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* [52] 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 [49], 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* [53] 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 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 lightweight

and 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: the history of eBPF (with real-world examples);
- Chapter 3: How eBPF works
- Chapter 4: Applications ad infrastructure of eBPF (BCC, libbpf, and many more from ebpf.io)
- Chapter 5: eBPF on Linux (installation and programs development);

- Chapter 6: eBPF on Windows (installation and programs development);
- Chapter 7: 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.

1.5 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 1.1. 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.

Figure 1.1: GitHub *Invertocat* logo [29].

Throughout the course of this master's thesis about 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

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 [42], the 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 was used as an invaluable resource for the eBPF community: during our research, we encountered several repositories (which we will discuss later) dedicated to developing and optimizing eBPF environments, tools and libraries. By studying and understanding their implementations, we were 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 us 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 2

The history of eBPF

This chapter digs in the historical journey of 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.

2.1 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* [43] 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.

2.2 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 another 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.

Another 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.

2.3 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.

2.4 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 written with the native kernel APIs.

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 2.1 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 2.1: 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 anyone wants

to read the details about the differences between cBPF and eBPF you can check "The Linux Kernel Archives" document that talks about this topic [21].

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*.

2.5 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, and so forth). 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.

2.6 eBPF in modern architecture

2.6.1 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 2.1.

Figure 2.1: eBPF logo.

2.6.2 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 2.2

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 2.2: 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* [27]. 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.

2.6.3 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 and many more, 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, etc.). 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.

2.7 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 of 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 was considered just an afterthought (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 [8] 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.

2.7.1 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 evolution. 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.

2.7.2 The temporary solution: BCC

The first thing that people started using for addressing this problem is *BPF Compiler Collection* (*BCC*) [5], 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 compiler (which consists of the *Clang-LLVM* combo), 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. First, the Clang-LLVM combo is a big library and resource heavy: this means that you have to deploy large binaries when you distribute your application and the process of compilation can require a lot of time. Second, it must be verified that the system on which the application is going to be installed has the kernel headers present, because BCC-based application do not work on kernels that have been custom built. Last but not least, working in an agile method is quite difficult because compilation errors will appear only at runtime and the application will have to be recompiled and restarted 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 another solution.

2.7.3 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 metadata format which describes the information of C programs and 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*, that takes the BPF object file (the program after its compilation) and triggers the phases of loading and verification;
- The CO-RE compliant kernel.

With BPF CO-RE, eBPF programs are compiled into a more compact, intermediate representation of a binary file that can be loaded and executed on multiple kernel versions and configurations. This reduces the need to recompile programs for different kernel versions, making eBPF programs more portable and efficient. For our purpose, these concepts about BTF are enough: however, anyone who wants to fully understand these notions can visit the Linux kernel documentation page related to BTF [17].

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 a program wants 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.

The last thing that we need to make BPF CO-RE work is the BTF information provided by the running kernel of the target host. In fact, libbpf relies on the kernel to expose its self-describing authoritative BTF information: it does so through

`sysfs` (short for *system file system*) at `/sys/kernel/btf/vmlinux`. `sysfs` is a virtual file system in Linux and other Unix-like operating systems that provides a way to interact with and configure kernel parameters, devices, subsystems, objects and many more kernel-related information. It is mounted at `/sys` in the file system hierarchy. `sysfs` is designed to expose information about the kernel in a structured and hierarchical manner: each device or parameter is represented as a directory or file within the `sysfs` directory tree. These files are also called *virtual*: it means that they are computed on request. Users and programs can read or write to these files to query or configure various aspects of the kernel and its devices thanks to the provided interface which allows to access and manage this information in a structured and standardized way. For our case, we are just interested in the object file `vmlinux`, an Executable and Linkable Format (ELF, a standard file format for executables, object code, shared libraries and core dumps) binary that contains the compiled bootable kernel inside it: when we build Linux, this file is one of the output artifacts and it is also typically packaged with major distributions.

The BTF information for the running kernel can be generated using *bpftool*, a command-line utility that is used to interact with and manage eBPF programs and related components, such as listing programs and maps, loading and unloading programs, attaching and detaching programs to various hook points in the kernel, querying program and map information, showing trace output or debugging information, accessing eBPF type information and many more tasks related to eBPF within the Linux kernel [12]. As many eBPF operations require elevated permissions, administrative privileges are tipically needed to use bpftool on the Linux system. The BTF information for the running kernel can be generated with the following command:

```
1  bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.
    h
```

Listing 2.1: vmlinux.h generation command

The command reads the `vmlinux` object file and generates a `vmlinux.h` header file which contains all kernel types that the installed kernel uses in it's own source code (it is a very large header file). We are going to see later that this file has

to be included in the programs that we are going to develop: by doing so, in our eBPF program we have all internal kernel types and we eliminate dependency on system-wide kernel headers, such as `linux/sched.h`, `linux/fs.h` and many more. In fact, it's pretty common in ebpf programs to read fields of data structures that are used in the kernel through the `BPF_CORE_READ` macro: when we import this header file, our ebpf program can read memory and know which bytes correspond to which fields of any struct that you want to work with. Unfortunately, BTF does not record `#define` macros, so some common macros and constants might be missing with `vmlinux.h`. However, most commonly missing ones might be provided as part of libbpf's kernel-side library.

Since the `vmlinux.h` file is generated from our installed kernel, an ebpf program could break if we try to run it without recompiling on another machine that is running a different kernel version. This is because, from version to version, definitions of internal structs change within the Linux source code (as we already talked about). However, 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 enables portability of eBPF programs by resolving and matching all the types and fields in the kernel and updates necessary offsets and other relocatable data: it uses macros that will analyze what fields we are trying to access in the types that are defined in our `vmlinux.h`. If the field we want to access has been moved within the struct definition that the running kernel uses, the macros will find it for us. Therefore, it doesn't matter if we compile our ebpf program with the `vmlinux.h` file we generated from our own kernel and then ran it on a different one. By doing so, we make sure that the eBPF program's logic is correctly functioning for the specific kernel.

So, the result is that it looks like the program was compiled specifically for the kernel of the target machine, but this happened after getting rid of kernel header dependency and without distributing Clang with our 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). Moreover, BPF CO-RE concept eliminates overhead associated with eBPF development and allows developers to write portable eBPF applications without modifications and runtime source code compilation on the target machine.

2.7.4 BPF CO-RE

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

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 eliminates 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 [8].

2.8 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 regularly 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, etc. 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.

Chapter 3

The eBPF subsystem

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" [25]. In a few words, eBPF is very powerful because it is fast and 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.

3.1 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 be evaluated.

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" [41]): 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 and many more. 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.

In the following we are going to look at the components mentioned above and how they work in practice, including how the program safety verification is done.

3.2 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.

3.3 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, etc.) and tooling (e.g. iproute2) as well as the safety guarantees which the kernel provides.

3.4 Hook points

eBPF programs are event-driven by design and are run when the kernel or an application triggers 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 [57], 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.

3.5 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.

3.5.1 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 ELF file.

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

3.5.2 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 satisfies all the 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 *Directed Acyclic Graph* (*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).

3.5.3 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 (called 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.

3.5.4 JIT compilation

Once the bytecode has been verified and hardened, the eBPF *JIT compiler* is executed. 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 allocation.

3.5.5 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.

3.6 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);
```

Listing 3.1: bpf() system call signature

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 [10] 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 [40], such as the latest Linux kernel code related to this system call [11] or the `bpf.h` header file [9] 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.

3.7 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 the specified eBPF program. 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.

The following describes what happens when a tail call is 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.

3.8 Helper functions

eBPF programs cannot call into arbitrary kernel functions. If this was allowed, eBPF programs would depend on particular kernel versions and would make the compatibility of programs more difficult. 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 [38] and Windows [55]). 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.

3.9 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 [26]. 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 4

eBPF toolchains

eBPF can be addressed with various level of sophistication: anyone can start using eBPF based tools from some package, but writing an entire working eBPF program from scratch is a more complex task because it requires a lot of time to make things work, from installing the libraries to understand all the errors that will for sure occur when you try to compile the program.

In fact, integrating eBPF into modern applications and infrastructures may require experience across different domains. 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 form of their GitHub projects, that we have 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 [27], under the section *Project landscape*, many other applications can be found and there is also a list of projects that represent the current major infrastructure of eBPF.

4.1 eBPF tools

In the course of our research and experimentation, we opted to approach the world of eBPF from a slightly different angle. Rather than diving directly into eBPF tools and low-level programming, we began by leveraging existing frameworks that encapsulate the complexity of eBPF while providing a higher-level interface for achieving specific tracing and monitoring tasks. This choice has been made at the beginning of our work because we wanted to study the state of art of eBPF today. However, using eBPF based tools is the simplest way to approach the world of eBPF: for this reason, they need an honorable mention in this paper.

The easiest way is to learn what tools are available on the distribution that you are using: for example, the Ubuntu 20.04.01 (or later) system is provided with a `bpfcc-tools` package that provides a few binaries to work with eBPF. It is a collection of command-line tools and utilities that leverage the extended eBPF technology which are designed to simplify the process of working with eBPF programs, allowing users to gain insights into various aspects of system behavior, networking and security. The `bpfcc-tools` package is built on top of the *BPF Compiler Collection* (`bpfcc`), which offers a suite of tools for developing, deploying, and managing eBPF programs. It also provides pre-built and ready-to-use tools that cover a wide range of tracing and analysis use cases.

Another list of raw observability tools to get started with eBPF can be found in the book *BPF Performance Tools: Linux System and Application Observability* [15], written by Brendan Gregg in 2019, and in its official GitHub repository [14]: 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.

eBPF tools are easy to use and very powerful, but there are a few characteristics that the beginner has to consider before using them:

- They are commands that have to be invoked on the command line and they need to be provided with some options and arguments (e.g. the events to which the tool has to react);

- You will be likely need to be root when you run the tools because the `bpf()` system call checks for the appropriate capability;
- To stop the tool from running you have to press `CTRL+C` or run them with the timeout command, specifying the time in seconds;
- To write or use a tool you may need to be familiar with kernel data structures or functions because, as we already mentioned in the portability problem, data structures can change based not only on kernel version, but also on kernel configuration and this may make the tool no longer work.

We just mentioned a couple of sources where anyone can find a list of tools to quickly access the benefits of eBPF technology without diving into the complexities of eBPF program development. The most important thing is the fact that they provide a convenient way to access the power of eBPF-based tracing and analysis without needing to write eBPF programs from scratch. By offering a collection of ready-to-use tools, they make the work of the developer, who want to leverage eBPF technology to gain insights into their systems, easier.

4.2 bpftrace

The next step in terms of complexity is to write some simple eBPF scripts: to do so, the easiest way is to use *bpftrace*, one of the two biggest and most popular eBPF projects when it comes to tracing [13]. It uses LLVM as a backend to compile scripts to eBPF-bytecode and it sits on top of BCC for interacting with the Linux eBPF system. However, instead of requiring users to write their own programs against the BCC API, it offers a more expressive higher level syntax.

bpftrace is a powerful dynamic tracing tool for Linux systems that utilizes a specialized tracing language to enable users to observe and analyze various aspects of system behavior (e.g. function calls, system calls and network events) and performance in real-time without the need for modifying or recompiling the kernel. This simple scripting language, available in semi-recent Linux kernels (4.x), is designed to provide a concise and expressive way to create custom tracing scripts without

requiring extensive knowledge of eBPF programming: it supports both "one-liner" commands for quick observations and complete scripts for more elaborate tracing scenarios. It also comes with a collection of pre-built scripts, or "one-liners", that can be used for common tracing tasks or as examples to write more complex tools.

In fact, one of the key advantages of bpftrace is its ease of use. Its high-level scripting language abstracts the complexity of eBPF while still offering a wide range of tracing functionalities and a user-friendly syntax, making it accessible to a broader range of users. It consists of a set of commands, functions and existing Linux tracing capabilities and attachment points that allow users to specify what events they want to trace and how they want to capture and analyze the associated data. This makes it easier to explore and troubleshoot system behavior, diagnose performance issues and gather insights into various aspects within the kernel and user-space applications.

4.3 BCC

To go one step further in writing an eBPF program, we now have to analyze the other biggest and most popular eBPF project when it comes to tracing: *BPF Compiler Collection* (*BCC*), a set of tools and libraries designed for working with eBPF programs in the Linux kernel [6].

This framework was invented before bpftrace: in fact, writing eBPF programs with BCC could be significantly complicated due to the need to keep in mind a lot of assumptions about the way eBPF programs work. However, even though moving from bpftrace to BCC looks like a jump backwards, it is a very important step to do because it will be the beginner's first encounter with writing an eBPF program from scratch.

Actually, BCC simplifies the development, analysis and monitoring of eBPF-based applications by providing a user-friendly interface and a range of utilities. It includes various modules and libraries that allow developers to write, load and manage eBPF programs without needing deep kernel knowledge. In particular, BCC simplifies the process of compilation of an eBPF program from C using LLVM (with a C wrapper around it) as well as the actual mechanics of loading an eBPF program

into the kernel and attaching it to the interested subsystem. BCC also makes eBPF programs easier to write thanks to the provided interface to interact with eBPF consisting of high-level programming languages, such as Python and Lua, which allows programmers to create complex tracing and monitoring tools.

Additionally, BCC provides a set of pre-built tools and examples that can be used for one-off troubleshooting use cases typical of eBPF, such as performance analysis, network traffic control and system introspection. However, it is important to note that much of what BCC uses requires Linux 4.1 and above. For the reasons mentioned above and for the problem of portability of eBPF (as already discussed), BCC is a good choice when writing moderately complex eBPF programs.

4.4 libbpf

Finally, the last step in complexity to enter the eBPF world is to write an eBPF program in C or C++. But before explaining how this can be done, it is crucial to introduce *libbpf*, a critical component within the eBPF ecosystem that provides a user-space C/C++ based library designed to interact with the eBPF subsystem of the Linux kernel [35]. The journey to the release of libbpf was long: it was introduced around 2019 together with a dedicated page in the Linux kernel documentation [39] and nowadays it is still maintained as part of the upstream Linux kernel. The ones that want to read more details about it and look at the major things that were introduced with this library can read the post on Andrii Nakryiko's blog [36].

libbpf plays the role of eBPF program loader, performing complex set up work (relocations, loading and verifying eBPF programs, creating eBPF maps, attaching to eBPF hooks, etc.), letting developers worry only about eBPF program correctness and performance. Such approach keeps overhead to the minimum and eliminates heavy dependencies, making the overall developer experience much more pleasant.

Like BCC, it enables developers to write and manage eBPF programs from user-space applications without needing to deal directly with low-level kernel interfaces. But throughout the years, libbpf received a major boost in capabilities and sophistication and closed many existing gaps with BCC as a library. Actually, there are a

few advantages in using libbpf instead of BCC.

The main thing about libbpf is its role in standardizing and simplifying the development process of eBPF programs. It provides a consistent and stable API that shields developers from the complexities of interacting directly with the eBPF subsystem. In fact, as eBPF became popular across various domains, libbpf ensures a smoother experience for programmers by offering a well-documented and well-maintained library for managing eBPF programs and resources [34], making it more accessible to a wider range of programmers. This library encapsulates various functionalities:

- The loader processes eBPF ELF files generated from the Clang/LLVM compiler and loads eBPF programs into the kernel;
- Program verification and JIT compilation;
- Support for important features not available in BCC such as global variables and eBPF skeletons, an alternative interface to libbpf APIs for working with eBPF objects;
- Ease in the managements of eBPF maps;
- Abstracts the interaction with the `bpf()` system call by providing easy to use library APIs for applications.

Moreover, libbpf is the canonical implementation of eBPF CO-RE, the approach of writing eBPF application that solved the problem of portability of eBPF programs that we already discussed: in fact, it does not require Clang/LLVM runtime being deployed to target servers, it does not rely on kernel-devel headers being available and it does not introduce overhead of performing compilation in runtime on the target host, but it does rely on kernel to be built with BTF type information. For these reasons libbpf is also known as the *eBPF CO-RE runtime library*, as it allows eBPF programs to be compiled once and run across different kernel versions without modification. In fact it addresses this challenge by providing an abstraction layer that allows eBPF programs to be compiled offline into a more compact

representation, which can then be loaded and executed in different kernels, even if the kernels have different versions or configurations. This innovation significantly improves the portability and ease of deployment for eBPF programs. Instead, they can rely on the libbpf library to handle the necessary translation and adaptation of the eBPF programs to the target kernel environment. The result is that developers are allowed to get an eBPF program "custom tailored" to a kernel on target host as if the program was specifically compiled for it.

So, libbpf is the latest and most advanced tool to work with eBPF: we encourage people new to ebpf as well as experienced ones new to this library to become familiar with it. In addition, for BCC's user interested into learning libbpf, there is a practical guide about converting a BCC-based eBPF application to libbpf in another post of Andrii Nakryiko's blog [7].

For the reader's knowledge, during the project of this thesis all the examples of eBPF programs that were used to understand the functioning and the state of art of eBPF both on Linux and on Windows were created using this library. Therefore, after describing the novelties introduced by libbpf, we are now going to overview the management of eBPF programs with this library.

4.4.1 Requirements

Building libbpf-based eBPF application using eBPF CO-RE consists of few steps:

- Generate "vmlinux.h" and include it in the eBPF program file with a few libbpf helper headers to add some missing macros;
- Compile the eBPF program source code with Clang into an object file with extension `.o`;
- Generate eBPF skeleton header file (which we are going to present later) from compiled eBPF object file;
- Include libbpf and skeleton headers in the user-space code to have necessary APIs ready to be used;

- Compile user-space code, which will get eBPF object code embedded in it, so that you don't have to distribute extra files with your application.

How exactly this is done will depend on the specific setup and build system. We are going to present later an environment where all this process is automated.

4.4.2 Program lifecycle

Now that we introduced what we are going to use, it's useful to explain the main libbpf concepts and phases that each eBPF application goes through. An eBPF application consists of a set of eBPF programs, either cooperating or completely independent, and eBPF maps and global variables, shared between all of them to allow the cooperation on a common set of data. Moreover, we already said that eBPF maps and global variables are also accessible from user-space to get or set any extra data necessary.

An eBPF application typically goes through the following phases:

- Open phase: libbpf parses the eBPF object file (generated by the compilation of the eBPF program) to discover maps, programs and global variables, but it does not create them. Before all the entities are created and loaded, user-space applications can make additional adjustments such as setting eBPF program types, pre-setting initial values for global variables, etc.;

- Load phase: libbpf creates eBPF maps, resolves relocations (if any) depending on the kernel version on the machine on which the program will run and verifies and loads the eBPF program into the kernel. However, no program has yet been executed. At the end of this phase, a user-space program can initialize the state of the created eBPF maps before the code execution;

- Attachment phase: libbpf attaches the eBPF program to the designated hook points and then the program can start performing the work it was created for (processing packets, updating maps or variables, etc.);

- Tear down phase: libbpf detaches eBPF programs, unloads them from the

kernel, destroys eBPF maps and frees all the resources used by the eBPF application.

4.4.3 Skeleton files

eBPF skeleton serves as an alternative interface to libbpf APIs, making the interaction with eBPF objects easier. It abstracts the underlying libbpf functionality, allowing to manage eBPF programs in user space in a more friendly way. This file incorporates a compact bytecode representation of the eBPF object, simplifying the distribution of the eBPF code: by embedding the eBPF bytecode directly, the need for additional files when deploying the application binary is eliminated. Moreover, the embedded bytecode representation of the object file ensures that the skeleton and the eBPF object file are always in sync. In fact, only the header file that contains simplified access functions for the eBPF object along with an embedded bytecode representation has to be deployed, avoiding the need to ship a separate eBPF object.

All of this is done to implement the CO-RE principle: by doing this, developers can write and compile eBPF programs using a skeleton on one kernel version and then run them on different kernel versions without the need for significant modifications. By doing so, since skeletons files abstract away kernel-specific details, such as the format of maps, structures or function calls, developers have to focus on the high-level logic of their eBPF programs rather than worrying about kernel-specific differences. Actually, the generated eBPF program is designed to be portable and compatible with various kernel versions, allowing developers to distribute the compiled eBPF program and expect it to work on different systems without modification. The result is that just a single source code has to be maintained for eBPF programs while the compatibility across different kernels is ensured.

The skeleton header file (file with extension `.skel.h`) for a specific object file can be generated by passing the eBPF object to bpftool:

```
1 bpftool [Options] gen COMMAND
2 Options := { { -j | --json } [{ -p | --pretty }] | { -d | --debug
    } | { -L | --use-loader } }
```

```
3  COMMAND := { object | skeleton | help }
```

Listing 4.1: bpftool command syntax

The syntax of bpftool goes beyond the aim of this thesis. However, we recognize that it is a powerful tool for managing and working with eBPF-related resources in the Linux kernel, making it easier for developers and administrators to interact with eBPF programs and maps. We will see later how it is currently used to develop a working eBPF program.

In addition to what we have said so far about the portability of eBPF programs, the generated eBPF skeleton file provides the following custom functions to trigger each phase of the eBPF lifecycle, each of them prefixed with the specific object name:

- `<name>___open()` creates and opens eBPF application;
- `<name>___load()` instantiates, loads, and verifies eBPF application parts;
- `<name>___attach()` attaches all auto-attachable eBPF programs;
- `<name>___detach()` detaches all eBPF programs and frees up all used resources.

Furthermore, the skeleton code makes the memory mapping of global variables as a struct into user space easier, offering a structured interface that enables user space programs to initialize eBPF programs ahead of the eBPF load phase and subsequently manipulate data from user space. Actually, the skeleton file reflects the object file structure by listing out the available maps, programs, etc. and provides direct access to all the eBPF maps and eBPF programs as struct fields. This eliminates the need for string-based lookups with `bpf_object_find_map_by_name()` and `bpf_object_find_program_by_name()` APIs (two methods whose function can be guessed from the name), reducing errors due to disparities between eBPF source code and user-space code.

Chapter 5

Linux development

In the previous chapters we went through the evolution of eBPF throughout the years and we analyzed all the components of its ecosystem. Now we are ready to jump into some coding and write our first eBPF program.

We are going to start to talk about the development process on Linux, since historically it was the first operating system where eBPF was introduced and there is a greater and more complete documentation. In fact, on the internet there are various tutorials and guides on writing your first eBPF program: however, we are going to present just a couple of projects that in our opinion are the best for starting with eBPF because they set up as many things as possible for beginner users to let them dive straight into writing eBPF programs and not get frustrated with various initials setup tasks. Moreover, at the beginning of the history of eBPF it was necessary to work a lot from the Linux terminal for verifying, loading into the kernel and tearing down an eBPF program: however, the projects that we are going to present also simplify this procedure as well.

5.1 Creation of the work environment

The first requirement that we have to be met is that Linux is installed on our machine. For our entire project, we used a computer with Windows 11 installed: this will be 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.

However, since we will have to work on another operating system, we have to take advantage of virtualization to create an isolated environment alongside the Windows host. 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 must be at least the 20.10 because this and the following versions are Linux distributions that come with kernel BTF already built in).

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 5.1 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 [51]. The virtual machine provided a native Linux platform for eBPF program development, compilation and testing.

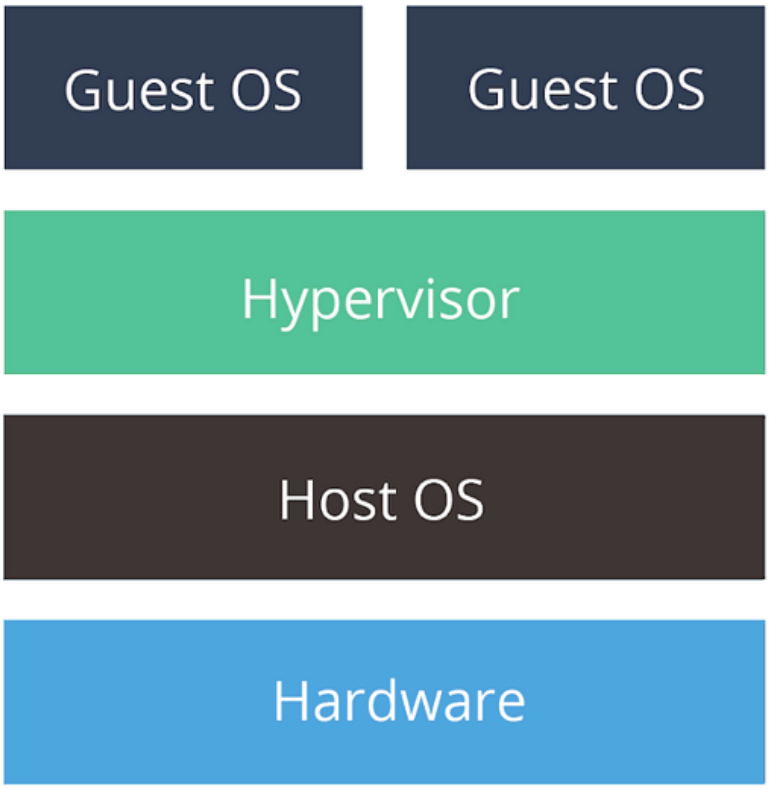

Figure 5.1: Type 2 (or hosted) hypervisor architecture [30].

5.2 Bumblebee

Every time anyone interfaces for the first time with something new, it is always nice to have anything ready with an explanation of what has been done in order to understand the new thing as quickly as possible. This is exactly what happened when we came across *BumbleBee* [20] by *solo.io*, a company known for its work in the field of cloud-native technologies, service mesh and API gateway solutions [48].

BumbleBee is an open-source project focused on simplifying the user experience around building eBPF tools [18]. It helps developers to build, run and distribute eBPF programs using Open Container Initiative (OCI) images, a standardized and portable way to package, distribute and run containerized applications across different container runtimes and platforms typically used in the DevOps and cloud-native ecosystem [44].

By doing so, it allows the developer to focus on writing eBPF code, while taking care of the user space components. We are going to see later that data is automat-

ically exposed data as metrics or logs.

5.2.1 Why BumbleBee

In the previous chapters we understood that eBPF can run sandboxed programs in an operating system kernel to enhance the kernel capabilities with rapidly evolving network and security (to name a few) technologies. Moreover, we went through its subsystem and the solution that was introduce to make eBPF programs portable across different kernel versions, i.e. BTF, a common type descriptor format. However, nowadays packaging and sharing these binary programs is not very well specified. In fact, developers usually write the user-space code and the eBPF program, but they usually do have to figure out on their own how to distribute their application.

This is where BumbleBee comes into play: the idea is to use the same BTF typing to auto-generate all user-space code. In fact, it is a tool that brings a Docker-like experience for automating critical steps in this process: its focus is on packaging, distribution and automatically generating user-space code for any eBPF program. By using a few and simple Command Line Interface (CLI, a text-based user interface used to run programs, manage computer files and interact with the computer) commands it makes developing, running and distributing eBPF programs really simple.

BumbleBee is built using libbpf and allows the developer to focus on writing the eBPF code while automatically taking care of the user space components. Moreover, it detects and displays maps in the program that allow data sharing between user-space and kernel-space programs. Everything is done in complete autonomy by BumbleBee: the trick found in the use of special eBPF conventions and keywords, i.e. maps and functions, the two things that make up ebpf programs.

On the GitHub repository of the project there are a few examples of programs, but now we are going through all the process of developing a working eBPF program from scratch with the help of BumbleBee so we can more deeply understand how this projects works and how it makes the developer's life easier while interfacing for the first time with an eBPF program.

5.2.2 Installation

We presented earlier a way to create a Linux environment with the help of VirtualBox. However, for non-Linux users, the BumbleBee project offers a Vagrant box [19] (with Docker, one of the main software for the portable deployment of application development environments) to help getting started with a Linux environment. For the purpose of this thesis we are going to stick to the use of a virtual machine running the Linux operating system.

Being a Linux technology, eBPF should work in any Linux kernel. However, the developers of BumbleBee suggest to run kernel 5.4 or newer. Moreover,the Linux kernel of the machine must be built with `CONFIG_DEBUG_INFO_BTF=y` configuration because the projects relies on BPF CO-RE and BTF support. A list of all the kernels that already support this configuration and a tutorial on how to build a custom kernel can be found on the libbpf GitHub repository [16].

Once the virtual machine is running, the first thing we need to do to get started with BumbleBee is to install the `bee` tool on our machine. The easiest way to do so is to use the installation script provided by the project which does not even require to clone the repository on the machine. Therefore, we have to open a terminal on our machine and write the following commands:

```
1  sudo apt install curl
2  sudo apt install docker.io
3  sudo -s
4  curl -sL https://run.solo.io/bee/install | sh
5  export PATH=$HOME/.bumblebee/bin:$PATH
```

Listing 5.1: bee installation commands

The first two commands must be performed only the first time the virtual machine is turned on and are important because they install the `curl` command (to make some browser calls from command line) and the package `docker.io`. Instead, from line 3 to 5 there are three commands that must be executed every time the virtual machine is started up. In particular:

- Line 3 is the standard way to give the user elevated privileges (these will be needed to run the `bee` command);

- Line 4 installs the CLI and, in particular, downloads the latest `bee` version which is currently the 0.0.14 (if, for any reason, somebody wants to install a specific version `x`, it has to be specified `BUMBLEBEE_VERSION=v0.0.8 sh` in the command instead of just `sh`);

- Line 5 adds the `bee` CLI to *PATH* (an environment variable that instructs a Linux system in which directories to search for executables and enables the user to run a command without specifying a path).

5.2.3 Write an eBPF program

Now that we have set up all the things that we need, we are ready to create our eBPF program. The first thing that we have to do is to open a terminal and give the user elevated privileges. Then we have to run the interactive command to bootstrap a new eBPF program.

```
1  bee init
```

Listing 5.2: bee init command

It will start a process of creating a program through a series of questions about the eBPF program you plan to build and will auto-generate the code template.. If, for any reason, the process has to be interrupted, it is enough to press `CTRL+C` at any moment.

The first choice that has to be done is about the language with which the program will be developed:

```
1  ? What language do you wish to use for the filter:
2  - C
```

Listing 5.3: bee language selection

Currently only C is supported, but the company has planned the support for Rust as well. In fact, a libbpf library for building eBPF applications in Rust which is called *Libbpf-rs* [37]. We will not enter in the details of this library since we have not used it: it is enough to know that it wraps eBPF functionality Rust-idiomatic interfaces and provides *libbpf-cargo* plugin to handle eBPF code compilation and

skeleton generation. This library makes the building of the user-space part of the eBPF application in Rust easier. However, the eBPF program themselves must still be written in plain C.

Selected the language, the process asks the type of program that is wanted. `bee` has currently two hook points for the program: network or file system. However, since eBPF enables developers to hook a program into any kernel functionality, we can expect that more of them will be added in the future.

```
? What type of program to initialize:
- Network
File system
```

Listing 5.4: bee type of program selection

Network programs will primarily target integration with various functions within the kernel networking stack, whereas file-system programs will interface with file operations, including *open()* calls.

Then, the interface questions about the desired type of global map for the eBPF program that is being built. Once again, eBPF has several types of map, but `bee` currently implements only two of them: `Ringbuffer` and `HasMap`.

```
? What type of map should we initialize:
- RingBuffer
HashMap
```

Listing 5.5: bee map type selection

`RingBuffer` is a generic map type that works as a queue and is usually used for the storage of many arbitrary data types. However, to allow `bee` to take care of all the user-space code, it has been imposed that only one type of data can be stored in a `Ringbuffer`: in fact we will see that a type is stored in the map definition, but this parameter is used by `bee` to correctly parse the data and it is not used in the kernel map definition. `Hashmap`, instead, works as a traditional map with both keys and values. Another substantial difference between the two types of maps is that `Ringbuffer` handles the data only once, while the `Hashmap` keeps its data until it is removed manually.

The last decision that has to be done is about the output format. This is what makes BumbleBee really interesting: normally, to develop an eBPF application, a developer has to write a user-space and a kernel-space program, but `bee` handles automatically the maps data and requires only to write kernel-space code.

```
? What type of output would you like from your map:
- print
counter
gauge
```

Listing 5.6: bee output format selection

`print` tells that the data in the maps will be displayed as text and `print` can be event based (if `RingBuffer` is used as map) or timer based (if `HashMap` is used). On the other hand, `counter` and `gauge` are two types of metrics that are currently supported (as before, new ones could be introduced with the evolution of BumbleBee): the first one is pretty self-explanatory and is used to count the number of times an event occurs, qhile the second one is used to track numeric values that can change over time.

In the end the last thing to do is to give a name to the file.

```
BPF Program File Location: file_name.c
```

Listing 5.7: bee file location

Instead of `file_name` put the name of the file: it will be saved in the directory from which the commands were executed.

If everything was done as explained, the following message will appear on the terminal:

```
Successfully wrote skeleton BPF program
```

Listing 5.8: bee successful program creation message

Now we have created our eBPF program. It must be specified that once the program has been created it is possible to modify it to add a specific functionality to the kernel: however, after the generation of the program, the file will have read-only permissions. So, the first thing that has to be done is to give all permissions to the file using the `chmod` tool.

Once the eBPF program is created, the next step is to compile it. To do so, once again we have to use the `bee` tool.

```
bee build file_name.c name:v1
```

Listing 5.9: bee built command

This command compiles the program and creates an OCI packaged eBPF image thanks to a *docker* build container that simplifies the building of the code. Then, the OCI image ca be shared to popular OCI compliant registries (Docker registry, GitHub Container Repository, or Google Container Repository, etc.), put in a workflow or deployed in a working environment. Once an OCI image is downloaded from somewhere, it will be unpacked into an OCI Runtime filesystem bundle which will be run by an OCI Runtime, according to the Runtime Specification.

Once the process of compilation ends, the following messages will appear on the terminal:

```
Successfully compiled "file_name.c" and wrote it to "file_name.o"
Saved BPF OCI image to name:v1
```

Listing 5.10: bee successful OCI creation messages

Now, under the same directory, we will have the eBPF program and the corresponding object file generated by the process of compilation. The OCI image, instead, will be saved somewhere in the machine.

We want to point out that the name of the eBPF program `file_name` and the one of the OCI image `name` do not have to be the same: however, if the two names correspond, it is easier to understand from which programs different images originated. Another important thing to say is the fact that if a program is modified after it has been compiled, the program has to go through a second process of compilation using the same command show above. However, if the changes to the file are not accepted by the compiler, the compilation gets interrupted and the OCI image of the program before the changes is deleted.

With the following command it is possible to look at all the OCI images stored locally that are ready to be run.

```
bee list
```

Listing 5.11: bee list command

Finally, we can run our program with a simple command.

```
bee run name:v1
```

Listing 5.12: bee run command

5.2.4 Some examples

Now that we understood how to create an eBPF program using BumbleBee, it is time to look at some code. There are seven possible programs that can be created with the process described above:

- Six for the `Network` program type (three output formats for each of the two map types gives in total six possible combinations);
- One for the `File system` program type because it will not ask to choose the map type and the output format.

We are going to show just one example of an eBPF program as it is created by the `bee` tool and use it to discuss the various selection options. The program that we are going to present is created after doing the following choices:

```
INFO  Selected Language: C
INFO  Selected Program Type: Network
INFO  Selected Map Type: RingBuffer
INFO  Selected Output Type: print
INFO  Selected Output Type: BPF Program File Location my_prog.c
```

Listing 5.13: bee choices for first program

This is what the program will look like:

```
#include "vmlinux.h"
#include "bpf/bpf_helpers.h"
#include "bpf/bpf_core_read.h"
#include "bpf/bpf_tracing.h"
#include "solo_types.h"
```

```
6
7   // 1. Change the license if necessary
8   char ___license[] SEC("license") = "Dual MIT/GPL";
9
10  struct event_t {
11    // 2. Add ringbuf struct data here.
12  } ___attribute___((packed));
13
14  // This is the definition for the global map which both our
15  // bpf program and user space program can access.
16  // More info and map types can be found here: https://www.man7.
     org/linux/man-pages/man2/bpf.2.html
17  struct {
18    ___uint(max_entries, 1 << 24);
19    ___uint(type, BPF_MAP_TYPE_RINGBUF);
20    ___type(value, struct event_t);
21  } events SEC(".maps.print");
22
23  SEC("kprobe/tcp_v4_connect")
24  int BPF_KPROBE(tcp_v4_connect, struct sock *sk)
25  {
26    // Init event pointer
27    struct event_t *event;
28
29    // Reserve a spot in the ringbuffer for our event
30    event = bpf_ringbuf_reserve(&events, sizeof(struct event_t), 0)
       ;
31    if (!event) {
32      return 0;
33    }
34
35    // 3. set data for our event,
36    // For example:
37    // event->pid = bpf_get_current_pid_tgid();
38
39    bpf_ringbuf_submit(event, 0);
40
```

```
41     return 0;
42 }
```

Listing 5.14: bee first program

Even though this program can appear really simple, there are many things to look at. First, we have to appreciate how our decisions have structured the code:

- `event_t` is an empty struct (for now);
- `events` is our `RingBuffer` eBPF map;
- `".maps.print"` tells that we have chosen the `print` output format;
- `SEC("kprobe/tcp_v4_connect")` indicates the `Network` program type;
- `BPF_KPROBE` is our eBPF program represented as a normal C function in a specially-named section that will be loaded into the kernel.

Second, we have to understand some things about a generic eBPF program:

- The first five lines are just includes for using eBPF technology (`bpf_helpers.h`, `bpf_core_read.h` and `bpf_tracing.h`), kernel symbols (`vmlinux.h`) and types that bee can automatically interpret and display (`solo_types.h`);
- `SEC("...")` is a macro defined in `bpf_helpers.h` that puts variables and functions into the specified sections;
- `___license` is a variable that defines the license of our eBPF code which is mandatory and enforced by the kernel (some eBPF functionality is unavailable to non-GPL-compatible code);
- The program is attached to `tcp_v4_connect` through a kprobe, which means that every time we do a browser call (by searching anything in the browser or through the `curl` command) the function is triggered;
- The second parameter of the function is a pointer to the struct which contains the information of the packet received when the program triggers the hook point and it must be done in this way because the program can only access

valid memory spaces. The only way to do so is to define through pointers all the header areas of the package which we intend to access with the written code.

Last, we have to look at the tricks that `bee` uses to take care of the user-space code. In fact, this program is not what we will have to write if we were not working with `bee`. There are a couple of things that are only present in programs generated with this tool: `____type` attribute in the map and `".maps.print"`. In fact, we already said that `RingBuffer` is a map that can contain different types of data, but `bee` forces it to storage just one type of data. Moreover, `bee` has to understand what type of output we want and to do so it adds `.print` inside the `SEC` macro of the eBPF map. If we decided to use another output format, we will have `counter` or `gauge` in the place of `print`.

Despite all the efforts that we have made so far for creating and understanding this eBFP program, it does nothing. In fact, if we compile and run the program we will see just a cool interface, but we will not display any information, as shown in Figure 5.2.

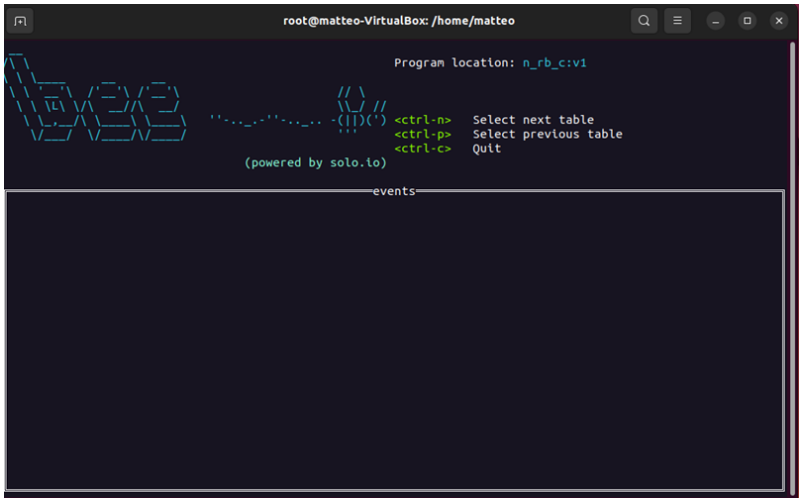

Figure 5.2: BumbleBee first program output.

This is due to the fact that in out program the part where we have to set the data for the event (line 37) is commented. To make things work we have to:

- Uncomment line 37 of our code;

- Add `u32 pid;` inside the `event_t` struct;

`u32` is one of many variable types that is used by `bee` to take care of the user-space code in automatic: in practice, it is just a redefinition of the classical `int` type.

If we re-compile the program and re-run it, we should look at a terminal as displayed in Figure 5.3.

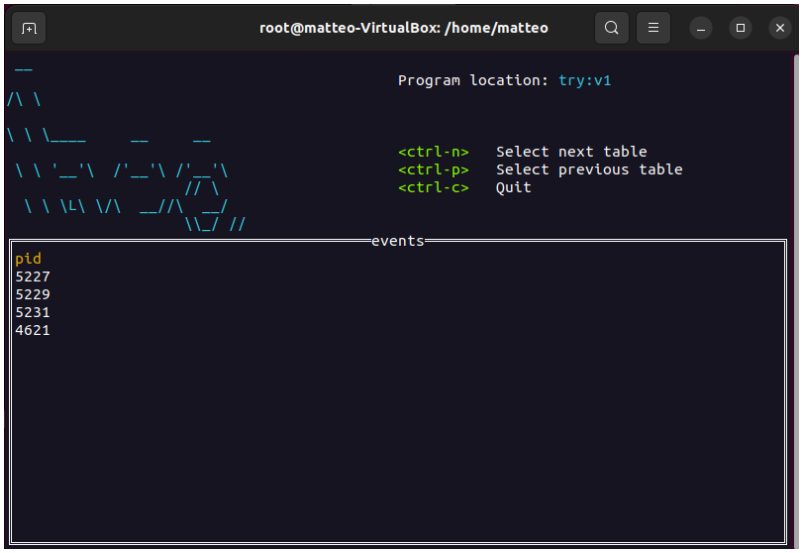

Figure 5.3: BumbleBee modified first program output.

Now we can see how the program really works: every time something happens in the kernel networking stack (e.g we search something on any browser), the eBPF program gets the ID (*pid*) of the process and puts it in the `RingBuffer` with the use of just a few eBPF helpers. Then, the "magic" behind `bee` shows it on the terminal.

With just a couple of changes in the script created by the `bee` tool we managed to develop a working eBPF program that shows us something interesting about the processes in the kernel networking stack.

Many things can be done with the following approach. Now we are going to present a more complex example that contains more things of what we have presented.

```
1  #include "vmlinux.h"
2  #include "bpf/bpf_helpers.h"
3  #include "bpf/bpf_core_read.h"
```

```
#include "bpf/bpf_tracing.h"
#include "solo_types.h"

// 1. Change the license if necessary
char ___license[] SEC("license") = "Dual MIT/GPL";

struct event_t {
  // 2. Add ringbuf struct data here.
  ipv4_addr daddr;
  u32 pid;
} ___attribute___((packed));

struct dimensions_t {
  ipv4_addr daddr;
} ___attribute___((packed));

struct {
  ___uint(type, BPF_MAP_TYPE_HASH);
  ___uint(max_entries, 8192);
  ___type(key, struct dimensions_t);
  ___type(value, u64);
} connection_count SEC(".maps.counter");

// This is the definition for the global map which both our
// bpf program and user space program can access.
// More info and map types can be found here: https://www.man7.
 org/linux/man-pages/man2/bpf.2.html
struct {
  ___uint(max_entries, 1 << 24);
  ___uint(type, BPF_MAP_TYPE_RINGBUF);
  ___type(value, struct event_t);
} events SEC(".maps.print");


SEC("kprobe/tcp_v4_connect")
int BPF_KPROBE(tcp_v4_connect, struct sock *sk, struct sockaddr *
 uaddr) {
```

```
39     struct event_t *event;
40     struct dimensions_t hash_key = {};
41     ___u32 daddr;
42     u64 counter;
43     u64 *counterp;
44 
45     // read in the destination address
46     struct sockaddr_in *usin = (struct sockaddr_in *)uaddr;
47     daddr = BPF_CORE_READ(usin, sin_addr.s_addr);
48 
49     // Reserve a spot in the ringbuffer for our event
50     event = bpf_ringbuf_reserve(&events, sizeof(struct event_t), 0)
   ;
51     if (!event) {
52       return 0;
53     }
54     // 3. set data for our event
55     event->pid = bpf_get_current_pid_tgid();
56     event->daddr = daddr;
57     // submit the event (this makes it available for consumption)
58     bpf_ringbuf_submit(event, 0);
59 
60     // increment the counter for this address
61     hash_key.daddr = daddr;
62     counterp = bpf_map_lookup_elem(&connection_count, &hash_key);
63     if (counterp) {
64       ___sync_fetch_and_add(counterp, 1);
65     } else {
66       // we may miss N events, where N is number of CPUs. We may
   want to
67       // fix this for prod, by adding another lookup/update calls
   here.
68       // we skipped these for brevity
69       counter = 1;
70       bpf_map_update_elem(&connection_count, &hash_key, &counter,
   BPF_NOEXIST);
71     }
```

```
72
73     return 0;
74   }
```

Listing 5.15: bee complex program

The starting point of this program is the same of the previous one, but we have written a few more lines of code to develop a more advanced (but still simple) logic:

- There are two structs that are used for storing two different types of data in two different maps;
- The program uses both a `RingBuffer` and an `HashMap` with two different output formats, respectively `print` and `counter`;

The `RingBuffer` stores `event_t` data which contains the process ID and the destination address (of type `ipv4_addr`) of the network call. The `HashMap`, instead, is a key-value store: the key consists of `dimensions_t` data (which is just the destination address) and the value is a classic `long` variable (redefined as `u64`). The idea of the program is that get the id of the process that makes the browser call and its destination address and you store it in the `RingBuffer`. Then, we increase the number of times that destination address has been searched. To do so, we have to work with the `HashMap` and make use of a few more helpers to interact with a different eBPF map.

- We check in the map if that destination address already exists;
- If so, we increase the value associated to that address by 1;
- If not, we put the new destination address as a new key in the map and its corresponding value is set to 1.

In Figure 5.4 we can see the output that will appear on our terminal after we compile and run the program that we just showed and we make a few browser calls.

Now that we can see the output, things get much more clear. There are two sections delimited by two white rectangles: *connection_count* and *events*, which correspond respectively to the `HashMap` and the `RingBuffer`. In each box we can find the information contained in each map:

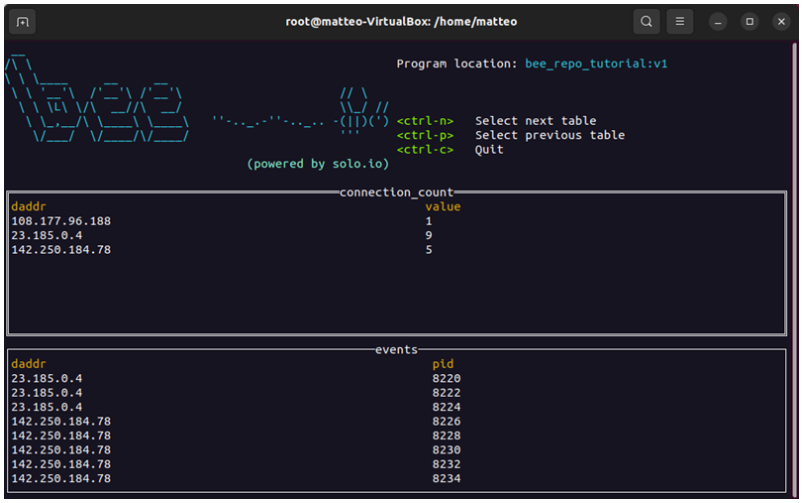

Figure 5.4: BumbleBee complete program output.

- In *connection_ count* we find the destination address of the network call and the number of times this address has been reached;
- In *events* we fin the destination address and the id of the process that made the browser call.

This is just a quick sample of what can be written in a program generated with the `bee` tool. Obviously the possibilities are much more than those shown, but we think that this is a good starting point to delve into the eBPF world.

5.3 libbpf-bootstrap

We have seen that BumbleBee greatly simplifies the development of an eBPF application since it takes care of many things: it generates all the code on which the eBPF program can be built and it manages the user-space part all by itself. However, for the purpose of this thesis, we need to study the development of an eBPF program more thoroughly by writing from scratch both the kernel-side and the user-side code.

Although this may seem difficult, luckily for us it comes to our aid *libbpf-bootstrap*, a project created, among other people, by Andrii Nakryiko and put on GitHub [33]. The idea behind this project is to provide an environment where as many things as possible are set up for beginner users to let them dive straight into

writing eBPF programs in C without worrying about the initial setup. In fact, at this point we might have understood that to give developers lots of power to extend the kernel functionalities without much kernel experience, eBPF requires the setup of a workflow through a series of steps that a new user has to unnecessarily know. libbpf-bootstrap handles this task and provides a convenient workflow with the best eBPF user experience to date.

5.3.1 Installation and overview

Being a GitHub repository, to install this project on out Virtual Machine that runs Ubuntu 22.04 we have to just clone it locally with the following terminal command:

```
1  git clone --recurse-submodules https://github.com/libbpf/libbpf-
    bootstrap
```

Listing 5.16: libbpf-bootstrap clone command

By doing this, we also install the submodules that libbpf-bootstrap requires, such as libbpf to exploit the BPF CO-RE concept and *bpftool* to build skeleton files from the eBPF code. More over, there is a simple `Makefile` that, although it can't be used directly, it's simple enough to just transfer the logic to whichever build system needs to be used.

The only thing that is missing on this repository is the compiler. We already mentioned that to compile a C eBPF program we have to use Clang. Even if Clang 10 should work fine for most eBPF features, the best thing that a user can do is to use the latest possible version: in fact, some features (some of the more recent and advanced CO-RE relocation built-ins) may require version 11 or 12. Furthermore, the system should also have `zlib` and `libelf` packages installed because they are necessary for libbpf to compile and run programs properly. To install all of these dependencies we just have to run the following prompt command:

```
1  sudo apt install clang libelf1 libelf-dev zlib1g-dev
```

Listing 5.17: libbpf-bootstrap install dependencies command

The last thing that we have to check is that since this projects relies on *BTF CO-RE* and BTF kernel support, the Linux kernel has to be built with `CONFIG_DEBUG_INFO_BTF=y`

(as it was before for BumbleBee).

Finally, with just a couple of terminal commands, we are all set up to create and run our first eBPF program from scratch. However, if anybody wants to take a further look inside some working programs, libbpf-bootstrap comes with a series of simple and documented examples both in C and Rust. Moreover, having provided one of the major contributions to this project, Andrii Nakryiko published a post on his blog about the `libbp-bootstrap` environment where it talks about some simple examples in the repository and explains in detail how the `Makefile` works [4].

5.3.2 "Hello world!" with eBPF

Every time anyone has to deal with a new programming language or technology, the first program that is presented is and "Hello world!"-like program. To explain how we can write the kernel-side and the user-side of an eBPF program from scratch and make it work in the libbpf-bootstrap environment, we are going to do exactly what mentioned above.

The one thing that we have to keep in mind when working with libbpf-bootstrap is that both the programs must have the same name: the difference can be seen by the extensions of these files. This is a very important naming convention since it helps the *Makefile* to compile all the eBPF applications. For the following example, we will call our program `example_helloworld`.

First, we show the eBPF C code that contain the logic which is going to be executed in the kernel context: this file has extension `.bpf.c` and has to be created under the `libbpf-bootstrap/examples/c` directory. We also have to create the user-side program with extension `.c` because it will be required in the process of compilation. However, we will keep this file empty (for now).

```
#include "vmlinux.h"
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

SEC("tracepoint/syscalls/sys_enter_execve")
```

```
8  int bpf_prog(void *ctx) {
9    char msg[] = "Hello, World!";
10   bpf_printk("invoke bpf_prog: %s\n", msg);
11   return 0;
12 }
```

example_helloworld.bpf.c

This program is really simple, but if we want to understand how to create a eBPF program from scratch there are a few things to look at:

- All the reported includes are a must for almost every eBPF program: we already greatly discussed about `vmlinux.h`, but usually an application makes use of some basic eBPF-related types and constants necessary for using the kernel-side BPF APIs (included in `linux/bpf.h`) and macros, constants and eBPF helper definitions (included in `bpf_helpers.h` provided by libbpf);
- `SEC(...)` and `LICENSE` are reported for the same reasons as we already discussed for the BumbleBee examples;
- This time the code defines a tracepoint eBPF program which will be called every time an `execve` system call is invoked from any user-space application (in other words, every time we open a new application on our machine).
- `bpf_printk(...)` is the eBPF equivalent to `printf(...)`. The printed messages can be read from a special `/sys/kernel/debug/tracing/trace_pipe` file (we will see later how to display it).

The logic of the program is self-explanatory: every time an `execve` is invoked, the program prints the message `invoke bpf_prog: Hello, World!`. We decided to just use this helper since it is the fastest and most convenient way for debugging a problem in an eBPF code. In fact, eBPF does not have a debugger that does the conventional things (setting a breakpoint, inspecting variables and maps or single-stepping through the code) and to understand what is going on in out eBPF program we just have to rely on logging some information about it. Due to its format, this helper is simple and easy to use, but it is computationally expensive, making it

appropriate only for ad-hoc debugging and unsuitable to be used in production. In reality, `bpf_printk()` is a macro defined by libbpf inside the `bpf/bpf_helpers.h` and internally calls the eBPF helper `bpf_trace_printk()`. We are not going to talk about this anymore: whoever wants more details can visit the Andrii Nakryiko's blog and read the post about this topic [3].

By just the few examples that we have covered, we could say that the trigger of our eBPF program is one of the most important things, if not the most important, because it reflects the hook point to which the program will attach: we could have different programs for different tracepoints or some other kernel events or define multiple programs with the same `SEC(...)` attribute. Moreover, we could declare multiple eBPF programs inside the same eBPF C code and they will share all the global state which is useful when we try to make different programs collaborate. Luckily, on the Linux kernel documentation there is a page with a complete explanation about program types and sections [45]. Furthermore, libbpf exposes a detailed list of expected names on its GitHub repository [46].

Once we have our kernel-side program, the next step that we have to do is to modify the `Makefile` under the same `libbpf-bootstrap/examples/c` directory by adding the name of our program (e.g. `example_helloworld`) after de `APP` variable. We are not going to cover all the detail of this file: it is enough to know that it takes care the process of compilation of this program.

Now it is time to compile our program: to do so, we have to open a terminal with root privileges and once we go in the `libbpf-bootstrap/build` folder of our local copy of libbpf-bootstrap, we have to type the following commands:

```
1  cmake ../examples/c
2  make
```

Listing 5.19: libbpf-bootstrap programs compilation commands

This is all it takes to compile our program. By doing this, in the `libbpf-bootstrap/build` folder a few files will be generated. In particular:

- After the first command, inside the `CMakeFiles` sub-folder, a sub-folder of extension `.dir` containing a series of files that will be used in the process of

compilation will be created for each application specified in the `Makefile` (in our case, we will find the `example_helloworld.dir` sub-folder);

- After the second command, the skeleton header, the object file and the executable file of the program will be generated.

If we want to speed up the process of compilation, we can specify the name of the only application that we want to compile in the second command (in our case, this would be `make example_helloworld`).

Now that we have generated it, it is time to analyze the skeleton header. This is a big file (388 lines in the case of our program) due to the fact that it includes the bytecode representation of our program, so we are going to show only the most important parts and omit the implementations of various methods.

```
1  /* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */
2
3  /* THIS FILE IS AUTOGENERATED BY BPFTOOL! */
4  #ifndef ___EXAMPLE_HELLOWORLD_BPF_SKEL_H___
5  #define ___EXAMPLE_HELLOWORLD_BPF_SKEL_H___
6
7  #include <errno.h>
8  #include <stdlib.h>
9  #include <bpf/libbpf.h>
10
11 struct example_helloworld_bpf {
12   struct bpf_object_skeleton *skeleton;
13   struct bpf_object *obj;
14   struct {
15     struct bpf_map *rodata_str1_1;
16     struct bpf_map *rodata;
17   } maps;
18   struct {
19     struct bpf_program *bpf_prog;
20   } progs;
21   struct {
22     struct bpf_link *bpf_prog;
23   } links;
```

```
24
25      #ifdef __cplusplus
26      static inline struct example_helloworld_bpf *open(const struct
        bpf_object_open_opts *opts = nullptr);
27      static inline struct example_helloworld_bpf *open_and_load();
28      static inline int load(struct example_helloworld_bpf *skel);
29      static inline int attach(struct example_helloworld_bpf *skel);
30      static inline void detach(struct example_helloworld_bpf *skel);
31      static inline void destroy(struct example_helloworld_bpf *skel)
        ;
32      static inline const void *elf_bytes(size_t *sz);
33      #endif /* __cplusplus */
34  };
35
36  ...
37
38  #endif /* __EXAMPLE_HELLOWORLD_BPF_SKEL_H__ */
```

example_helloworld.skel.h

The most important thing about this file is the `example_helloworld_bpf` struct: it contains a series of fields and methods that we can be directly used and accessed in our user-space program. In particular:

- The struct `bpf_object *obj` can be passed to libbpf API functions;
- `maps` contains all the maps defined in the kernel-side code with `SEC(".maps")`, plus the standard `rodata` pointer (we can have more than just one map in a single kernel-side code);
- `progs` and `links` contain the pointers to the eBPF programs defined in our code with `SEC("tracepoint")` which, in our case, is just `bpf_prog` (we can have more than just one program in a single kernel-side code);
- `*bss`, `*data` and `*rodata` are optional structs that can be used to allow direct access to eBPF global variables (they are not present in our skeleton file because we do no declare them). These sections are created if we define

in the kernel-side program, respectively, zero-initialized and mutable or non-zero-initialized and mutable or read-only variables;

- The methods `open_and_load`, `open`, `load`, `attach`, `detach` and `destroy` will be used to perform the now known standard operations on an eBPF program;
- The method `elf_bytes` contains the bytecode representation of the kernel-side program.

Last, we present the user-space C code (remember that we left it empty in the `libbpf-boostrap/examples/c` folder), which loads eBPF code and interacts with it throughout the lifetime of the application. Just for completeness, it is possible to define a `.h` header file with the common type definitions and is shared by both eBPF and user-space code of the application. To keep things as simple as possible, we did not do that.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/resource.h>
4  #include <bpf/libbpf.h>
5  #include "example_helloworld.skel.h"
6
7  static int libbpf_print_fn(enum libbpf_print_level level, const
    char *format, va_list args)
8  {
9    return vfprintf(stderr, format, args);
10 }
11
12 int main(int argc, char **argv)
13 {
14   struct example_helloworld_bpf *skel;
15   int err;
16
17   /* Set up libbpf errors and debug info callback */
18   libbpf_set_print(libbpf_print_fn);
19
20   /* Open BPF application */
```

```
21    skel = example_helloworld_bpf___open();
22    if (!skel) {
23      fprintf(stderr, "Failed to open BPF skeleton\n");
24      return 1;
25    }
26
27    /* Load & verify BPF programs */
28    err = example_helloworld_bpf___load(skel);
29    if (err) {
30      fprintf(stderr, "Failed to load and verify BPF skeleton\n");
31      goto cleanup;
32    }
33
34    /* Attach tracepoint handler */
35    err = example_helloworld_bpf___attach(skel);
36    if (err) {
37      fprintf(stderr, "Failed to attach BPF skeleton\n");
38      goto cleanup;
39    }
40
41    printf("Successfully started! Please run `sudo cat /sys/kernel/
    debug/tracing/trace_pipe` "
42    "to see output of the BPF programs.\n");
43
44    for (;;) {
45      /* trigger our BPF program */
46      fprintf(stderr, ".");
47      sleep(1);
48    }
49
50    cleanup:
51    example_helloworld_bpf___destroy(skel);
52    return -err;
53  }
```

example_helloworld.c

This code does no access to any maps, programs or variables, but it just performs

a series of operation on the eBPF kernel-side code. It is what a eBPF user-side program should look like at its minimum, so it is important to understand how it works:

- Among the includes, `libbpf.h` and the skeleton file are a must to be able to exploit the methods that we used in our code;
- `libbpf_set_print()` provides a custom callback for all libbpf logs which is very useful, especially during active development, because it allows to capture helpful libbpf debug logs (by default, libbpf will log only error-level messages, if something goes wrong, but debug logs are helpful to get an extra context on what's going on and debug problems faster). In this case, it just puts everything in `stdout`;
- Then, we use the auto-generated methods from the skeleton file to open, load and attach the eBPF program (`example_helloworld` in our case). After every operation, we check if everything went well: if so, now the program is waiting in the kernel for any `execve` invocation to then start executing the eBPF code; if not, the `destroy` method will clean up all the resources on both kernel and user side (the kernel usually cleans up resources when an application crashes, but it is a good practice to do it explicitly because some eBPF program types which will stay active in the kernel even if the owner user-space process dies);
- `printf` prints a string that will work as a reminder for the developer on how he can see the debug output of the kernel-side program (we will see how in a short time);
- The endless loop makes sure that our program `example_helloworld` stays attached in the kernel until user kills the process (e.g., by pressing `CTRL+C`). In addition to that, it will invoke the `execve` system call periodically (once a second) through `fprintf()` to monitor internals of the kernel from `example_helloworld` and how the state changes over time.

This code does the basic things that are needed to interact with an eBPF program. We invite anyone who wants to develop its own eBPF application to copy this

code and, after the appropriate changes (e.g. methods and skeleton file names) use it as a starting point to develop his custom user-side program. One simple use is just to access the global variables defined in the kernel-side code: from user-space, they can be read and updated only through the `skel` variable and those updates will be immediately reflected on the eBPF side. In fact, kernel-side global variables are not global variables on the user-space side, but they are just members of the eBPF skeleton's `rodata`, `bss` or `data` members, which are initialized during the skeleton load phase. Declaring exactly the same global variable in eBPF code and user-space code will declare completely independent variables, which won't be connected in any way: to access the kernel-side global variable from user-space we have to use the C pointer notation `skel->skeleton_member->variable_name`.

Now that we have all the components that we need, we can finally run our program and display the output. To do so, the first thing that we have to do is to open two terminals, both with root privileges.

In one of them, we go in the `libbpf-bootstrap/build` folder and we start the execution of our program by typing the following command:

```
1  ./example_helloworld
```

Listing 5.22: libbpf-bootstrap program execution command

If everything was done correctly, we will see a list of messages related to sections and maps of our program and the following string:

```
1  Successfully started! Please run `sudo cat /sys/kernel/debug/
    tracing/trace_pipe` to see output of the BPF programs.
```

Listing 5.23: libbpf-bootstrap program successful execution message

In the other terminal we do exactly what the previous message says and we run the following command (remember to use this terminal with root privileges too): in fact, `bpf_printk()` emits a formatted string to the special file at `/sys/kernel/debug/tracing/trace_pipe` which we can cat to see its contents from the console:

```
1  cat /sys/kernel/debug/tracing/trace_pipe
```

Listing 5.24: libbpf-bootstrap program successful execution message

It is important to say that both of these terminal will continue executing the command until the user types `CTRL+C`.

Now, we can finally see in the second terminal the output of our eBPF program. Every time and `execve` system call is invoked, a new line will appear in the terminal and it will have the following format:

```
1  task_name-task_pid   [CPU_number] options timestamp:
    bpf_trace_printk: invoke bpf_prog: Hello, World!
```

Listing 5.25: bpf_printk output message

To give a better understanding of the output, we have to go through of every detail:

- The `task_name` is the current task name that invoked the `execve` system call (some examples are `nautilus` for the file manager, `google-chrome` and `firefox` for the notorious browsers, etc.);
- The `task_pid` is the process ID of the current task which is represented by a number of four or five digits;
- The `CPU_number` is the number of the CPU on which the task is running;
- The `options` is a set of characters where each of them refers to a set of complex options;
- The `timestamp:` is the timestamp since system boot;
- The `bpf_trace_printk:` indicates the helper that is being used to display the message;
- The `invoke bpf_prog: Hello, World!` is the part controlled by the program consisting in the message that we wanted to print.

5.3.3 A more complex program

Now that we have seen the functioning of the libbpf-bootstrap environment and explained how to debug and eBPF program, we are going to look at a more complex

example that uses an eBPF map and from which we learned some useful things about eBPF helpers and how an eBPF program has to be written.

We would like to underline that the example we are going to show reflects the purpose of this thesis, but has no practical use. In fact, we will see in the next chapter how the same example can be written for Windows, but to make this comparison we had to write the following program using a limited set of helpers due to the fact that their number on Windows is much lower than on Linux.

For this example we are going to show only the kernel-side code as we have already seen the structure and functioning of the skeleton file and the user-side program. In fact, on both of this files there would be just minor changes that we already explained in the previous paragraph: in the skeleton file

- The struct will have a different internal structure because in this kernel-side code a map is declared;
- The pointer to the program will have a different name simply because the name of the program that gets attached to the hook is different;
- The name of the implemented methods in the skeleton file used to perform the various operations (opening, loading, attaching and tearing down) on the eBPF program will be different because the name of the file that contains the eBPF program is different.

Moreover, the user-side program will have to include a different skeleton file and use the methods implemented in it to open, load, attach and destroy the eBPF program.

In the following we can see the kernel-side program.

```
1  #include "vmlinux.h"
2  #include <bpf/bpf_helpers.h>
3  #include <bpf/bpf_tracing.h>
4  #include <bpf/bpf_core_read.h>
5
6  // Set the license of the code
7  char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

```
8
9   // Define the hash map
10  struct {
11    ___uint(type, BPF_MAP_TYPE_HASH);
12    ___uint(max_entries, 256 * 1024);
13    ___type(key, pid_t);
14    ___type(value, u64);
15  } hash_map SEC(".maps");
16
17  static void hash_map_use(pid_t pid, u64 time_stamp){
18    bpf_printk("### BEGIN HASH MAP WORK ###");
19
20    u64 update;
21    u64 delete;
22
23    u64 *found;
24    u64 *deleted;
25
26    update = bpf_map_update_elem(&hash_map, &pid, &time_stamp,
     BPF_ANY);
27
28    bpf_printk("UPDATE %d (update %d).", pid, update);
29
30    bpf_printk("key pid %d - ts %llu - update %d.", pid, time_stamp
     , update);
31
32    found = (u64 *)bpf_map_lookup_elem(&hash_map, &pid);
33
34    if (found) {
35      bpf_printk("FOUND IN HASH MAP (*found %llu).", *found);
36    } else {
37      bpf_printk("NOT FOUND IN HASH MAP");
38    }
39
40    delete = bpf_map_delete_elem(&hash_map, &pid);
41
42    bpf_printk("DELETE %d (delete %lld).", pid, delete);
```

```
43
44    deleted = (u64 *)bpf_map_lookup_elem(&hash_map, &pid);
45
46    if (!deleted) {
47      bpf_printk("DELETED IN HASH MAP (*deleted %llu & deleted %p).
   ", *deleted, deleted);
48    } else {
49      bpf_printk("NOT POSSIBLE TO DELETE IN HASH MAP");
50    }
51
52    bpf_printk("### END HASH MAP WORK ###\n\n");
53  }
54
55  SEC("kprobe/tcp_v4_connect")
56  int print_pid(tcp_v4_connect)
57  {
58    pid_t pid;
59    u64 time_stamp;
60
61    pid = bpf_get_current_pid_tgid() >> 32;
62    time_stamp = bpf_ktime_get_ns();
63
64    bpf_printk("BPF triggered from PID %d at time %d.\n", pid,
   time_stamp);
65
66    hash_map_use(pid, time_stamp);
67
68    return 0;
69  }
```

Listing 5.26: libbpf-bootstrap kernel-side program with maps

What the program does is does not have a specific function, but it is useful only for learning purposes: it exploits a series of helpers (whose descriptions will not be explained, but they can be easily found on the related Linux manual page [38]) to retrieve some data and work with the declared `HashMap`. More specifically:

- Gets the "PID" (or "TGID" in internal kernel terminology) of the process

encoded in upper 32 bits of `bpf_get_current_pid_tgid()`'s return value and the time from system boot from `bpf_ktime_get_ns()`;

- Calls the sub-program `hash_map_use()` with the two previous values as parameters to perform a series of operations on the `HashMap`:
  - Inserts the `time_stamp` value using `pid` as key with the `bpf_map_update_elem()` helper;
  - Looks for the `pid` value in the map using the `bpf_map_lookup_elem()` helper;
  - Deletes the `pid`-`time_stamp` couple through the `bpf_map_delete_elem()` helper;
  - Looks again for the `pid` value in the map using the `bpf_map_lookup_elem()` helper.

The fact that the operations are executed in a sequence ensures that they are always successful: in particular, once inserted the `pid` is found and once deleted it will not be found.

If we follow the same process described for the `example_helloworld` program, thanks to the debugging messages that this program prints, on terminal we should see the following strings (we are reporting only the strings printed by `bpf_printk()`, omitting the initial part of each message that, as we already discussed, contains information about the `task_pid`, `task_name`, etc.):

```
1  BPF triggered from PID ‘‘pid’’ at time ‘‘time_stamp’’.
2
3  ### BEGIN HASH MAP WORK ###
4  UPDATE ‘‘pid’’ (update 0).
5  key pid ‘‘pid’’ - ts ‘‘time_stamp’’ - update 0.
6  FOUND IN HASH MAP (*found ‘‘time_stamp’’).
7  DELETE ‘‘pid’’ (delete 0).
8  DELETED IN HASH MAP (*deleted 0 & deleted 0000000000000000).
9  ### END HASH MAP WORK ###
10
```

Listing 5.27: libbpf-bootstrap more complex program debugging messages

All these messages are used for debugging purposes: `‘‘pid’’` and `‘‘time_stamp’’` are values that depend on the particular execution of the program. In fact, every time we perform an operation on the map, we check if it was successful and then we analyze the returned values:

- If the update and delete operations finish correctly, they return 0;
- If the `pid` search is successful, the lookup method returns a generic pointer (`void *`) to the value associated to the `pid` key (like in the first lookup after the insertion of the `pid`-`time_stamp` couple), otherwise the pointer will be `NULL` (to be more clear, in the code we manually cast the pointer to a `u64 *` because it will point to the `time_stamp` value that is `u64`).

After these explanations, if we read once again the output messages, the control flow of the program becomes clear.

Although we understood that this program does not have a specific practical application, in addition to allowing us to understand how helpers work, we structured it this way to understand further things about eBPF programs:

- The manual cast from `void *` to `u64 *` is not mandatory, but it makes the program more clear. In fact, the generic pointer returned by the lookup method can point to any cell in the memory, but in the case of this program it points (if successful) to a `u64` value;
- We tried to develop the exact same program using an `BPF_MAP_TYPE_ARRAY` and we discovered that the delete operation cannot be performed on this type of map. Therefore, we concluded that not all helpers work with all types of maps and programs;
- All the instructions inside `hash_map_use()` could be written inside the main `print_pid` program and there was no need to write a sub-program. However, by doing so we can understand how they can be used inside an eBPF program. If the type of the sub-program is `void` and we do not use the `static` modifier, it might exceed the maximum size of the eBPF program causing some issues.

This happens because the eBPF verifier cannot check if the program end or not, since the `void` type does not return anything. To solve this, we have two ways:

- Keeping the `void` type and adding the `static` modifier so that the verifier will treat the sub-program as a single big function, since it will be initialized at the beginning of the execution of the program;
- Use `int` as the type of the sub-program and add a return line (e.g. `return 0;`) at its end to help the verifier understand where the program starts and ends.

In conclusion, this highlights that even though the development of eBPF programs can be relatively simple when we have acquired a little bit of experience, there are a few details that we have to take in consideration to avoid annoying errors during the compilation phase.

Chapter 6

Windows development

In the previous chapters, we delved into the world of eBPF and explored its versatile capabilities within the Linux ecosystem. But eBPF is now a cross-platform technology: we are going to continue our journey beyond the confines of Linux to explain the possibilities of eBPF development on the Windows operating system. This chapter serves as a guide for developers and enthusiasts that want to utilize the power of eBPF for Windows-centric applications.

While eBPF is natively integrated into the Linux kernel, recent advancements have extended its reach to the Windows platform, making it accessible to a broader audience. This development opens up new horizons for network monitoring, security and performance analysis within Windows environments.

Since May 2021, Microsoft has been working on bringing eBPF to Windows. In fact, in recent times there have been significant developments in the integration of eBPF on the this platform. As of the time of writing this thesis in September 2023, eBPF for Windows is in a state of rapid evolution and expansion. In this chapter we'll be looking at setting up a Windows-eBPF build environment, followed by developing, running and debugging eBPF programs on Windows, providing insights into its current status and potential future directions.

Unlike what we saw for linux, to date there is only one way to work with eBPF on Windows: it involves using the *ebpf-for-windows* open-source GitHub project by Microsoft [24]. This repository is full of documents that describe how to to get started and use eBPF on Windows. We will make several references to these

documents to avoid making the reading too heavy, but we will highlight the crucial passages. To start working with ebpf on Windows, in fact, the experience is not as user-friendly as it was on Linux where it was enough to clone a repository. This statement is not intended to discourage anyone, but it will immediately be clear that not so much the coding part as the setup of an environment will be very long and complex.

Moreover, we are going to present another project, called *windows-ebpf-starter*, that was created to make the experience of eBPF programming within the Windows ecosystem easier [54]: the result is that this project is for Windows what libbpf-bootstrap is for Linux. However, at the time of writing, the parallelism is not true in every aspect: we are going to present later some issues that we came across while developing eBPF applications.

6.1 Creation of the work environment

In the previous chapter mentioned the fact that we needed a Linux environment in which we could develop various programs: to do so, we used VirtualBox. Now we have to create another environment, this time for Windows, as we will study the state of the art of eBPF on this latest operating system.

Remembering that the computer on which the process was carried out has Windows 11 as its operating system, for this purpose we used the *Hyper-V Console Manager*, a native Windows feature, 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 the need for an underlying host operating system. The illustrative representation of the architecture just described is depicted in Figure 6.1. 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.

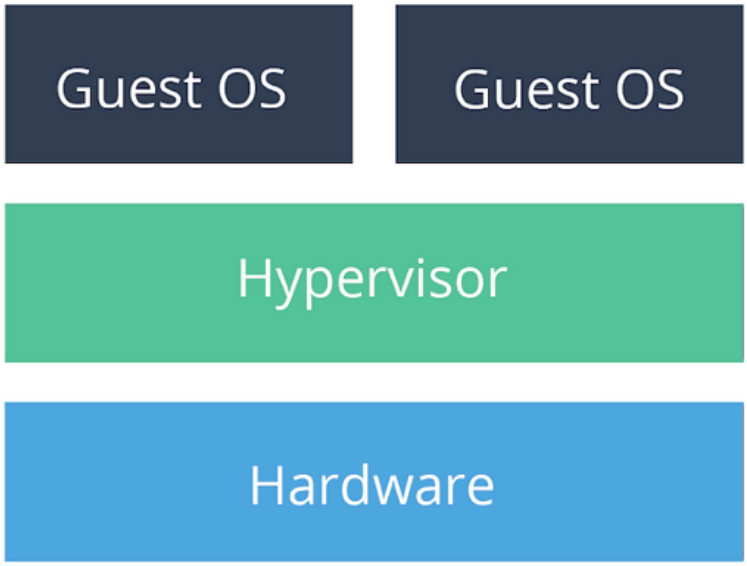

Figure 6.1: Type 1 (or bare metal) hypervisor architecture [30].

The greatest benefits of hypervisors are their robustness and scalability, enabling the efficient virtualization of large-scale applications and services. However, the choice of creating a virtual machine using the Hyper-V Console Manager was dictated by two other reasons:

- The setup instructions described on the ebpf-for-windows GitHub repository tell the user to install a Windows virtual machine;
- The so created isolated Windows 11 development environment provided a controlled space for testing and optimizing eBPF programs on the Windows platform. In fact, if anything goes wrong in this environment, we can just delete the virtual machine and create a new one, while if something bad happens on our host machine, we could break our computer.

So, to install eBPF on Windows the first thing that we have to do is to install our virtual machine. To do so, we have to follow the instructions reported in the *vm-setup.md* document [56]. Besides the fact that that the 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" is the only one that can be selected since

our host computer has Windows 11 as operative system (the tutorial tells to choose the "Windows 10 dev environment" probably because at the time of writing of this document version 10 of Windows was the highest available, but Windows 11 works as well).

After the "one-time setup" procedure is done, we have to decide how we are going to debug our virtual machine. After a careful analysis of the requirements needed to install eBPF we decided to configure a kernel debugging connection over IP address. In fact, since the eBPF for Windows binaries are not yet signed by Microsoft, they will only work on a machine with a kernel debugger attached and running or test signing is enabled. Between the two, we decided to took the first route because it seemed easier. To do so there are a few articles on the Microsoft Learn website, under the documentation section, that we have (once again) to follow by heart. The two things that are worthy of note with this approach are the following:

- On our host computer we have to install a set of debugging tools for Windows. There are a few available [23], but we decided to stick with the classic *WinDbg*, a debugger that can be used to analyze crash dumps, debug live user-mode and kernel-mode code, and examine CPU registers and memory [32]. After following the installation path of this tool, we will find it under `C:\Program Files (x86)\Windows Kits\10\Debuggers\x64`;
- Since it is very likely that sometimes we will shut down our virtual machine, every time that we are going to turn it on we have to start the kernel debugger attached to it (we will present later how to do it). This is quite inconvenient due to the fact that it requires a bit of time every time.

At this point we have all the components that we will need on our host computer. Now we have to install a series of applications on the virtual machine: under the *Prerequisites* of *Building eBPF for Windows* in the *GettingStarted.md* document there is a list of things to install in order to build the repository project [28]. Moreover, to make WinDbg work and debug the virtual machine over IP, we have to install *KDNET*, a debugging feature in Windows that allows remote kernel debugging over a network connection.

Once we have installed all the required tools, we are ready to start debugging our virtual machine. With KDNET we have to set up the target machine (the one we want to debug) and the host machine (the one we will use for debugging) to communicate and then we can start the debugging session: an article on the Microsoft Learn websites tells us what to do [47]. However, even after we have followed all the listed steps the first time, whenever we want to turn on our virtual machine to work with eBPF, we have to redo some of these steps. In particular, we have to:

- Open a *Command prompt* with administrator privileges on both the machines;
- Check the host IP address with the command `ipconfig` because if we let the *Dynamic Host Configuration Protocol* (*DHCP*, a network management protocol used on IP networks for automatically assigning IP addresses and other communication parameters to devices connected to the network using a client–server architecture) to assign automatically an IP address to our computer the address may vary;
- On the virtual machine, in the `C:\KDNET` folder (that we should have created if we followed the last mentioned article) we have to run `C:kdnet <YourIPAddress> <YourDebugPort>`, where the debug port must be within the range 50000-50039. This command will give us another command that we have to copy and run on the host machine. It will look like this: `windbg -k net:port=<YourDebugPort>,key=<YourKey>`, where the key consists of four alphanumeric strings separated by three dots;
- On our host machine we have to:
  - Go to the folder where wh have installed WinDbg, which is `C:\Program Files (x86)\Windows Kits\10\Debuggers\x64`;
  - Run the command that we have copied from the virtual machine.

  After we have run the command, WinDbg will start on our host machine. However, for now, it says "Debuggee not connected". We have to do a couple more steps to make it work;

- Disable *Enhanced session* on the virtual machine using the *View* pull down menu in the VM;

- Restart the virtual machine with the command `shutdown -r -t`. If after we restart the virtual machine one time the "Debuggee not connected" string did not change, we have to restart it a second time. If we do so, we should be able to see "Debugger is running...".

After everything is done we now have started our virtual machine with a kernel debugger attached.

TIGURE BE-
TORE AND
AFTER THE
TURN ON
OF THE DE-
BUGGER

At this point, remember to do the last point on the *vm-setup.md* document which is *Enable Driver Verifier on eBPF drivers*.

The last step that we have to make is to install ebpf-for-windows. To do so, we have to follow the instructions given by the *InstallEbpf.md* document [31]. The easiest way to install eBPF into a test virtual machine is to stick to the so called *Method 1*. For this thesis we worked with the *v0.9.0* version, but at the time of writing versions *v0.10.0* and *v0.11.0* were released. We can understand how fast this technology is evolving on Windows.

Once we have done with all the set up part, we can now start developing some examples using the ebpf-for-windows project. We must point out that from now on we will assume that we are working on a virtual machine that has been turned on with a kernel debugger attached, as we explained previously.

6.2 ebpf for Windows

To develop eBPF applications on Linux we used two projects that made this task relatively simple once we understood the logic behind eBPF. Unfortunately, with ebpf-for-windows this doesn't happen. However, the repository provides several documents in which it explains how to develop simple initial programs and how to debug them.

But before doing so, we have to build our project: to do so, we have to follow the "How to clone and build the project using Visual Studio" section in the

GettingStarted.md document which consists of some operations besides cloning the repository.

[50]

[22]

6.3 windows ebpf starter

Chapter 7

Conclusions

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 03/2023).

[3] *Andrii Nakryiko bpf-printk() guide.* URL: https://nakryiko.com/posts/bpf-tips-printk/ (visited on 03/2023).

[4] *Andrii Nakryiko libbpf-bootstrap guide.* URL: https://nakryiko.com/posts/libbpf-bootstrap/ (visited on 03/2023).

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

[6] *BCC repository.* URL: https://github.com/iovisor/bcc (visited on 03/2023).

[7] *BCC to libbpf practical guide.* URL: https://nakryiko.com/posts/bcc-to-libbpf-howto-guide/ (visited on 03/2023).

[8] *BPF CO-RE post.* URL: https://nakryiko.com/posts/bpf-portability-and-co-re/ (visited on 03/2023).

[9] *BPF header for user-space use.* URL: https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h (visited on 04/2023).

[10] *BPF Linux manual page.* URL: https://man7.org/linux/man-pages/man2/bpf.2.html (visited on 03/2023).

[11] *BPF syscall Linux kernel related code.* URL: https://github.com/torvalds/linux/blob/master/kernel/bpf/syscall.c (visited on 03/2023).

[12] *bpftool GitHub repository.* URL: https://github.com/libbpf/bpftool (visited on 03/2023).

[13] *bpftrace repository.* URL: https://github.com/iovisor/bpftrace (visited on 03/2023).

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

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

[16] *BTF kernel configuration.* URL: https://github.com/libbpf/libbpf#bpf-co-re-compile-once--run-everywhere (visited on 04/2023).

[17] *BTF Linux kernel documentation page.* URL: https://www.kernel.org/doc/html/latest/bpf/btf.html (visited on 03/2023).

[18] *BumbleBee repository.* URL: https://github.com/solo-io/bumblebee/tree/main (visited on 04/2023).

[19] *Bumblebee Vagrant development.* URL: https://github.com/solo-io/bumblebee/blob/main/docs/contributing.md#Development (visited on 04/2023).

[20] *BumbleBee website.* URL: https://bumblebee.io/EN (visited on 04/2023).

[21] *cBPF vs eBPF.* URL: https://www.kernel.org/doc/html/latest/bpf/classic_vs_extended.html (visited on 03/2023).

[22] *Debugging document.* URL: https://github.com/microsoft/ebpf-for-windows/blob/main/docs/debugging.md (visited on 05/2023).

[23] *Debugging tools for Windows.* URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools (visited on 05/2023).

[24] *eBPF for Windows GitHub repository.* URL: https://github.com/microsoft/ebpf-for-windows/tree/main (visited on 05/2023).

[25] *eBPF Linux journal.* URL: https://www.linuxjournal.com/content/bpf-observability-getting-started-quickly (visited on 03/2023).

[26] *eBPF Linux maps documentation.* URL: https://docs.kernel.org/bpf/maps.html (visited on 03/2023).

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

[28] *Getting started document*. URL: https://github.com/microsoft/ebpf-for-windows/blob/main/docs/GettingStarted.md (visited on 05/2023).

[29] *GitHub Logo*. URL: https://allvectorlogo.com/github-logo/ (visited on 02/2023).

[30] *Hypervisors architectures images*. URL: https://medium.com/teamresellerclub/type-1-and-type-2-hypervisors-what-makes-them-different-6a1755d6ae2c (visited on 03/2023).

[31] *Install eBPF document*. URL: https://github.com/microsoft/ebpf-for-windows/blob/main/docs/InstallEbpf.md (visited on 05/2023).

[32] *Install the Windows debugger*. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/ (visited on 05/2023).

[33] *libbpf-bootstrap GitHub repository*. URL: https://github.com/libbpf/libbpf-bootstrap (visited on 04/2023).

[34] *libbpf documentation*. URL: https://libbpf.readthedocs.io/en/latest/api.html (visited on 03/2023).

[35] *libbpf GitHub repository*. URL: https://github.com/libbpf/libbpf (visited on 03/2023).

[36] *libbpf journey Andrii Nakryiko post*. URL: https://nakryiko.com/posts/libbpf-v1/ (visited on 03/2023).

[37] *libbpf Rust GitHub repository*. URL: https://github.com/libbpf/libbpf-rs (visited on 04/2023).

[38] *Linux helpers manual page*. URL: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html (visited on 03/2023).

[39] *Linux kernel documentation libbpf page*. URL: https://www.kernel.org/doc/html/latest/bpf/libbpf/index.html (visited on 03/2023).

[40] *Linux kernel repository*. URL: https://github.com/torvalds/linux/tree/master (visited on 03/2023).

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

[42] *Master thesis repository.* URL: https://github.com/Matteo-Locatelli/master_thesis (visited on 02/2023).

[43] 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.

[44] *OCI repository.* URL: https://github.com/opencontainers/image-spec (visited on 04/2023).

[45] *Section Linux kernel documentation page.* URL: https://docs.kernel.org/bpf/libbpf/program_types.html#program-types-and-elf (visited on 03/2023).

[46] *Section list on libbpf GitHub repository.* URL: https://github.com/libbpf/libbpf/blob/787abf721ec8fac1a4a0a7b075acc79a927afed9/src/libbpf.c#L7935-L8075 (visited on 03/2023).

[47] *Setting up network debugging of a Hyper-V virtual machine.* URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-network-debugging-of-a-virtual-machine-host (visited on 05/2023).

[48] *solo.io website.* URL: https://www.solo.io/ (visited on 04/2023).

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

[50] *Tutorial document.* URL: https://github.com/microsoft/ebpf-for-windows/blob/main/docs/tutorial.md (visited on 05/2023).

[51] *Ubuntu ISO image download page.* URL: https://www.ubuntu-it.org/download (visited on 03/2023).

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

[53] *Unibg website.* URL: https://www.unibg.it/ (visited on 02/2023).

[54] *Windows eBPF starter GitHub repository.* URL: https://github.com/SubconsciousCompute/windows-ebpf-starter/ (visited on 05/2023).

[55] *Windows helpers manual page*. URL: https://microsoft.github.io/ebpf-for-windows/bpf__helper__defs_8h.html (visited on 05/2023).

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

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