Chapter 1. Introduction

Over the last few decades computing systems have only grown in complexity. Reasoning about how software behaves has created multiple business categories, all of them trying solve the challenges of gaining inshight into complex systems. One approach to get this visibility is by analyzing data generated by all applications running in a computing system as logs. Logs are a great source of information. They can give you precise data of how an application is behaving. However, they constrain you to know only the information that the engineers that built the application exposed in those logs. Gathering any additional information in log format from any system can be as challenging as decompiling the program and looking at the execution flow. Another popular approach is to use metrics to reason why a program behaves the way it does. Metrics differ from logs in the data format, while logs give you explicit data, metrics aggregate data to measure how a program behaves at an specific point in time.

Observability is an emergent practice that approach this problem from an different angle. People define Observability as the capacity that we have to ask arbitrary questions and receive complex answers from any given system. A key difference between Observability and log, and metrics, aggregation, is the data that you collect. Since by practicing observability you need to answer any arbitrary question at any point in time, the only way to reason about data is by collecting all data that your system can generate, and only aggregate it when it’s necessary to answer your questions.

Nassim Nicholas Taleb, the author of best-seller books like *Antifragile: Things That Gain From Disorder*, popularized the term *Black Swan* for unexpected events, with major consequences, that could have been expected if they had been observed before. In his book *The Black Swan*, he rationalizes how having relevant data could help in risk mitigation for these rare events. Black Swan events are more common that we think in software engineering, and they are inebitable. Since we can assume that we cannot prevent these kind of events, our only option is to have as much information as possbile about them to address them without impacting bussiness systems in a crytical way. Observability helps us build robust systems and mitigate future black swan events because it’s

based on the premise that you're collecting any data that can anwer any future question. The study of Black Swan events and practicing Observability converge in once central point, the data that you gather from your systems.

Linux Containers are an abstraction on top of a set of features on the Linux Kernel to isolate and manage computer processes. The kernel, traditionally in charge of resource management, also provides task isolation and security. In Linux, the main features that containers are based on are Namespaces and Cgroups. Namespaces are the components that isolate tasks from each other. In a sense, when you're inside a namespace, you experience the operating system like if there was no other tasks running in the computer. Cgroups are the component that provide resource management. From an operational point of view, they give you fine grained control to any resource usage, like CPU, disk I/O, network and so on. In the last decade, with the raise in popularity of Linux Containers, there have been a swift in the way software engineers architect large distributed systems and compute plaftforms. Multi-tenant computing has grown completely relyant of these features in the Kernel.

By relying so much on the low level capabilities of the Linux Kernel, we've tapped into a new source of complexity and information that we need consider when we design observable systems. The kernel is an evented system, which means that all work is described and executed based on events. Opening files are a kind of event, executing an arbitrary instruction by a CPU is an event, receiving a network packet is an event, and so on. Berkeley Packet Filter (BPF) is a subsystem in the kernel that can inspect those new sources of information. BPF allows you to write programs that are safely executed when the kernel triggers any event. BPF gives you strong safety guarantees to prevent you from injecting system crashes and malicious behavior in those programs. BPF is enabling a new wave of tools to help system developers observe and work with these new platforms.

In this book, we're going to show you the power that BPF offers you to make any computing system more observable. We're also going to show you how to write BPF programs with the help of multiple programming languages. We've put the code for your programs in GitHub, so you don't have to copy and paste the code. You can find it in a git repository companion to this book.

But before starting to focus on the technical aspects of BPF, let us tell you how everything began.

BPF's History

In 1992, Steven McCanne and Van Jacobson wrote the paper "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In this paper, the authors described how they implemented a network packet filter for the Unix kernel that was 20 times faster than the state of the art in packet filtering at the time. Packet filters have a very specific purpose, provide applications that monitor the system's network with direct information from the kernel. With this information, applications could decide what to do with those packets. BPF introduced two big innovations in packet filtering:

- A new virtual machine designed to work efficiently with register based CPUs.
- The usage of per application buffers that could filter packets without copying all the packet information. This minimized the amount of data BPF required to make decisions.

These drastic improvements made all Unix systems adopt BPF as the technology of choice for network packet filtering, abandoning old implementations that consumed more memory and were less performant. This implementation is still present in many derivatives of that Unix kernel, including the Linux kernel.

In early 2014, Alexei Starovoitov introduced the extended BPF implementation. This new design was optimized for modern hardware, making its resulting instruction set faster than the machine code generated by the old BPF interpreter. This extended version also increased the number of registers in the BPF virtual machine from two 32-bit registers to ten 64-bit registers. The increase in the number of registers, and their width, opened the possibility to write more complex programs, since developers were free to exchange more information using function parameters. These changes, among other improvements, made extended BPF up to 4 times faster than the original BPF implementation.

The initial goal for this new implementation was to optimize the internal BPF instruction set that processed network filters. At this point, BPF was still restricted to kernel space, and only a few programs in user-space could write BPF filters for the kernel to process, like Tcpdump and Seccomp, which we'll talk in future chapters. Today, these programs still generate bytecode for the old BPF interpreter, but the kernel translates those instructions to the much improved internal representation.

In June of 2014, the extended version of BPF was exposed to user-space. This was an inflection point for the future of BPF. As Alexei wrote in the patch that introduced these changes:

```
this patch set demonstrates the potential of eBPF.
```

BPF became a top level kernel subsystem, and it stopped being limited to the networking stack. BPF programs started to look more like kernel modules, with a big emphasis towards safety and stability. Unlike kernel modules, BPF programs don't require to recompile your kernel, and they are guaranteed to complete without crashing.

The BPF Verifier, which we'll talk about in the next chapter, added these required safety guarantees. It ensures that any BPF program will complete without crashing, and it ensures that programs don't try to access memory out of range. These advantages come with certain restrictions though, programs have a maximum size allowed, and loops need to be bounded, to ensure that the system's memory is never exhausted by a bad BPF program.

With the changes to make BPF accessible from user-space, the kernel developers also added a new system call, `bpf`. This new syscall will be the central piece of communication between user-space and the kernel. We'll talk about how to use this system call to work with BPF programs and maps in Chapters 2 and 3 of this book.

BPF maps will become the main mechanism to exchange data between the kernel and user-space. We'll see in Chapter 2 how to use these specialized structures to collect information from the kernel, as well as sending information to BPF programs that already running in the kernel.

The extended BPF version is the starting point for this book. In the last five years, BPF has evolved significantly since the introduction of this extended version, and we'll cover in detail the evolution of BPF programs, BPF maps, and kernel subsystems that have been affected by this evolution.

Architecture

BPF's architecture within the kernel is fastinating. We'll dive into its specific details through the whole book, but we want to give you a quick overview about how it works in this chapter.

As we mentioned earlier, BPF is a highly advanced virtual machine, running code instructions in an isolated environment. In a sense, you can think of BPF in a similar way you think about the Java Virtual Machine, a specialized program that runs machine code compiled from a high level programming language. Compilers like LLVM, and GCC in the near future, provide support for BPF, allowing you to compile C code into BPF instructions. Once your code is compiled, BPF uses a verifier to ensure that the program is safe to run by the kernel. It will prevent you from running code that might compromise your system by crashing the kernel. If your code is safe, the BPF program will be loaded in the kernel. The Linux kernel also incorporates a Just-In-Time(JIT) compiler for BPF instructions. The JIT will transform the BPF bytecode into machine code right after the program is verified, avoiding this overhead on execution time. One interesting aspect of this architecture is that you don't need to restart your system to load BPF programs, you can load them on demand, and you can also write your own init scripts that load BPF programs when your system starts.

Before the kernel runs any BPF program, it needs to know which execution point the program is attached to. There are multiple attachment points in the kernel, and the list is growing. The execution points are defined by the BPF program types, we'll talk about them in the next chapter. When you choose an execution point, the kernel also makes available specific function helpers that you can use to work with the data that your program receives, making execution points and BPF programs tightly coupled.

The final component in BPF's architecture is responsible for sharing data between the kernel and user-space. This component is called BPF Maps, and we'll talk about them in Chapter 3. BPF Maps are bi-directional structures to share data. This means that you can write and read on them from both sides, the kernel and user-space. There are several types of structures, from simple arrays and hash maps, to very specialized maps that allow you to save entire BPF programs in them.

We'll cover every component in BPF's architecture in more details as the book progresses. You'll also learn to take advantage of BPF's extensibility and data sharing with specific examples covering from stack trace analysis to network filtering and runtime isolation.

Conclusion

We've wrote this book to help you get familar with the basic BPF concepts that you're going to need in your day to day when working with this Linux subsystem. BPF is still a technology in development, new concepts and paradigms are growing as we write this book. Hopefully, this book will help you expand your knowledge easily by giving you a solid base of its fundational components.

The next chapter dives directly into the structure of BPF programs and how the kernel runs them. It also covers the points in the kernel where you can attach those programs. This will help you get familiar with all the data that your programs can consume and how to use it.