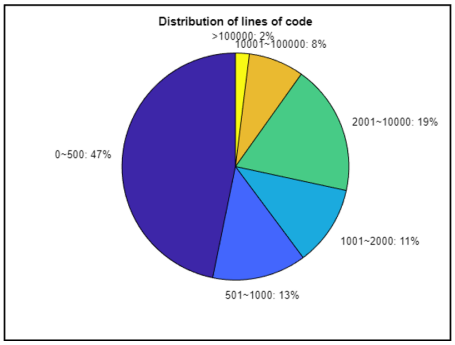

Figure 4: Distribution of lines of code

We can find out that most of the tool consists less than 500 lines codes, while there are still 10 percents of tools consists more than 100k lines. This indicates that the development workload of eBPF spans a wide range, which can be from a few hundred lines to over 100,000.

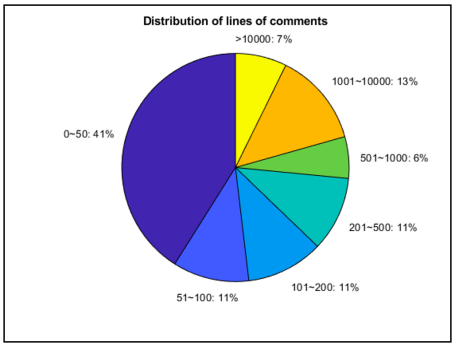

Figure 5: Distribution of lines of comments

Assuming that more comments indicates more helpful tool, we can find that more than 20 percent of the tools have more than 1000 lines of comments from Fig. 5. But more than 40 percent of the software has less than 50 lines of comments, and more than half of the software has it less than 100 lines, indicating that most eBPF software does not have a good user manual or instructions.

we also analysed the main programming language used by every tools. We can find from Fig. 6 that C/C++ is the most popular language for eBPF tools, which matched the expectation. In fact that, python language accounts for 12 percent of all programming because of eBPF applications can be wrapped into a Python library.

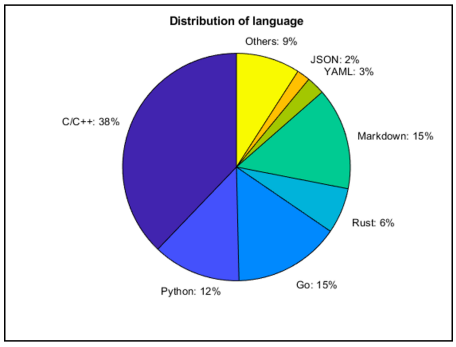

Figure 6: Distribution of languages

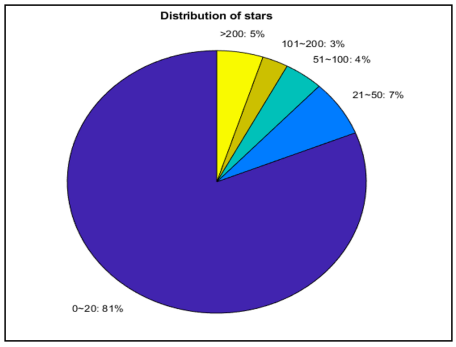

Figure 7: Distribution of stars

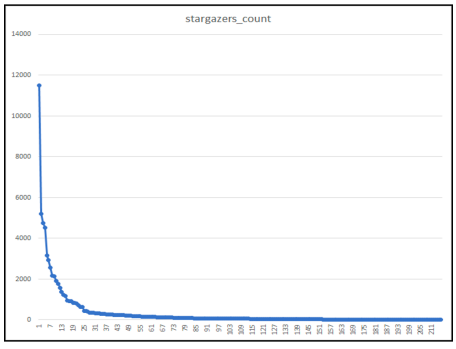

Figure 8: number of stars

The number of stars of an open source project in Github can reflect the degree of attention the project receives from the community. Figure. 7 shows the distribution of the number of stars for 1000 eBPFs.

From Fig. 7 and Fig. 8 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.

3.3 Detailed Analysis

We conducted a more detailed analysis of the top 50 applications in terms of number of stars on github, and here are the results and the theoretical explanations.

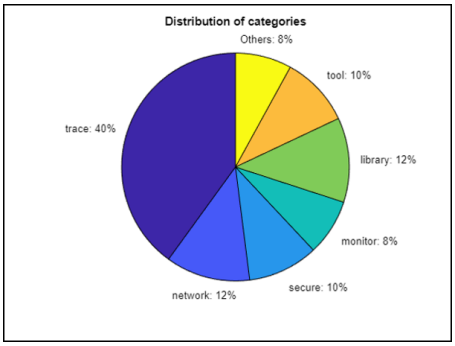

Figure 9: distribution of functions

Among all functional categories, Tracing has the largest share, followed by network (see Fig. 9).Then Network and eBPF libraries account for the same 12 percent. Security, monitoring, tools and other each account for about 10 percent. It can be seen that tracing is the most important feature for eBPF users and developers, and nearly half of the popular eBPF applications are extensions or enhancements of linux tracing, reflecting to some extent the need to enrich the default syscall or API in the linux kernel for tracing functionality. Surprisingly, learning materials for eBPF account for 10 percent of all repositories, which indicates that eBPF is receiving more and more attention from users and developers.

Repository issues let developers track their work on GitHub, where development happens. We also analyzed the open issue number of these top ranked eBPF applications. The number of open issues expresses the features that will be fixed or developed. More than 90 percent of repositories have fewer than 5 open issues, and only 2 percent of these repositories have more than 40 open issues. That illustrates most of these eBPF applications are well-programmed or lack-maintained.

from Fig. 10 we can see that among all 50 applications, only 8 percent has less than 500 lines of code, 35 percent had 2,000 to 10,000 lines of code, and nearly 50 percent of applications has more than 10,000 lines of code. This is a very different result from the analysis of lines of code done for 1000 applications previously in this paper. Since all the

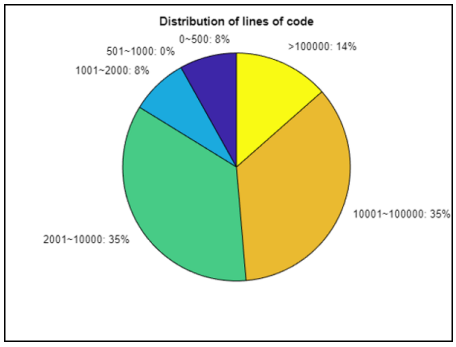

Figure 10: distribution of lines

software analyzed in this section have a high number of stars, it can be inferred that the number of stars an eBPF receives from the open source community is to some extent positively correlated with the number of lines of code of the software.

3.4 Tracing

Among the top 50 popular projects, we find that 40% of applications focus on tracing. And for tracing applications, most are based on bcc, and the rest are based on tracee or some self-constructed tools. Fig. 9 and fig. 11 show these results.

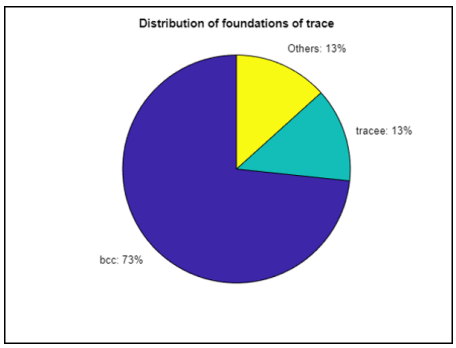

Figure 11: distribution of tools for trace

BCC [4] is a toolkit for creating efficient kernel tracing and manipulation programs built upon eBPF and includes several useful command-line tools and examples. It can attach eBPF programs (user-defined, sandboxed bytecode executed by the kernel) to kprobes. This allows user-defined instrumentation on a live kernel image that can never crash, hang or interfere with the kernel negatively. As we can see in fig. 12, bcc offers several tools to trace the information from different layers in linux. The left image indicates what bcc could trace in the linux system. As we can see, Thanks to the data source provided by linux, such as kprobes for events and map.lookup() for maps, bcc could attach them to support tracing different layers' information.

And tracee [3] is a linux runtime tool which is using Linux

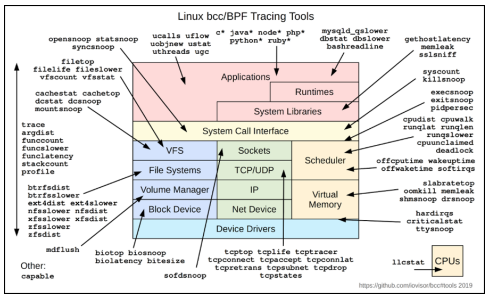

Figure 12: Linux bcc tracing tools

eBPF technology to trace your system and applications at runtime, and analyze collected events to detect suspicious behavioral patterns.

4 Related Work

Tracing performance has been studied [1]. The most popular use of BPF is performance tracing: instrumenting in various part of the kernel, including the network stack, disk I/O, CPU perfomance analysis along various metrics, memory, file systems. There are three main ways to do BPF programming: bpf_trace, BCC and LLVM. bpf_trace is the highest-level one, and is useful to write one liners quickly to solve a particular problem. BCC is a compiler allowing you to write C code and compile it into BPF programs. LLVM IR can also be used to write BPF, but is only really used by developers of BCC or bpf_trace.

Firewall capabilities of operating systems are traditionally provided by inflexible filter routines or hooks in the kernel. These require privileged access to be configured and are not easily extensible for custom low-level actions.Dominik Scholz et al. [5] analysed the performance of Packet Filtering with Linux eBPF. They analyzed the performance of the eXpress Data Path(XDP). XDP uses eBPF to process ingress traffic before the allocation of kernel data structures which comes along with performance benefits. In the second case study, eBPF is used to install application-specific packet filtering configurations acting on the socket level. They found out that eBPF can be used for versatile and high performance packet filtering applications. XDP, hooking at the lowest level before the network stack, is well suited for coarse packet filtering such as DoS prevention. XDP can yield four times the performance in comparison to performing a similar task in the kernel using common packet filtering tools. While latency outliers exist, which will likely be improved with increasing maturity of the XDP code base, the median latency also shows improvements. JIT compiled code yields up to 45% improved performance at the cost of more and higher latency outliers.

Network security is traditionally based on the analysis and dissection of network packets. The widespread use of data encryption and the increase of network traffic created many

challenges in terms of visibility and performance, making security tools less effective and both hard to deploy and maintain as network size and speed increase. The advent of eBPF in modern Linux systems enables introspection and adds the ability to inject code in the kernel at specific tracepoints. Luca Deri et al. [2] leverage eBPF to combine system introspection with a novel system-level security policer that enables the creation of fine grained security policies tailored for specific users, processes and containers. This is a major advance for network security applications that can benefit from system introspection to enrich information extracted from network packets, paving the way for the implementation of system and network aware security polices that combine visibility and security at a fraction of the computational cost of existing solutions.

5 Conclusion

We implemented a web crawler and downloaded 1000 open source eBPF tools and applications from Github. We analysed these codes and conducts some general features regarding distribution of lines and languages. We then analysed top 50 most popular projects. We found that tracing is the most common function among those 50 projects. And bcc is the tool that researchers prefer to leverage for tracing.

To go further, the distribution of hookers is an interesting topic to analyse. The Intrusion Analysis, such as bugs, vulnerabilities, local privilege escalations and other issues in the kernel caused by the introduction of eBPF would be an important feature to reflect the evolution of eBPF and recent Linux kernel versions.

6 Metadata

The presentation of the project can be found at:
https://docs.google.com/presentation/d/1uhrBGcPNbg-VwHn3xfEvVgodYwBsKU9V/edit?usp=sharing&ouid=104663309855861168411&rtpof=true&sd=true
The code/data of the project can be found at:
https://github.com/alexsssu/A-Study-of-Linux-eBPF_CS571

References

[1] Matthieu De Beule. eBPF for perfomance analysis and networking. June 2020.

[2] Luca Deri, Samuele Sabella, Simone Mainardi, Pierpaolo Degano, and Roberto Zunino. Combining system visibility and security using ebpf. In *ITASEC*, 2019.

[3] https://github.com/aquasecurity/tracee.

[4] https://github.com/iovisor/bcc.

[5] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 1, pages 209–217. IEEE, 2018.