UNIVERSITÀ DEGLI STUDI DI BERGAMO

Department of Ingegneria Gestionale, dell'Informazione e della Produzione

Master Degree in Ingegneria Informatica

Class LM-32

Exploring eBPF for Windows

Implementation analysis and comparison with Linux

Advisor

Chiar.mo Prof. Stefano Paraboschi

Master Thesis

Matteo Locatelli

Student number 1059210

ACADEMIC YEAR 2022/2023

Abstract

Berkeley Packet Filter (BPF), an originally Unix-based packet filtering technology, has evolved into a versatile tool with significant impact on network performance and security. This master thesis aims to explore the story of BPF, tracing its development from its inception on Unix-based systems to its adaptation on Windows platforms. Through a comparative analysis, we investigate the challenges, solutions and advancements that have led to the successful integration of BPF in the Windows environment. By studying its history, architecture and programs development, we explore the potential of BPF to revolutionize network engineering on Windows and contribute to the broader understanding of cross-platform technology adoption.

Acknowledgements

Completing this master thesis on the evolution and adaptation of Berkeley Packet Filter on both Linux and Windows platforms has been an enriching journey for me. I am deeply grateful to the individuals whose guidance, encouragement and support have made this research possible. Without their firm belief in my abilities, this effort would not have come to completion.

First and foremost, I extend my heartfelt gratitude to my esteemed advisor, professor Stefano Paraboschi, whose expertise, mentorship and invaluable feedback have been instrumental in shaping this thesis.

My sincere appreciation must be extended to the people in the *Unibg Security Lab* team who actively participated in the development of this thesis: their continuous support throughout the entire research process have motivated me to push my boundaries and aim for excellence. I am grateful for their patience, insightful discussions and profound knowledge in the fields of computer engineering and systems security, which have significantly contributed to the depth and quality of this work: their willingness to share their expertise has been essential in overcoming various challenges faced during this study and in refining the ideas presented in this research. Their support has made this academic pursuit not only a productive venture, but also an enjoyable one. Also, they provided me with the LaTex template that I used to write this thesis.

Speaking of people that gave me something practical that helped me to work on this project, I have to thank Subconscious Compute, an Indian IT company founded in 2020 that works on security for distributed devices and data. Their decision to open-source their GitHub repository, which is under AGPL license, and grant me access to it has been fundamental in enabling me to develop eBPF programs on the Windows platform and to do a better comparison with eBPF on Linux, which was the scope of my master's thesis. The availability of the repository not only provided me with a lot of resources and code examples, but also allowed me to gain insights into best practices and advanced techniques in programming for Windows using eBPF.

Moreover, I would like to express my obligation to the wider academic community of the *Università degli Studi di Bergamo* for providing an environment that encourages learning, curiosity and innovation. I am deeply grateful to everyone who played a part, big or small, in the ending of my academic journey. The education I received has been invaluable and I am fortunate to have had such exceptional guidance throughout my academic period. This thesis stands as a testament to the collective effort and support of those who have been part of my academic journey. The knowledge and experiences I have gained throughout the last five years have been instrumental in shaping my growth as a computer engineering student.

Last but not least, I must express my very profound gratitude to my parents for their love, encouragement and support throughout eighteen years of education. Their belief in my capabilities and constant motivation have been the driving force of my academic achievements, especially during the demanding period of writing this thesis. I owe my successes to them because with their sacrifices they have allowed me to focus on my studies and achieve my academic goals, celebrating every milestone with infinite joy and pride. In conclusion, I am grateful for the life lessons and values they instilled in me, which have shaped me into the person I am today.

Thank you.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In the ever-evolving landscape of computer science and networking, the demand for efficient, flexible and secure packet filtering technologies has been dominant. The Berkeley Packet Filter (BPF), an innovative technology developed in the Unix environment, has emerged as a powerful tool for network monitoring, traffic analysis and security enforcement. Over the years, BPF has undergone significant advancements, culminating in the birth of Extended Berkeley Packet Filter (eBPF), a groundbreaking extension that has revolutionized network engineering and performance analysis.

1.1 Background

Computer networks establish the backbone of modern communication, enabling the seamless exchange of information across the globe.

The rapid growth of network traffic, the rise of complex cyber threats and the increasing need for real-time monitoring have motivated researchers and engineers to explore innovative solutions to enhance network performance and to build robust security mechanisms. Packet filtering, a fundamental networking technique, serves as a first line of defense in safeguarding networks and optimizing data transmission.

Originally conceived in the 1990s, the Berkeley Packet Filter (BPF) was designed as a mechanism to filter packets at the kernel level for the Berkeley Software Distribution (BSD) operating system (a discontinued operating system based on the early versions of the Unix operating system). However, its potential, consisting of its lightweightand versatile design, far exceeded its initial purpose and it evolved into a versatile technology with applications across various networking domains.

Over the years, BPF has undergone significant developments and adaptations, until it resulted in the advent of eBPF: with the introduction of a new virtual machine and bytecode, eBPF allowed for the dynamic execution of custom programs within the kernel context, extending its applicability beyond traditional packet filtering to areas such as network monitoring, tracing and deep packet inspection.

1.2 Motivation

Despite the extensive use of eBPF in Unix-based systems, its incorporation into Windows environments has remained a challenge. As Windows continues to be a prominent operating system in both personal and enterprise computing, unlocking the potential of eBPF on this platform becomes crucial for achieving cross-platform network engineering and security solutions.

This thesis will focus on the historical progression of BPF and its adaptation on the Windows platform. In addition to that, we will explore the advancements introduced to eBPF on both operative systems and study the current state of art of eBPF on Windows to show its differences with the Linux environment.

1.3 Objectives

This master's thesis aims to provide an in-depth analysis of eBPF's architecture, installation and functionalities in both operating systems, while showing the history, development and impact of eBPF in the world of computer science and network engineering.

The primary objectives of this research are as follows:

- Tell the history of eBPF: by understanding the origins of BPF, we gain insights into the motivations that led to the creation of eBPF and we can identify the key challenges faced during its integration into Windows and the innovative solutions designed to overcome them. A look into the historical context provides a solid foundation for exploring eBPF's potential, from a simple packet filtering mechanism to a versatile technology with broader network real-world applications;

- Installation and integration of eBPF on Linux and Windows: we will investigate the process of installing eBPF into both Linux and Windows operating systems. By understanding the differences in installation procedure and requirements on these platforms, we are enabled to leverage the cross-platform capabilities of this technology;

- Development of eBPF programs on Linux and Windows: this thesis will cover the development process of eBPF programs on both Linux and Windows platforms. We will explore the process of creating, loading and executing eBPF programs. Furthermore, by studying the eBPF API, we will:

  - Demonstrate the creation of custom programs to achieve specific networking tasks;
  - Show how far they have come in the development of the technology in the two operating systems;
  - Examine the methods used to safely load eBPF programs into the kernel.

1.4 Organization of the Thesis

BETTER ORGANIZATION -> TEXT OR LIST

The subsequent chapters of this thesis will be organized as follows:

- Chapter 2: Technologies used for working with eBPF;
- Chapter 3: the history of eBPF (with real-world examples);
- Chapter 4: How eBPF works
- Chapter 5: Applications ad infrastructure of eBPF (BCC, libbpf,... from ebpf.io)

- Chapter 6: eBPF on Linux (installation and programs development);
- Chapter 7: eBPF on Windows (installation and programs development);
- Chapter 8: Future prospects of eBPF on Windows;
- Chapter 9: Conclusion.

Through this master's thesis, we hope to offer a comprehensive understanding of eBPF's significance, capabilities and potential in modern networking environments. We also have the ambition to contribute to the field of computer engineering by closing the gap between Unix and Windows-based network technologies and security measures. By exploring the installation and development processes on both Linux and Windows, we present a comparative analysis of eBPF's cross-platform capabilities.

Chapter 2

Technologies used

Since we already announced that we are going to work on both Linux and Windows, before diving into the installation process of eBPF on both Linux and Windows, it is important to describe the technologies that allowed us to develop programs using eBPF.

2.1 The host environment

The project started with a single Windows 11 PC serving as the host environment for all research and development activities.

The computer has a 64 bit operating system with a processor based on x64, a 16 GB RAM and a Solid-State Drive (SSD) with a capacity of 1TB as for storage. Windows 11, with its user-friendly interface and vast software ecosystem, combined with the power given by the four cores of the Intel Core i7 processor, provided an efficient platform for general computing requirements.

Given the fact that other operating systems were required for this project, the integration of virtualization was crucial to create isolated environments alongside the Windows host.

2.2 Virtual machine for Linux development

For installing and developing programs with eBPF on Linux, a virtual machine running Ubuntu 22.04 was set up within VirtualBox (the version of the Ubuntu operating system is not important).

VirtualBox is a type 2 or hosted hypervisor suitable for individual use and small-scale virtualization scenarios. It is a software application that runs on top of an existing operating system (called host OS) and provides the capability to create and manage virtual machines. Figure 2.1 shows a schematic representation of the architecture just described. VirtualBox allows you to test, develop and run multiple guest operating systems within your host operating system simultaneously, providing a good level of isolation between the host and guest operating systems. As a type 2 hypervisor, VirtualBox relies on the host operating system's kernel to manage hardware resources: it uses device drivers and services from the host OS to interact with the physical hardware, which can introduce some overhead and may affect performance compared to a type 1 hypervisor.

Even though VirtalBox relies on the host OS for certain operations, which can lead to performance differences and potential resource conflicts, it was chosen over a type 1 hypervisor for its user-friendly virtualization solution.

The installation process involved creating a virtual disk, configuring memory and CPU allocation and selecting the Ubuntu 22.04 ISO file previously downloaded for installation [6]. The virtual machine provided a native Linux platform for eBPF program development, compilation and testing.

2.3 Virtual machine for Windows development

Since the main focus is the analysis of eBPF state of art on Windows, the project also demanded the capability to develop eBPF programs specific to the Windows platform. For this purpose, the Hyper-V Console Manager, a native Windows feature, was used to create a separate Windows 11 virtual machine.

Hyper-V is a type 1 or bare-metal virtualization software, also known as a Virtual Machine Monitor (VMM), which runs directly on the physical hardware without

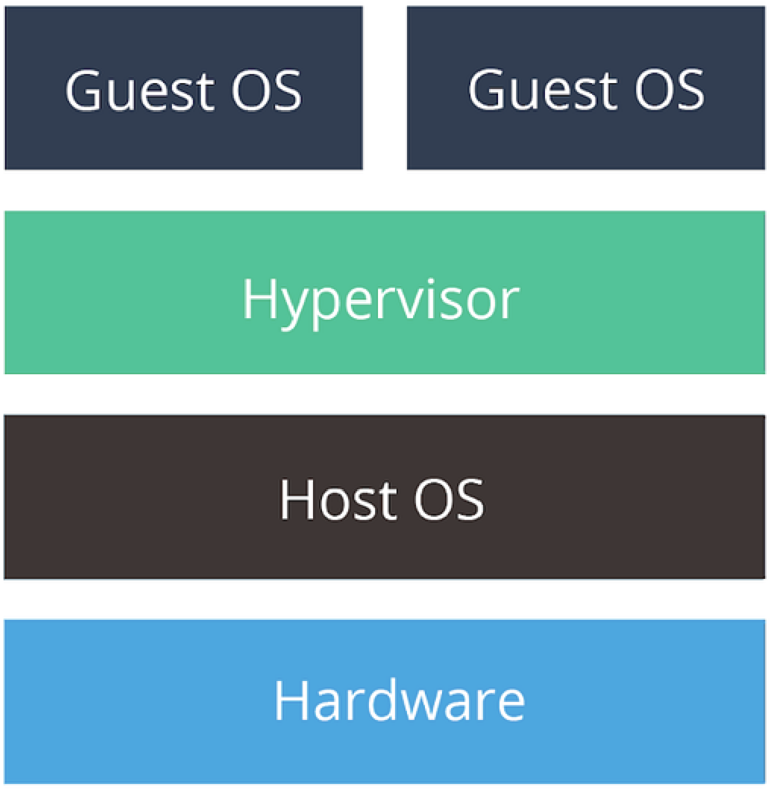

Figure 2.1: Type 2 (or hosted) hypervisor architecture [3].

the need for an underlying host operating system. The illustrative representation of the architecture just described is depicted in Figure 2.2. As the core software responsible for managing virtual machines and allocating hardware resources to each VM, Hyper-V ensures better security and resource utilization by isolating each VM from others and the host OS. With direct access to the physical hardware, it efficiently allocates resources, resulting in improved performance, isolation and scalability compared to type 2 hypervisors like VirtualBox.

Even though hypervisors are favored for their robustness and scalability, enabling the efficient virtualization of large-scale applications and services, the choice of creating a virtual machine using the Hyper-V Console Manager was dictated by the setup instructions described on the *ebpf-for-windows* GitHub repository [7].

We are going to talk about the eBPF installation on Windows later: for now, besides the fact that that he virtual machine was configured with adequate resources to support development tasks effectively, the only thing worth noting is that during the quick creation of the virtual machine the option of “Windows 11 dev environment”

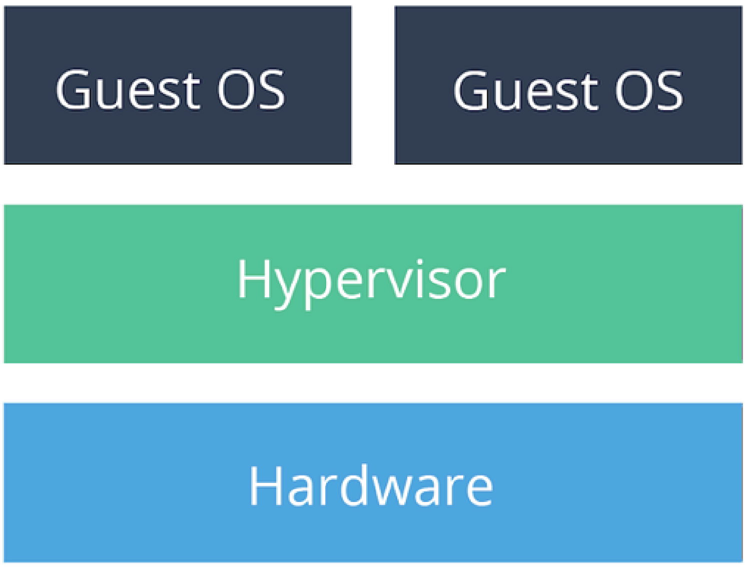

Figure 2.2: Type 1 (or bare metal) hypervisor architecture [3].

must be selected (the tutorial tells to choose the "Windows 10 dev environment", but Windows 11 works as well).

The so created isolated Windows 11 development environment provided a controlled space for testing and optimizing eBPF programs on the Windows platform.

2.4 Repository of the project

GitHub is a platform and cloud-based service for software development and collaborative version control using Git, a distributed version control system that tracks changes in any set of computer files, allowing developers to store and manage their code, owned by the company GitHub Inc., whose logo is displayed in Figure 2.3. It provides the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration and wikis for every project. It is commonly used to host open source software development projects.

Throughout the course of my master's thesis on eBPF, GitHub was an indispensable platform that played a dual role in enhancing my research journey.

Firstly, it served as an efficient instrument to share the progress of my work with my co-advisors and make easier the collaboration during the entire development process. Its version control system allowed me to keep track of changes, maintain

Figure 2.3: GitHub *Invertocat* logo [2].

a detailed history of my project and collaborate consistently with my co-advisors, ensuring a smooth and efficient development workflow. By regularly pushing updates to the repository [4], my co-advisors were able to monitor the evolution of my work, review code changes, provide timely feedback and offer valuable suggestions for improvement.

Secondly, GitHub served as an invaluable resource for the eBPF community: during my research, I encountered several repositories (which we will discuss later) dedicated to developing and optimizing eBPF environments, tools and libraries. By studying and understanding their implementations, I was able to build upon the expertise and contributions of the open-source community, so that the quality and scope of my research have been enriched.

The open-source spirit of GitHub made knowledge exchange and collective growth easier, enabling me to contribute to the eBPF community while benefiting from the collective expertise it had to offer. In fact, the public visibility of the GitHub repository of this project opens up the possibility of sharing my work with the wider community. By making the repository public, we hope that others can benefit from the knowledge and insights gained during the project, encouraging collaboration and contributions from future researchers and developers in the field of eBPF and its applications.

CITARE TEXSTUDIO E LATEX

Chapter 3

The history of eBPF

This chapter digs in the historical journey of extended Berkeley Packet Filter (eBPF), starting from the first ideas of packet filtering to its current state as a powerful and versatile technology. By exploring the foundations of packet filtering and the development of traditional BPF, we lay the groundwork for understanding the motivations behind eBPF's emergence. We will uncover how eBPF has revolutionized networking, observability and security in contemporary computing environments, from its initial applications in Unix-based systems to its widespread adoption in modern computing.

3.1 The beginning of packet filtering

The acronym BPF was first used in December 1992 in a document written by Steven McCanne and Van Jacobson while at Lawrence Berkeley Laboratory (Berkeley, California), titled *The BSD Packet Filter: a New Architecture for User-level Packet Capture* [5]. Fun fact, at the beginning of its story, the *B* in BPF standed for Berkeley Software Distribution (BSD), a discontinued operating system based on the early version of Unix, which was developed and distributed by the Computer Systems Research Group at the University of California in Berkeley.

In this article they talk about the packet-capture techniques existing at the time and they describe the BSD packet filter (BPF), "a new kernel architecture for packet capture". The authors first start to describe the need to manage network traffic efficiently and how it was performed with the facilities implemented to those days. Then, they present the design of BPF, showing its model and a pseudo-machine that would work as a filter with BPF. In the end, they do some examples of packet filtering with BPF and with other technologies and compare their performances on the same hardware, showing how and why BPF performs substantially better than other approaches.

3.2 The characteristics of BPF

While the previous article was the first to cover BPF, it offers a broad view of the improvements this technology would bring to the world of network monitoring:

- It outperforms other facilities of that time in their filtering mechanisms;
- It has a programmable pseudo-machine model that demonstrated to be general and extensible;
- It is portable and runs on most BSD systems which, due to their Unix-like basis, were a synonym of high quality networking back then;
- It can interact with various data-link layers.

Given these characteristics, it can be understood how BPF was ahead of its time: it was used to speed up packet filtering and analyze network traffic, since packets rates could be very high, even for the computers at the time when McCanne and Jacobson wrote their article. In fact, the original BPF was designed for capturing and filtering network packets that matched specific rules: to do so, a user-space process was allowed to supply a filter program that specifies which packets it wants to receive. This program would be run on a register-based virtual machine inside the kernel, once they were compiled in the most efficient instructions.

The fact that BPF worked in a way similar to a virtual machine in the kernel was the most interesting part about this new technology: the filter programs were in the form of instructions for a virtual machine, which were interpreted or compiled into machine code by a just-in-time (JIT, the process of compiling a program during its execution) mechanism and executed in the kernel.

The features of this virtual machine are described in the document mentioned above: it was a 32-bit machine with fixed-length instructions, one accumulator and one index register. Programs in that language could perform different types of operations, like fetching data from the packet, performing arithmetic operations on data from the packet and comparing the results against constants or against data in the packet or test bits in the results, accepting or rejecting the packet based on the results of those tests.

How can traditional Unix-like BPF implementations be used in user-space, despite being written for kernel-space? This is accomplished using preprocessor conditions. A preprocessor is a program that receives an input and produces an output that it will be used as an input for an other program. This is a typical features of compilers, computer programs that translate computer code written in one programming language (the source language) into another language (the target language). This name is primarily used for programs that translate source code from a high-level programming language to a low-level programming language (e.g. assembly language, object code, or machine code) to create an executable program. We brought the example of compilers because we are going to see later that the process of loading a BPF program inside the kernel requires, among many things, a compiler.

An other interesting feature about BPF was the fact that it provided a raw interface to various data-link layers, permitting layer 2 packets to be sent and received. Sometimes, BPF is used specifically in reference to its filtering capabilities, rather than encompassing the entire interface. Across various systems, like Linux, other raw interfaces to the data link layer exist and they utilize BPF's filtering mechanisms for their own purposes. In fact, the BPF filtering mechanism is not only available on a select few systems, but is extensively found in most Unix-like operating systems and, in recent times, in the latest versions of Windows.

3.3 Limitations of BPF

The decision to run user-supplied programs within the kernel proved to be highly advantageous, but certain aspects of the original BPF design faced difficult challenges over time:

- The virtual machine and its fixed-length instruction set architecture (ISA, a part of the abstract model of a computer that defines how the CPU is controlled by the software) were outpaced as modern processors transitioned to 64-bit registers and introduced new instructions for multiprocessor systems, such as the atomic exchange-and-add instruction (XADD),compromising its ability to efficiently handle complex tasks on contemporary hardware;

- The initial focus on offering a limited number of Reduced Instruction Set Computer (RISC, a computer architecture designed to simplify the individual instructions given to the computer to accomplish tasks in order to achieve higher performances) instructions no longer aligned with the demands of contemporary processors because it did not provide a sufficient instruction set to handle advanced filtering and analysis task effectively;

- As new networking functionalities emerge, incorporating them into the traditional BPF framework became challenging, because it lacked robust mechanisms for extensions and overloading of instructions, making very difficult its adaptability to ever-evolving network architectures;

- Since BPF was primarily designed for execution within the kernel space, its use in certain user-space scenarios and in other potential applications was limited due to its lack of versatility;

- As modern networks handle higher data rates and voluminous traffic, processing and filtering massive amounts of packets in real-time with BPF could cause performance bottlenecks, impacting overall system responsiveness, because it might not scale efficiently;

- BPF was missing built-in safety mechanisms, making it vulnerable to errors or malicious code which could lead to system crashes or security breaches;
- BPF was not designed to handle efficiently complex packet structures or protocols, limiting its ability in analyzing and filtering non-standard or highly intricate network traffic;
- As networking technologies continue to evolve rapidly, BPF's rigidity may create challenges in adapting to emerging protocols, data formats and network architectures, potentially making it less suitable for future innovations.

It is essential to consider these limitations when evaluating the appropriateness of BPF for modern networking requirements. In fact, all of the problems about BPF described above can be referred to the fact that in the IT world things evolve really quickly and at its beginning BPF was not flexible and extensible to the innovations that would be introduced in the years to come.

Recognizing its historical significance and contributions, it is clear that BPF was not enough to keep up with the technological advancements that would be done in modern hardware. To try to address many of the described limitations, in 2014 Extended BPF (eBPF), a more versatile and future-ready technology for advanced networking and observability needs, was introduced by Alexei Starovoitov and Daniel Borkmann, creators and current maintainers of this project.

3.4 Introduction to eBPF

eBPF is a technology that can run sandboxed programs in a privileged context such as the operating system kernel. It has first appeared in the Linux Kernel 3.18 after the extension of the inner BPF virtual machine and makes the original version, which has been retroactively renamed to *classic* BPF (cBPF), mostly obsolete. In Table 3.1 we can see the main differences that were brought with the introduction of eBPF.

Moving to 64-bit registers and an increasing the number of registers from two to ten (since modern architectures have far more than two registers), allowed parame-

	Classic BPF	Extended BPF
Word size	32-bit	64-bit
Registers	2	10+1
Storage	16 slots	512 byte stack + infinite map storage
Events	packets	many event sources

Table 3.1: Comparison between cBPF and eBPF main features.

ters to be passed to functions in eBPF virtual machine registers just like on native hardware and virtually gave the virtual machine unlimited storage. While these changes were introduced in eBPF due to of progresses made in computer hardware, there have also been several revolutions regarding the technology itself:

- The most important one is the fact that an eBPF program, instead of only being attached to packets, it can now be attached to many different event sources and run many programs within the kernel, making this technology very powerful and allowing it to start being used in a wide variety of applications, including networking and tracing;

- At the lowest level, beyond the use of ten 64-bit registers, eBPF introduced different jump semantics, a new *BPF_CALL* instruction to call in-kernel functions cheaply and corresponding register passing convention, new instructions and a different encoding for these instructions;

- The ease of mapping eBPF to native instructions made it suitable for JIT compilation, which was supported by many architectures, bringing an improvement in the performance ("The original patch that added support for eBPF in the 3.15 kernel showed that eBPF was up to four times faster on x86-64 than the old cBPF implementation for some network filter microbenchmarks, and most were 1.5 times faster" [1]);

- eBPF was made more flexible and as the Linux Kernel evolved in versions after 3.18, new functionalities (that we will discuss later) were subsequently added, such as the use of loops.

The described changes made eBPF appear to the world as a revolution. Originally, eBPF was only used internally by the kernel and loaded cBPF bytecode was transparently translated into an eBPF representation in the kernel before program execution. Finally, in 2014 the eBPF virtual machine was exposed directly to user space and nowadays the Linux kernel runs eBPF only. Moreover, in 2021, due to its success in Linux, the eBPF runtime has been ported to other operating systems such as Windows. cBPF, instead, passed to history as being the packet filter language used by *tcpdump*, a data-network packet analyzer computer program that runs under a command line interface and allows the user to display TCP/IP and other packets being transmitted or received over a network to which the computer is attached. Funny enough, tcpdump is free software written in 1988 by a team of people including Van Jacobson and Steven McCanne who were, at the time, working in the Lawrence Berkeley Laboratory. We could say that these two people were the parents of eBPF and conceived and developed it in the same house for over thirty years.

3.5 What is eBPF?

Even though the name Extended Berkeley Packet Filter hints at a packet filtering specific purpose, the instruction set was made generic and flexible enough that nowadays there are many use cases for eBPF apart from networking. In fact, eBPF is a highly flexible and efficient virtual machine-like construct in the Linux kernel allowing to execute bytecode at various hook points in a safe manner: it is used in a number of Linux kernel subsystems, most prominently networking, tracing and security (e.g. sandboxing).

The mind-blowing feature about eBPF is the fact that, at its core, it allows a user (in some cases privileged) to inject near general-purpose code in the kernel. Such code will then be executed at some point in time, usually after certain events of interest happen in the kernel. In theory, this sounds really similar to Loadable Kernel Modules (LKM), the traditional way with which users could extend the features of the kernel. In fact, LKM consist of a compiled general purpose C code loaded at run time inside the kernel and the code of a kernel module usually hooks into various kernel subsystems so that it gets automatically called upon the occurrence of certain events. This has been useful for developers who want to implement support for new hardware devices or tracing functions, for example. Even though both approaches want to extend the capabilities of the kernel at runtime, the big difference between them is the fact that, unlike LKM, eBPF will only run code that has been evaluated completely safe to run. This means that it will never lead to a kernel crash or kernel instability, which is something currently difficult to achieve with other technologies without giving up some serious flexibility. We could say that eBPF does the same job as LKM, but it does not require to change kernel source code or load kernel modules and does it in a safe and efficient manner.

How could this safety be achieved? It is provided through an in-kernel verifier which performs static code analysis and rejects programs which crash, hang or otherwise interfere with the kernel negatively (e.g. programs without strong exit guarantees like loops without exiting conditions, programs dereferencing pointers without safety-checks, ...). Programs that pass the verifier are loaded in the kernel where they will be either interpreted or JIT compiled for native execution performance. Once again, the interpreted or compiled eBPF program is verified before running to prevent denial-of-service attacks. Due to the fact that the execution model is event-driven, programs can be attached to various hook points in the operating system kernel and are run upon triggering of a specific event.

3.6 eBPF in modern architecture

3.6.1 Name and logo

Nowadays, BPF is a technology name and no longer just an acronym because, even though it evolved from BPF as an extended version, its use case outgrew networking. Some people still call it eBPF to really make the point that it's new: however kernel engineers tend to stick to BPF, meaning a generic internal execution environment for running programs in the kernel. Consistently with the research aim of this thesis, we are going to distinguish eBPF from cBPF to make more clear what we are referring to, even if from this point on we are going to talk exclusively about eBPF.

eBPF was also provided with an official logo: at the first eBPF Summit there was a vote taken and they decided to use the bee. So, Vadim Shchekoldin created what is named “eBee”, which we can see in Figure 3.1.

Figure 3.1: eBPF logo.

3.6.2 eBPF Foundation

Since its introduction in the infrastructure software world, the number of eBPF-based projects has exploded in recent years and more and more companies announced their intent to start adopting this technology. As such, there was the need to collaborate between projects to ensure that the core of eBPF would be well maintained and equipped with a clear path and vision for the bright future ahead of eBPF.

To respond to this demanding need, in August 2021, some companies, including Meta, Google, Microsoft, Isovalent and Netflix, founded the “eBPF Foundation”, establishing an eBPF steering committee (BSC) to take care of the technical direction and vision of eBPF. As one might expect, among the few members of the committee, there are Alexei Starovoitov and Daniel Borkmann. The logo of this institution can be seen in Figure 3.2

The purposes of this foundation were various and numerous:

- Expand the contributions being made to extend the powerful capabilities of

Figure 3.2: eBPF Foundation logo.

eBPF and grow beyond Linux (as we already mentioned before, eBPF is now also available on Windows);

- Raise funds in support of various open source, open data and/or open standards projects relating to eBPF technologies to further drive the growth and adoption of the eBPF ecosystem;

- Defining the minimal requirements of eBPF runtimes and maintain eBPF technical project lifecycle procedures to ensure a smooth and efficient progress of eBPF initiatives;

- Create a strong community that would collaborate among projects, attend technical workshops and conferences to discuss ongoing research, development efforts and use cases around eBPF, so that as many people as possible are involved in the project.

3.6.3 Use cases of eBPF

We understood that eBPF programs are verified within the kernel to avoid various risks: therefore eBPF programs pose less risk compared to an arbitrary loadable LKM and they also impose less overhead for many observation tasks compared to related tools. For this reasons, throughout the years, many more companies have joined this project and stated using eBPF. Nowadays, eBPF has been adopted by a number of large-scale production users, like Google, Meta, Netflix, Apple, Android, Microsoft,... mostly for network observability, security enforcement and layer 4 (in the ISO/OSI model) load balancing.

However, due to the fact that eBPF is very versatile, performing and programmable, people have found innovative solutions in various areas:

- Thanks to the networking and security revolution, eBPF allows administrators to create custom filters and access controls at the kernel level, offering powerful packet filtering and firewall capabilities while minimizing performance overhead (firewalls, intrusion detection systems and DDoS protection.);
- Given eBPF's real-time observability capabilities, achieved by attaching programs to kernel hooks, enable developers to gain deep insights into system calls, network activity and resource utilization, empowering efficient monitoring with low-latency and non-intrusive measurements in the dynamic environment of many systems and applications;
- In containerized environments, eBPF emerges as a game-changer, allowing administrators to efficiently control and optimize network traffic between containers, improving isolation, security and performance while, thanks to its programmability, consistently aligning with the dynamic nature of container orchestration platforms, like Kubernetes;
- In the middle of the evolving cloud landscape, eBPF assumes a central role, enabling efficient load balancing, traffic shaping and service discovery within the cloud infrastructure, ensuring optimal resource utilization and networking agility;
- Developers are enabled to look into application behavior and system performance through event capture and analysis at the kernel level using tracing tools that serve as instrumental support for diagnosing performance issues and debugging complex systems;
- eBPF is also used for real-time protection against malicious network activities due to the fact that it allows Intrusion Prevention Systems (IPS) to quickly inspect and filter packets, enabling rapid threat detection and prevention, while applying custom security policies and filtering rules;

- To reduce latency and increase efficiency for critical networking functions, eBPF uses custom in-kernel processing, efficiently offloading specific tasks to eBPF programs.

The modern use of eBPF continues to expand, as developers and organizations explore its capabilities and integrate it into various innovative applications. With its ever-growing ecosystem of tools, libraries and frameworks, eBPF is at the vanguard of driving efficiency, security and observability in contemporary computing environments.

3.7 Future and potential of eBPF

During our discussion, we mentioned the fact that eBPF tools surround functionalities in both kernel and user space, which aim at providing stable interfaces, such as kernel and user space tracepoints. However, it's essential to note that eBPF tools can also refer to functionality like functions or field names that may lack stability. For this reason, eBPF programs may not be portable across different kernels. Furthermore, certain older kernels might not incorporate the required functionality and some kernels may lack the necessary configuration to support eBPF. As a result, it becomes evident that BPF cannot be universally considered portable or universally available. In fact, even if eBPF is now supported on multiple platforms, as the beginning of 2023 there is no standard specification to formally define its components.

However, the world of eBPF evolves quickly and distributions appear to regularly support BPF and provide a package of BPF tools for easy installation. Furthermore, there is currently some work in progress to define and publish a standard for the instruction set, under the auspices of the eBPF Foundation.

So, for now, if you are running a recent version of the kernel and you can invoke eBPF as a privileged user, you should have eBPF functionality available. But if some eBPF tools do not work with your kernel, do not get disappointed: there are many people that are joining forces to make BPF programs more portable. For example, one of the natural challenges for tools that use kernel data structures (like eBPF) is that the offsets for fields can vary based on kernel version and configuration. We will see later how this problem has been solved.

Despite the need to standardize the technology and integrate it into as many platforms as possible, making it more accessible to developers and organizations worldwide, the future of eBPF is bright, due to its potential to twist the world of modern computing. This innovative technology is ready to unlock new frontiers and revolutionize various domains thanks to some key aspects:

- As network requirements keep evolving, with the help of eBPF's programmability, administrators have to implement newer custom network protocols, load balancing algorithms and traffic shaping mechanisms;
- As cloud adoption continues to rise, eBPF's indispensability in the cloud-native ecosystem will grow further, offering fine-grained control over container networking for optimal isolation, advanced security and efficient resource utilization, making agility and scalability essential for modern cloud infrastructure;
- The future of debugging and optimization for complex and distributed systems belongs to eBPF's real-time observability and tracing capabilities which allow developers to exploit its potential for capturing, analyzing and visualizing diverse system events with the purpose of providing unparalleled insights into application behavior, performance bottlenecks and resource utilization;
- As cyber threats become increasingly sophisticated, eBPF will persist in strengthening security measures, expanding its role in intrusion detection systems and security applications, providing real-time packet inspection, protocol analysis and advanced filtering capabilities;
- In a world where Artificial Intelligence and machine learning are increasingly in the spotlight, eBPF's programmability prepares to integrate them within the kernel, promoting a powerful synergy that drives intelligent decision-making, automated resource management and dynamic adaptation to satisfy shifting workloads and network conditions;

- With the help of the thriving eBPF community, new tools, libraries and frameworks are developed rapidly, pushing the boundaries of eBPF's potential and encouraging the creation of innovative solutions.

- As the world of Internet of Things and edge computing expands, eBPF's lightweight and efficient nature makes it an ideal match for devices with limited resources, finding applications in intelligent edge gateways for data filtering, analysis and real-time decision-making;

The future and potential of eBPF are full with possibilities. As it evolves, eBPF is set to reshape networking, observability and security paradigms, enabling developers to build efficient, secure and adaptable systems in the always evolving world of computing. With its impact and large adoption, eBPF is ready to become a landmark of next-generation software-defined infrastructures and beyond.

ROM THIS
OINT ON

3.8 eBPF in the world

eBPF use cases include (but are not limited to) networking such as XDP, tracing and security subsystems. Given eBPF's efficiency and flexibility opened up new possibilities to solve production issues, Brendan Gregg famously coined eBPF as "superpowers for Linux". Linus Torvalds expressed that "BPF has actually been really useful, and the real power of it is how it allows people to do specialized code that isn't enabled until asked for". Due to its success in Linux, the eBPF runtime has been ported to other operating systems such as Windows.

Chapter 4

How eBPF works

4.1 Writing an eBPF program

4.2 Architecture

4.3 Tools

4.4 Compiling and loading an eBPF program

4.5 The bpf() system call

4.6 Helpers

4.7 Maps

Chapter 5

Applications and infrastructure of eBPF

Bibliography

[1] *A thorough introduction to eBPF.* URL: https://lwn.net/Articles/740157/ (visited on 03/2023).

[2] *GitHub Logo.* URL: https://allvectorlogo.com/github-logo/ (visited on 07/2023).

[3] *Hypervisors architectures images.* URL: https://medium.com/teamresellerclub/type-1-and-type-2-hypervisors-what-makes-them-different-6a1755d6ae2c (visited on 07/2023).

[4] *Master thesis repository.* Mar. 2023. URL: https://github.com/Matteo-Locatelli/master_thesis.

[5] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: (Dec. 1992). URL: https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[6] *Ubuntu ISO image download page.* URL: https://www.ubuntu-it.org/download (visited on 04/2023).

[7] *Virtual machine installation instructions.* URL: https://github.com/microsoft/ebpf-for-windows/blob/main/docs/vm-setup.md (visited on 05/2023).

[8] *Windows eBPF Starter.* URL: https://github.com/SubconsciousCompute/windows-ebpf-starter (visited on 06/2023).