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* [24] 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 [22], 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* [25] 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 [23]. 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

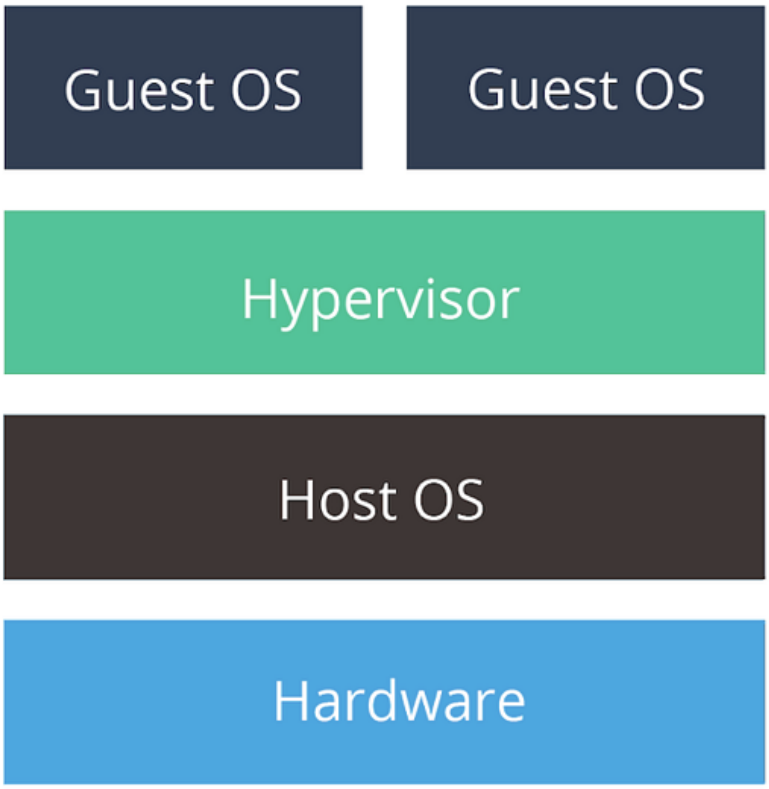

Figure 2.1: Type 2 (or hosted) hypervisor architecture [15].

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 [26].

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"

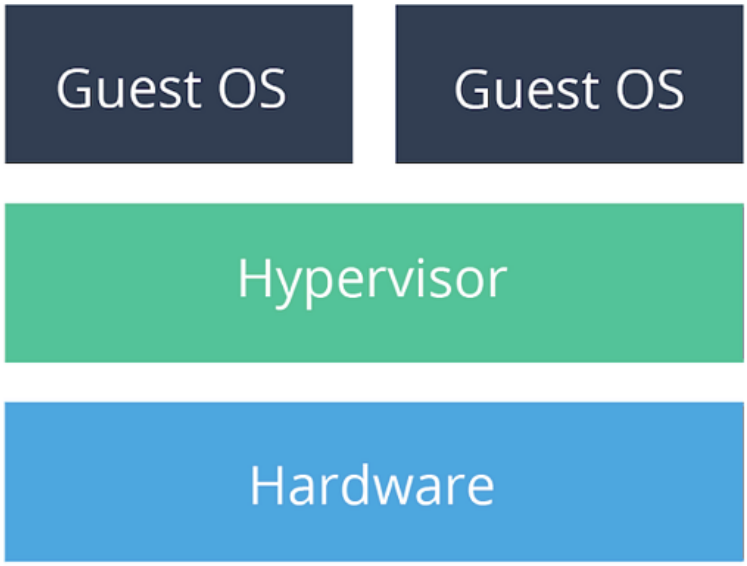

Figure 2.2: Type 1 (or bare metal) hypervisor architecture [15].

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 [14].

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 [20], 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.

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 working at Lawrence Berkeley Laboratory (Berkeley, California, USA), titled *The BSD Packet Filter: a New Architecture for User-level Packet Capture* [21] and presented at the 1993 Winter USENIX conference in San Diego, California, USA (it is just an 11 pages document and it is worth giving it a read). Fun fact, at the beginning of its story, the *B* in BPF stood 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 fact, at its beginning, BPF was running only on the FreeBSD operating system.

In this article they talk about the packet-capture techniques existing at the time and they describe the BSD packet filter (BPF) including its placement in the kernel and implementation as a virtual machine, defining it as "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 plan behind BPF, showing its model and designing a virtual machine (perhaps, the most important thing) that would work as a filter with BPF, emphasizing on expandability, generality and performance. They defined the design of the virtual machine by the following five statement:

- "It must be protocol independent. The kernel should not have to be modified to add new protocol support.";
- "It must be general. The instruction set should be rich enough to handle unforeseen uses";
- "Packet data references should be minimized.";
- "Decoding an instruction should consist of a single C switch statement.";
- "The abstract machine registers should reside in physical registers.".

In the end, they do some examples of packet filtering with BPF and with other technologies to compare their performances on the same hardware, showing how and why BPF performs substantially better than other approaches.

There are two last things that are worth noting in this paper. First, when the paper was published, BPF was approximately two years old in which it had been tested and already found its way into multiple tools This shows that the development of BPF was a gradual one, something that continues with the technologies that succeeded it. Second, it mentions tcpdump as the program that uses BPF the most at the time of writing. tcpdump is 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. Still to our days, it is one of the most widely used network

debugging tools: this shows that tcpdump has used BPF technology for at least thirty years. 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.

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 outperformed other facilities of that time in their filtering mechanisms;
- It had a programmable pseudo-machine model that demonstrated to be general and extensible;
- It was portable and ran on most BSD systems which, due to their Unix-like basis, were a synonym of high quality networking back then;
- It could 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. Then, the filter programs were interpreted by the Linux kernel and executed by the virtual machine.

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 because it was the thing that BPF did so differently that its predecessors: it used a well thought out memory model and then exposed it through an efficient virtual machine inside the kernel. Without requiring the overhead of copying packets between user space and

kernel space, BPF filters could do traffic filtering in an efficient manner while still maintaining a boundary between the filter code and 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, "an accumulator, an index register, a scratch memory store, and an implicit program counter". 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.

But 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, allowing it to work with different types of network interfaces and packet formats. This feature made it a powerful tool for packet filtering and analysis across different network technologies, making possible to apply BPF in a wide range of networking scenarios 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.

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. Therefore, eBPF enables the safe and efficient extension of kernel capabilities without the need to modify kernel source code or load kernel modules.

Historically, the operating system has been an optimal platform for implementing the functionalities that eBPF was designed for (e.g. observability, security, and networking) because it benefits from the kernel's privileged ability to oversee and

control the entire system. However, evolving an operating system kernel is very challenging due to its central role and critical need for stability and security, resulting in traditionally lower innovation rates compared to functionalities implemented outside the operating system. eBPF radically transforms this approach by enabling the execution of sandboxed programs within the operating system, empowering application developers to add extra capabilities at runtime. With the help of a Just-In-Time (JIT) compiler and a verification engine, the operating system also ensures a safe and efficient execution of the programs: in fact, compiling programs into native machine code that could be executed directly by the CPU, addressed the limitations of cBPF regarding the lack of performance and flexibility, improving the execution speed and versatility of eBPF programs compared to the cBPF filtering programs that were written in assembly-like instructions that represent the bytecode and they were interpreted by the kernel's cBPF interpreter that processes each instruction in the program sequentially for every packet.

eBPF has first appeared in the Linux kernel version 3.18 released in December 2014 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.

	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.

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 parameters to be passed to functions in eBPF virtual machine registers just like on native hardware and virtually gave the virtual machine unlimited storage. If you want to

read the details about the differences between cBPF and eBPF you can check "The Linux Kernel Archives" document that talks about this topic [10].

While these changes were introduced in eBPF due to the 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.

- More efficient global data stores, which eBPF calls "maps", were introduced, allowing the state of a process to persist between events and thus be aggregated for uses including statistics and context aware behavior (we will discuss about them in the next chapter).

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 and its simple virtual machine on which eBPF runs, 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*.

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 with origins in the Linux kernel allowing to execute bytecode at various hook points in a safe manner: it processes a virtual instruction set and provides a safe way to extend kernel functionalities. To make a comparison with a famous programming language we can say that eBPF does to the kernel what JavaScript does to websites: it allows the creation of all sort of applications. 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 JIT compiled for native execution performance. Once again, the 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 its use case outgrew networking, even though it evolved from BPF as an extended version. Due to the fact that the acronym does no longer make a lot of sense, eBPF is now considered a standalone term that does not stand for anything. 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. Moreover, BFP adn eBPF are generally used interchangeably in documentation and various tools. 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, named "eBee". So, Vadim Shchekoldin created the eBPF logo, 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 are various and numerous:

- Expand the contributions being made to extend the powerful capabilities of eBPF and grow beyond Linux (as we already mentioned before, eBPF is now also available on Windows);

Figure 3.2: eBPF Foundation logo.

- 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.

Basically, the foundation wants to get as many people as possible to adopt eBPF and involve them into the project. To do so, they also created a place where everybody can learn and collaborate to the topic of eBPF which is called *eBPF.io* [13]. Throughout the years eBPF has been surrounded with an open community and everybody can participate and share: eBPF.io is a website where anyone can learn something about eBPF, from reading a first introduction to listen to some community talks, and become a contributor to major eBPF projects.

3.6.3 Use cases of eBPF

We understood that eBPF programs are verified within the kernel to avoid various threats: 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.

To summarize what we have seen until now, eBPF has only been in the Linux kernel since 2014, but has already worked its way into a number of different uses in the kernel for efficient event processing (socket filtering, capturing information, analyzing performances, attaching programs to hook points or probes,...). However, 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 The portability 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 in the kernel that may lack stability. For this reason, eBPF programs may not be portable across different kernels.

In fact, the main priority of the eBPF community since its creation was to make the development od eBPF application as simple as possible, making it a similar experience to developing any application in user-space. Even though there were many usability improvements during the years, the aspect of portability has always been forsaken (mostly for technical reasons).

"BPF portability is the ability to write a BPF program that will successfully compile, pass kernel verification, and will work correctly across different kernel versions

without the need to recompile it for each particular kernel", says Andrii Nakryiko, a kernel engineer at Meta and member of the BSC, in a post [4] published on his blog [2]. 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.

3.7.1 The problem of portability

So far we understood that the power of eBPF is the fact that a piece of user-provided code (the program) is injected straight into a kernel and, after the phases of verification and loading, executes in kernel context, operating inside kernel memory space with access to all the internal kernel state available to it. However, at the beginning of eBPF this powerful capability also created some portability problems: eBPF programs do not control memory layout of a surrounding kernel environment. This means that they have to work with what they get from independently developed, compiled and deployed kernels.

Moreover, new kernel versions are continuously released (as of September 2023 the Linux kernel is at version 6.5, far from the 3.18 of December 2014): so, kernel types and data structures are in constant flux. The problem is that kernel version may differ under various architectural aspects: struct fields are shuffled around inside a struct or even moved into a new inner struct, fields can be renamed or removed, their types changed, either into some compatible ones or completely different ones, structs and other types can get renamed or just plain removed.

Even if not all eBPF programs need to look into internal kernel data structures and eBPF machinery inside kernel provides a limited set of stable interfaces that eBPF programs can rely on to be stable between kernels (in reality, underlying structures and mechanisms do change, but these eBPF-provided stable interfaces abstract such details from user programs), things change all the time between kernel releases and yet BPF application developers are expected to address this problem in some way.

Even if not all eBPF programs require direct access to kernel data structures and the eBPF framework offers a limited set of stable interfaces that abstract changes

between kernel versions (underlying structures and mechanism do change), things mutate all the time between kernel releases and yet BPF application developers are expected to address this problem in some way.

3.7.2 The temporary solution: BCC

The first thing that people started using for addressing this problem is *BPF Compiler Collection* (BCC) [3], a toolkit for creating kernel programs suited for different tasks, such as network traffic and performance analysis. To make sure that the running kernel's memory layout is the same as the one expected by the eBPF program, when the application is executed by the host, BCC calls its embedded Clang/LLVM, puts the headers into the kernel and does compilation on the fly. Additionally, you can define and rename any optional stuff not available on the kernel configuration that you are using and the Clang will adapt your eBPF program code to the specific kernel.

While this workflows work, it has some problems. Firstly, the Clang/LLVM combo is a big library and resource heavy: this means that you have to install big binaries when you distribute your application and the process of compilation can require a lot of time. Secondly, you have to hope that the system on which you are going to install your application has the kernel headers present, because BCC-based application do not work on kernels that have been custom built. Lastly, working in an agile method is quite difficult because you will get compilation errors only at runtime and you have to recompile and restart your application every time.

Although BCC is a great tool for experimenting small tools, when we look at some example of widely deployed, complex and real-world eBPF application we have to think of an other solution.

3.7.3 The solution: BPF CO-RE

BPF Compile Once - Run Everywhere (BPF CO-RE) is a feature in the eBPF ecosystem that aims to solve the problem of portability of eBPF programs across different versions and architectures of the Linux kernel which was presented at the

"Linux Storage, Filesystem and Memory Management (LSF/MM) Summit for 2019".

BPF CO-RE allows to easily write portable eBPF programs: to do so, it requires the integration and the cooperation of different components:

- *BPF Type Format* (BTF), a compact, but expressive enough format to describe the information of C programs, that is used to enhance the verifier's capabilities;
- A support for the compiler, as Clang had to be extended with built-ins that allow the capture of field offset, existence and size, type size and relocation and enum values and existence;
- A loader, named *libbpf* [16], that takes the BPF object file (the program after its compilation) and triggers the phases of loading and verification;
- The kernel, which does not need many changes to support BPF CO-RE.

With BPF CO-RE, eBPF programs are compiled into a more compact, intermediate representation that can be loaded and executed by various kernel versions. This reduces the need to recompile programs for different kernel versions, making eBPF programs more portable and efficient.

To enable BPF CO-RE and let eBPF loader to adjust an eBPF program to a particular kernel running on target host, the new built-ins for Clang release *BTF relocations* which capture a high-level description of what pieces of information the eBPF program code want to read. If you want to access a certain field in a struct inside the kernel and this field has been moved to a different offset inside the same struct or even to a different struct, the developer can find that field by just its name and type information.

Once we have the object file of the compiled eBPF program, the Clang relocations and the BTF information provided by the running kernel of the host, the loader makes sure that the logic of that program is correctly functioning for the specific kernel by matching these information.

The result is that it looks like the program was compiled specifically for the kernel of that machine, but this happened without distributing Clang with your

application and performing compilation in runtime on target host. In fact, thanks to a good separation of concerns, after the loader has processed the eBPF program, from the kernel perspective you see a valid eBPF program code (and everything is done without worrying about the kernel version).

3.7.4 BPF CO-RE as today

BPF CO-RE is now a mature technology used across a wide variety of projects: it is used for the development of many eBPF-based applications, due to its capability to handle, in a single compiled-once eBPF, application both simple cases of changing field offsets and much more advanced cases of kernel data structures being removed, renamed or completely changed.

We understood how BPF CO-RE helped the developers to solve the portability issues of eBPF. We do not have to forget that it also provides a good usability and familiar workflow of compiling C code into binary and distributing lightweight binaries around. This eliminated the need to install a heavy-weight compiler library together with your application and the cost of precious runtime resources for runtime compilation. Furthermore, there is no more need to catch sneaky compilation errors at runtime.

There are other complex things that BPF CO-RE makes easier for the user for dealing with different kernel versions and configuration differences: the curious ones can read more about this topic on Nakryiko's post [4].

3.8 Future and potential of eBPF

In the previous paragraph we showed how the problem of portability was resolved. However, 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 eBPF cannot be universally considered portable or 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.

Nevertheless, the world of eBPF evolves quickly and distributions appear to reg-

ularly support eBPF and provide a package of eBPF 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 eBPF programs more portable.

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;

It's highly likely that the trend of using eBPF for safe and efficient event handling will continue. Thanks to its restrictive and simple implementation, eBPF offers a highly portable and performant way to process events. More than that, however, eBPF makes a change in how problems are solved: instead of using objects and stateful code, it exploits just functions and efficient data structures to store state. By doing so, the possibilities of a program's design are reduced, but allows eBPF to be used with nearly any method of program design (synchronously, asynchronously, in parallel, distributed, ... depending on the coordination needs with the data store).

In conclusion, 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.

ttps://www.ferrisellis.com/tags/ebpf/
> PART 1
> KERNEL-
PACE VM ?
NEW COM-
MANDS

Chapter 4

How eBPF works

From what we have learned from the previous chapter we can try to give a definition to eBPF: it is a "verified-to-be-safe, fast to switch-to, mechanism, for running code in Linux kernel space to react to events such as function calls, function returns, and trace points in kernel or user space" [11]. In a few words, eBPF is very powerful because it is fast and it is safe.

Given also eBPF's efficiency and flexibility, Brendan Gregg, an internationally famous expert in computing performance, famously coined eBPF as "superpowers for Linux". Linus Torvalds, the author of the first version of the Linux kernel, 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". Once again, we mention the fact that due to its success in Linux, the eBPF runtime has been ported to other operating systems such as Windows.

Like all superheroes are shocked when they first come across their superpowers, eBPF too can seem overwhelming at first glance. To fully appreciate it, the goal of this chapter is to explain everything you need to know about eBPF.

4.1 Writing an eBPF program

In the previous chapter we understood the fact that, to achieve safety guarantee, eBPF is essentially implemented as a process virtual machine in the kernel which runs safe programs on behalf of the user. eBPF exposes to the user a virtual pro-

cessor, with a custom set of RISC-like instructions and also provides a set of virtual CPU registers and a stack memory area. Thanks to this features, developers can write programs in eBPF bytecode (the form in which the Linux kernel expects eBPF programs) and pass them to the virtual machine to make them run in this environment.

While it is of course possible to write bytecode directly, developers do not have to create eBPF bytecode from scratch when writing a new program. It has been implemented an eBPF back-end for Low-Level Virtual Machine (LLVM, "a collection of modular and reusable compiler and toolchain technologies" [19]): as a result Clang, the LLVM front-end compiler for C-derived programming languages, can be used to compile a subset of standard C code in an eBPF object file. While the C to eBPF translation must be done in a very cautious way, it massively expands the use cases of eBPF due to the fact that it makes relatively easy to write new eBPF code in a familiar programming language such as C.

At this point it is important to mention that in a lot of scenarios, eBPF is used indirectly via projects like Cilium, bcc, bpftrace,... The peculiarity of these projects is the fact that they provide an abstraction on top of eBPF and do not require writing programs directly: instead, they offer the ability to specify intent-based definitions which are then implemented with eBPF. If no higher-level abstraction exists, programs need to be written directly. We are going to look at some of this projects in the next chapter of this paper.

For now, in the next paragraphs of this chapter, we are going to look at the components mentioned above and how they work in practice, including how the program safety verification is done.

4.2 Architecture

We understood that the architecture of eBPF (extended Berkeley Packet Filter) is characterized by its ability to provide programmability within the kernel, offering a powerful framework for safe and efficient extension of kernel functionality. At its core, eBPF operates as an in-kernel virtual machine, running sandboxed programs

that are designed to enhance kernel behavior without requiring changes to the kernel source code or loading kernel modules.

When we talk about an eBPF program, we have to consider a big infrastructure of things that make this technology interesting:

- The instruction set, which defines the main characteristics of eBPF;
- Maps, efficient key/value data structures;
- Helper functions, to exploit kernel functionalities;
- Tail calls, for calling into other eBPF programs;
- Hook points, which are points of execution in the kernel to which an eBPF program is attached;
- A verifier, a program used to determine the safety of a program;
- A compiler, used to compile the program in an object file that can be loaded in the kernel;
- The kernel subsystem that uses eBPF.

When an eBPF program passes the verification process, it is then compiled, loaded in the kernel and attached to a hook point. When the associated event or condition occurs in the kernel, the attached eBPF program is triggered to execute and it receives some input data coming from the kernel (e.g. the system call arguments passed by the user space process invoking the system call to the kernel, if the program is attached to a system call execution via a system call tracepoint): the program can then manipulate the input data tu perform various operations, such as filtering a packet (for networking use), compute a set of metric (typically for tracing, where the programs are attached to a very busy execution point in the kernel) or interact with the kernel, as defined by the program's logic.

The following paragraphs provide further details on individual aspects of the eBPF architecture.

4.3 The instruction set

In order to guarantee good performance on the kernel side, the RISC instruction set of an eBPF program is simple enough that it can be relatively easily translated into native machine code via a JIT step embedded inside the kernel. This means that right after the verification of the safety of the program, the runtime will not actually suffer the performance overhead of having to execute the eBPF bytecode via the virtual machine. It will just execute straight native machine code, significantly improving the performance.

Moreover, the general purpose RISC instruction set was designed for writing eBPF program in a subset of C which can be compiled into BPF instructions through a compiler back end (e.g. LLVM), so that the kernel can later on map them through an in-kernel JIT compiler into native opcodes for optimal execution performance inside the kernel.

There are several advantages for pushing these instruction into the kernel:

- The kernel is made programmable without having to cross the boundaries between kernel-space and user-space;

- Programs can be heavily optimized for performance by compiling out features that are not required for the use cases the program solves;

- eBPF provides a stable Application Binary Interface (ABI, the machine language interface between the operative system and its applications) towards user space and does not require any third party kernel modules because it is a core part of the Linux kernel that is shipped everywhere, making eBPF programs portable across different architectures;

- eBPF programs work with the kernel, making use of existing kernel infrastructure (e.g. drivers, netdevices, sockets, ...) and tooling (e.g. iproute2) as well as the safety guarantees which the kernel provides.

4.4 Hook points

eBPF programs are event-driven by design and are run when the kernel or an application passes a certain *hook point*. When the designated code path is traversed, any eBPF program attached to that point is executed. In the kernel there are some pre-defined hooks, including system calls, function entry/exit, kernel tracepoints, network events and several others. It is also possible to create custom hook points to attach eBPF programs almost anywhere in kernel or user applications by creating a kernel probe (kprobe) or user probe (uprobe).

Given its origin, eBPF works really well writing network programs and it's possible to write programs that attach to network sockets, enabling the user to do many different operations such as traffic filtering, classification and network classifier actions. Even the modification of established network socket configurations can be achieved through eBPF programs. A notable use case is the eXpress Data Path (XDP) project [28], which leverages eBPF to carry out high-performance packet processing by executing eBPF programs at the network stack's lowest level, immediately following packet reception.

In addition to network-oriented applications, we already discussed that eBPF has many other purposes: it can filter and restricting system calls, debug the kernel and carry out performance analysis. To do so, programs can be attached to tracepoints, kprobes and perf (a tool to analyze performance in the Linux kernel) events. Because eBPF programs can access kernel data structures, developers can write and test new debugging code without having to recompile the kernel (the implications are obvious for engineers whose work is to debug issues on live and running systems).

When the desired hook has been identified, the eBPF program can be loaded into the Linux kernel for verification and further use using the *bpf* system call. This is typically done using one of the available eBPF libraries.

4.5 Compiling and loading an eBPF program

Once we have decided where we want to attach our eBPF program (based on the operation that we want to do), the eBPF framework will start executing this program

only after verifying that they are safe from an execution point of view.

An eBPF program has to go through a series of steps before being executed inside the kernel.

4.5.1 Compilation

We already said that an eBPF program is written in a high-level programming language, such as C. The first thing that happens to a program is its compilation using Clang with its eBPF backend LLVM: this process generates eBPF bytecode which resides in an Executable and Linkable Format (ELF, a standard file format for executables, object code, shared libraries, and core dumps.) file.

As this file is loaded into the Linux kernel, it passes through two steps before being attached to the requested hook: verification and JIT compilation.

4.5.2 Verification

There are security and stability risks with allowing user-space code to run inside the kernel. So, a number of checks are performed on every eBPF program before it is loaded. The generated eBPF bytecode undergoes verification by a safety tool, the eBPF *verifier*, within the kernel to ensure that the eBPF program is safe to run. It is not a security tool inspecting what the programs are doing. The verifier checks the bytecode for safety, ensuring that it sticks to certain constraints and security rules to prevent potential security vulnerabilities. The safety of the eBPF program is determined in two steps.

The first test ensures that the eBPF program terminates and does not contain any loops that could cause the kernel to lock up. To do so, the verifier does DAG check to disallow loops and a depth-first search of the program's control flow graph (CFG). Any program that contains unreachable instructions will fail to load, as they are strictly prohibited (though classic BPF checker allows them). Furthermore, there must not be infinite loops: programs are accepted only if the verifier can ensure that loops contain an exit condition which is guaranteed to become true.

The second part requires the verifier to run all the instructions of the eBPF

program one at the time: from the first instruction, the verifier descends all possible paths, simulating the execution of all instructions and observing the state change of registers and stack. Then, the virtual machine state is checked before and after the execution of every instruction to ensure that register and stack state are valid. This step is done to check two major things:

- If programs are trying to access invalid memory or out-of-range data (outside the 512 byte of stack designated to each program), due to the presence of out of bounds jumps, and use uninitialized variables because they should not have the ability to overwrite critical kernel memory or execute arbitrary code;
- If programs have a finite complexity (the verifier must be capable of completing its analysis of all possible execution paths within the limits of the configured upper complexity limit).

Although this second operation seems expensive in computation terms, the verifier is smart enough to know when the current state of the program is a subset of one it's already checked. Since all previous paths must be valid (otherwise the program would already have failed to load), the current path must also be valid. This allows the verifier to perform a sort of "pruning" to some branches and skip their simulation.

Another thing that is not generally allowed by the eBPF verifier is pointer arithmetic because it works under a "secure mode" which enables only privileged processes to load eBPF programs. The idea is to make sure that kernel addresses do not leak to unprivileged users and that pointers cannot be written to memory. Unless unprivileged eBPF is enabled (and secure mode is not enabled), then pointer arithmetic is allowed but only after additional checks are performed (e.g. all pointer accesses are checked for type, alignment and bounds violations).

In general, untrusted programs cannot load eBPF programs: all processes that want to load eBPF programs in the kernel must be running in privileged mode. However, you can enable "unprivileged eBPF" which allows unprivileged processes to load some eBPF programs subject to a reduced functionality set and with limited access to the kernel.

Lastly, the verifier uses the eBPF program type (covered later) to restrict which kernel functions can be called from eBPF programs and which data structures can be accessed. In fact, an eBPF program cannot randomly modify data structures in the kernel and arbitrary access kernel memory directly. To guarantee consistent data access, an eBPF program running is allowed to modify the data of certain data structures inside the kernel only if the modification can be guaranteed to be safe and it can access the data outside of the context of the program via only eBPF helpers (which we will discuss later).

4.5.3 Hardening

Once the verifier has successfully completed his job, the eBPF program undergoes an hardening process according to whether the program is loaded from privileged or unprivileged process.

Hardening refers to the process of enhancing the security and safety of eBPF programs to prevent potential vulnerabilities and ensure their reliable and controlled execution within the kernel. This is particularly important because, as we already know, eBPF programs have the capability to run within the kernel's context, which requires robust measures to mitigate risks.

This step includes two main operations:

- The kernel memory holding an eBPF program is protected and made read-only and any attempt to modify the eBPF program (through a kernel bug or malicious manipulation) will crush the kernel instead of allowing it to continue executing the corrupted/manipulated program;
- All constants in the code are blinded to prevent attackers from injecting executable code as constants which, in the presence of another kernel bug, could allow an attacker to jump into the memory section of the eBPF program to execute code (JIT spraying attacks, similar to JavaScript injection);

By following these practices, developers can minimize security risks and ensure that eBPF programs operate safely and reliably within the kernel's context, ensuring that only safe and well-behaved programs are allowed to run. This process of

hardening helps prevent potential security vulnerabilities and ensures the reliable and secure operation of eBPF programs.

4.5.4 JIT compilation

Once the bytecode has been verified and hardened, the eBPF JIT compiler takes over. It translates the verified eBPF bytecode into native machine code that corresponds to the target CPU architecture which can be directly executed by the processor. This native code is generated on-the-fly and is specific to the underlying hardware, ensuring optimal execution of eBPF programs by eliminating the overhead of interpreting bytecode. The JIT compilation step makes eBPF programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module.

In fact, JIT compilers speed up execution of the eBPF program significantly since they reduce the per instruction cost compared to the interpreter used in cBPF. Most of the times, instructions can be mapped one-to-one with native instructions of the underlying architecture. This also reduces the resulting executable image size of the program and is therefore more instruction cache friendly to the CPU. Moreover, during JIT compilation, the compiler can apply various optimization techniques to enhance the efficiency of the generated machine code, which aim to reduce redundant operations, improve memory access patterns and optimize CPU registers usage.

4.5.5 Loading and execution

The resulting native machine code is then loaded into the kernel's memory space: this is done in Linux using the *bpf()* system call (see the next paragraph). When the predefined event or hook associated to the eBPF program is triggered (e.g., a network packet arrival or a system call), its native machine code generated by the JIT compiler is executed directly by the CPU. This execution is significantly faster than interpreting bytecode, leading to improved performance.

As eBPF serves diverse purposes across various kernel subsystems, each eBPF program type has a distinct procedure for attaching to its relevant system. Once the program is attached, it becomes operational, engaging in activities such as filtering,

analysis or data capture, according to its intended function. Subsequently, user-space programs can manage active eBPF programs, involving actions like reading states from eBPF maps and, if designed accordingly, modifying the eBPF map to influence program behavior.

Furthermore, while the program is running, the JIT compilation process allows for dynamic adaptation of the eBPF program's behavior based on the runtime environment: if changes occur in the system or the program's requirements, the eBPF JIT compiler can recompile the bytecode into a different native machine code to ensure optimal performance.

4.6 The bpf() system call

The creation of the eBPF program as byte code and attaching the loaded program to a system in the kernel are two steps in the process of using an eBPF program that vary by use case. However, the step in between these two, that is loading the program into the kernel and creating necessary eBPF-maps, is the core of eBPF and it is what all eBPF applications have in common. In Linux, this step is done by the *bpf()* system call, which was introduced in the Linux kernel version 3.18, released on the 7th of December 2014, along with the underlying machinery in the kernel: it is an interface provided by the Linux kernel that allows user programs to interact with and utilize eBPF functionality. It serves as a bridge between user space and the kernel, acting as a gateway for user applications to utilize the power of eBPF within the kernel. This system call allows for the byte code to be loaded along with a declaration of the the type of eBPF program that's being loaded and provides many more key functionalities, such as program execution, maps initialization for data exchange, helper function invocation and error handling.

Below we can see the necessary syntax of this system call:

```
1 #include <linux/bpf.h>
2 int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

The first line is a must when we want to exploit eBPF functionality: the *linux/bpf.h* header file in the Linux kernel contains a collection of macro definitions,

function prototypes and data structures related to the eBPF subsystem and programs. This header file provides the necessary interfaces and definitions for user space programs to interact with the eBPF subsystem in the kernel: it includes various constants, helper function prototypes, map data structure definitions and other components that are essential for programming with eBPF in the Linux kernel.

The second line, instead, shows the syntax of the *bpf()* system call:

- The *cmd* argument tells the operation that has to be performed and essentially defines an API since the type of program loaded in the kernel dictates where the program can be attached, which in-kernel helper functions the verifier will allow to be called, whether network packet data can be accessed directly and the type of object passed as the first argument to the program.;
- The *attr* argument, a pointer to a union of type *bpf_ attr*, is an accompanying argument which allows data to be passed between the kernel and user space in a format that depends on the *cmd* argument (the unused fields and padding must be zeroed out before the call);
- The "size" argument is the size of the union pointed by *attr* in bytes.

We are not going to describe in detail all the possible values that there are for the *cmd* and *attr* arguments: the ones who want to deepen these topics can read the Linux manual page related to the *bpf()* system call [6] or can go through different files directly related to using eBPF from user-space that can be found on the GitHub repository of the Linux kernel [18], such as the latest Linux kernel code related to this system call [7] or the *bpf.h* header file [5] for assisting in using it. .

The most important thing to know is that the *bpf()* macro is not meant to be directly called in eBPF programs; instead, it serves as a placeholder to indicate the invocation of helper functions during the JIT compilation process. When we write eBPF programs, we don't explicitly use *bpf()* in our code. Instead, we use the names of specific helper functions provided by the eBPF runtime. These helper functions are then invoked indirectly through the *bpf()* macro during the JIT compilation process: it essentially tells the eBPF verifier and JIT compiler that a helper function

is being called at that point in the program. The actual mapping from *bpf()* to the appropriate helper function is handled by the eBPF runtime during the loading and verification process. So, while there is only one *bpf()* macro, there are many different eBPF helper functions that it represents, each with its own specific functionality and usage.

4.7 Tail and function calls

eBPF programs are modular thanks to the the concepts of tail and function calls.

Function calls allow defining and calling functions within an eBPF program: this is a standard procedure in all programming languages. But there are a couple of things that developers have to consider when they declare a function in an eBPF program: At the beginning of eBPF, all the reusable functions have to be declared *inline*, resulting in duplication of these functions in the object file of the program. The main reason was that the loader, the verifier and the JIT compiler were not supporting the call of functions. From Linux kernel 4.16 and LLVM 6.0, this constrain got lifted and eBPF programs do not longer need to use *inline* everywhere. This was an important performance optimization since it heavily reduces the generated eBPF code size and therefore becomes friendlier to a CPU's instruction cache. Moreover, it is a good practice to put *static* in the signature of all methods of eBPF programs: since they are written in a restricted set of C, static functions are not visible outside the translation unit, which is the object file the program is compiled into, increasing the level of safety in the program.

Tail calls, however, are a mechanism within the eBPF programming framework that enables one eBPF program to efficiently invoke another eBPF program and replace the execution context, similar to how the *execve()* system call operates for regular processes, without returning back to the old program. This second mechanism has minimal overhead (unlike function calls) and it is implemented as a long jump, reusing the same stack frame: this allows the modularization and reuse of eBPF logic, promoting code organization, maintainability and performance.

When an eBPF program encounters a tail call instruction, it effectively transfers

control to another eBPF program specified by the instruction. The key characteristic of a tail call is that it replaces the current program's execution context with the context of the called program. This replacement avoids the need for an additional return from the called program, which can help reduce execution overhead and improve overall performance.

Moreover, the programs have to observe a couple of constraints to be tail called:

- Only programs of the same type can be tail called and they also need to match in terms of JIT compilation (either JIT compiled or only interpreted programs can be invoked, but not mixed together);
- Programs are verified independently of each other.

Tail calls are particularly useful in scenarios where multiple eBPF programs share common logic or need to perform similar tasks. Instead of duplicating code across multiple programs, developers can create a single eBPF program that encapsulates the shared logic and other programs can invoke it using tail calls. This approach improves code reuse, simplifies maintenance and reduces the potential for errors.

How is a tail call performed? There are two components:

- A special map (key-value data structure), called *BPF_MAP_TYPE_PROG_ARRAY*, has its values populated by file descriptors of the tail called eBPF programs (currently it is write-only from user-space side);
- A *bpf_tail_call()* helper is called, where the context, a reference to the program array and the lookup key of the map are passed to.

Then, the kernel inlines this helper call directly into a specialized eBPF instruction. It takes the key passed to the helper and looks for that value in the map to pull the file descriptor: then , it atomically replaces program pointers at the given map slot.

If the provided key is not present in the map, the kernel will just continue the execution of the old program with the instructions following after the bpf_tail_call().

The use of tail calls is an optimization technique that contributes to the efficiency of eBPF programs. By minimizing the overhead associated with program transitions

and context switches, eBPF tail calls enhance the performance of activities (e.g. packet processing and tracing) carried out by eBPF programs within the kernel. Furthermore, during runtime, a developer can alter the eBPF program's execution behavior by adding or replacing atomically various functionalities.

Up to Linux kernel 5.9, subprograms and tail calls were mutually exclusive: eBPF programs that used tail calls could not take advantage of reducing program image size and faster load times. Since Linux kernel 5.10, the developer is allowed to combine the two features, but with some restrictions:

- Each subprogram has a limit on the stack size of 256 byte;
- If in an eBPF program a subprogram is defined, the main function is treated as a sub-function as well;
- The maximum number of tail calls is 33, so that infinite loops can't be created.

In total, with this restriction, the eBPF program's call chain can consume at most 8 kB of stack space. Without this, eBPF programs will run on a stack size of 512 bytes, resulting in a total size of 16 kB for the maximum number of queue calls that could overload the kernel stack on some architectures.

4.8 Helper functions

eBPF programs cannot call into arbitrary kernel functions. If this was allowed, eBPF programs would be tied to particular kernel versions and would complicate compatibility of programs. Instead, eBPF programs can use helper functions, which are implemented inside the kernel in C and are thus hardcoded and part of the kernel ABI.

These helpers are one of the major things that makes eBPF different from cBPF: they are a set of predefined functions provided by the eBPF runtime environment to assist eBPF programs in performing various tasks and interacting with the kernel. In a few words, they execute some operation on behalf of the eBPF program, natively, to interact with the system or with the context in which they work.

Being functions, their signature is the typical one that all functions in C have: a return type, an name of the helper and a list of arguments. The specific signatures of eBPF helpers may vary based on the helper's purpose and the operations it supports. It's important to refer to the eBPF documentation or header files for the precise signatures and usage details of each helper function (both for Linux [17] and Windows [27]). These functions are invoked by the eBPF program itself using a mechanism similar to a function call: when an eBPF program encounters a helper function call, it generates a specific bytecode instruction that indicates which helper function to invoke and which required arguments need to be provided. Then, the kernel's eBPF verifier checks these instructions and only if they are safe and valid the program can continue its execution.

There are a few more things that a developer has to take into account when using eBPF helper functions:

- Since there are several eBPF program types and that they do not run in the same context, each program type can only call a subset of those helpers;
- Due to eBPF conventions, a helper can not have more than five arguments;
- For how an helper call behaves, we can understand that calling helpers introduces no overhead, thus offering excellent performance (internally, eBPF programs call directly into the compiled helper functions without requiring any foreign-function interface).

Therefore, eBPF helpers serve as a bridge between the eBPF program and the underlying kernel, providing a safe and controlled way to perform operations that would otherwise be restricted due to the isolated nature of eBPF programs, such as accessing and manipulate data, performing calculations, interacting with external resources and making decisions based on specific conditions. Although developers can do many operations with current helpers, the set of available helper calls is constantly evolving. Some common functionalities of eBPF helper functions include:

- Allowing eBPF programs to read from and write to memory locations to ensure that memory access is properly bounded and does not violate kernel memory protection;

- Enabling eBPF programs to inspect and modify network packets, headers and data, used for tasks like packet filtering, classification modification;
- Getting access to various time-related information, such as timestamps and timers, allowing eBPF programs to track time and perform time-sensitive operations;
- Doing mathematical operations, enabling eBPF programs to perform calculations, manipulate numeric values and generate random numbers;
- Inserting, updating and deleting key-value pairs in maps, since eBPF programs can interact with eBPF maps;
- Helping eBPF programs implement synchronization mechanisms to safely access shared data structures;
- Enabling eBPF programs to interact with tracepoints and perf events, allowing for efficient tracing and profiling of kernel and user-space events;
- Allowing eBPF programs to interact with files and sockets, enabling I/O operations and communication between eBPF programs and user space;
- Letting the program to print debugging messages.

To sum it up, eBPF helpers provide a standardized way for eBPF programs to consult a core kernel defined set of function calls in order to perform essential tasks (retrieve/push data from/to the kernel) without compromising safety and security. They are a critical component of the eBPF ecosystem and contribute to the versatility and power of eBPF programs in all of its use cases.

4.9 Maps

Another substantial difference between cBPF and eBPF is the introduction of maps: they are more or less generic key-value data structures that reside in kernel space used to allow efficient storage and low-throughput data flow between user and kernel space while being persistent across different invocations. In particular, eBPF maps

can be accessed from eBPF programs using helper functions as well as from applications in user space via a system call. They serve as a mechanism for communication and coordination between eBPF programs and user applications.

The life cycle of maps is very simple: when a map is successfully created, a file descriptor associated with that map is returned and they are normally destroyed by closing the associated file descriptor. eBPF maps enable the following functionalities:

- Store and retrieve any data, from counters, statistics and configuration settings to complex data structures;
- Allow the exchange of data between kernel and user space, useful for scenarios where an eBPF program needs to provide information to a user application or vice versa;
- Enable multiple eBPF programs (which are not required to be of the same program type) to interact with the same map for collaborating and sharing data, important for implementing advanced use cases (e.g. packet filtering, flow tracking and more);
- Allow the same eBPF program to access many different maps directly;
- Persist data across different executions of eBPF programs or even across system reboots, making them suitable for long-term data storage and retrieval.

eBPF maps come in different types, each designed for specific use cases. It is not in the interest of this paper to present all map types: the ones who want to check them can visit the Linux kernel documentation about eBPF maps [12]. It is enough to know that each map is defined by four values: a type, a maximum number of elements, a value size in bytes and a key size in bytes. Furthermore, there are generic maps with per-CPU and non-per-CPU flavor that can read and write arbitrary data and some map types work with additional eBPF helper functions that perform special tasks based on the map contents.

So, eBPF maps provide a powerful mechanism for eBPF programs to interact with the wider system, enabling dynamic data sharing and coordination between the kernel and user space.

Chapter 5

Applications and infrastructure of eBPF

Integrating eBPF into modern applications and infrastructures can involve various levels of experience. Analyzing Linux kernel issues with eBPF, for instance, might demand significant kernel expertise, identifying relevant kernel functions and understanding their arguments. While running an eBPF tool can be easy, understanding its output and choosing the right things to look at can present considerable challenges.

In this chapter we will address these challenges by reviewing a list of applications, in the for of their GitHub projects, that I have personally used to enter the world of eBPF, as they were either important to its evolution or were designed to make the development of programs easier.

For the curious people, on the eBPF.io website [13], under the section *Project landscape*, many other applications can be found and there is also a list of projects that represent the major infrastructure of eBPF.

If anyone is interested in the raw observability tools to get started with eBPF, in addition to the applications of eBPF, you can check the book *BPF Performance Tools: Linux System and Application Observability* [9], written by Brendan Gregg in 2019, and the official GitHub repository of the book [8]: by presenting the utility, the capabilities and the value of different eBPF tools, the author hopes that the book can help the reader to improve performance, reduce costs and solve software

issues of systems and applications.

5.1 BCC

5.2 bpftrace

5.3 libbpf

5.4 Bumblebee

5.5 libbpf-bootstrap

5.6 ebpf for Windows

Bibliography

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

[2] *Andrii Nakryiko blog.* URL: https://nakryiko.com/ (visited on 04/2023).

[3] *BCC GitHub repo.* URL: https://github.com/iovisor/bcc/ (visited on 04/2023).

[4] *BPF CO-RE post.* URL: https://nakryiko.com/posts/bpf-portability-and-co-re/ (visited on 04/2023).

[5] *BPF header for user-space use.* URL: https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h (visited on 06/2023).

[6] *BPF Linux manual page.* URL: https://man7.org/linux/man-pages/man2/bpf.2.html (visited on 06/2023).

[7] *BPF syscall Linux kernel related code.* URL: https://github.com/torvalds/linux/blob/master/kernel/bpf/syscall.c (visited on 06/2023).

[8] *Brendan Gregg BPF performance tools book GitHub repo.* URL: https://github.com/brendangregg/bpf-perf-tools-book.

[9] *Brendan Gregg BPF performance tools book website.* URL: https://www.brendangregg.com/bpf-performance-tools-book.html.

[10] *cBPF vs eBPF.* URL: https://www.kernel.org/doc/html/latest/bpf/classic_vs_extended.html (visited on 04/2023).

[11] *eBPF Linux journal.* URL: https://www.linuxjournal.com/content/bpf-observability-getting-started-quickly (visited on 04/2023).

[12] *eBPF Linux maps documentation.* URL: https://docs.kernel.org/bpf/maps.html (visited on 06/2023).

[13] *eBPF.io website.* URL: https://ebpf.io/ (visited on 04/2023).

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

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

[16] *libbpf GitHub repo.* URL: https://github.com/libbpf/libbpf (visited on 04/2023).

[17] *Linux helpers manual page.* URL: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html (visited on 06/2023).

[18] *Linux kernel repository.* URL: https://github.com/torvalds/linux/tree/master (visited on 06/2023).

[19] *LLVM project website.* URL: https://llvm.org/.

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

[21] 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.

[22] *Subconscious Compute website.* URL: https://www.subcom.tech/ (visited on 06/2023).

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

[24] *Unibg Seclab website.* URL: https://seclab.unibg.it/ (visited on 02/2023).

[25] *Unibg website.* URL: https://www.unibg.it/ (visited on 06/2023).

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

[27] *Windows helpers manual page*. URL: https://microsoft.github.io/ebpf-for-windows/bpf__helper__defs_8h.html (visited on 06/2023).

[28] *XDP website*. URL: https://www.iovisor.org/technology/xdp.