Learning eBPF

High performance observability, networking, and security programming on Linux

Michael Kehoe

www.bpbonline.com

First Edition 2025

ISBN: 978-93-65898-859

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true and correct to the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but the publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Dedicated to

My Mum, Dad, and brother for their unwavering love, support, and belief in me

About the Author

Michael Kehoe is a distinguished author, speaker, and senior staff cloud and reliability architect at Confluent. In his current role, he is spearheading a comprehensive initiative to revamp the company's cloud platform.

Previously, as a senior staff **site reliability engineer** (**SRE**) at LinkedIn, Michael played a pivotal role in orchestrating LinkedIn's seamless transition to the Microsoft Azure platform. His expertise in reliability engineering was instrumental in ensuring the platform's stability and performance. During his tenure at LinkedIn, Michael led critical initiatives in incident response, disaster recovery, visibility engineering, and reliability principles. He was also embedded with the profile, traffic, and espresso (KV Store) teams. Having successfully overseen the construction of LinkedIn's final physical data center, Michael assumed a pivotal role in shaping the infrastructure blueprint for LinkedIn's expansion into Microsoft Azure.

About the Reviewers

- **Yusheng Zheng** is a software engineer, researcher, and open-source advocate with a strong passion for eBPF and **large language models** (**LLMs**). As the creator of the eunomia-bpf project, Yusheng has been actively involved in advancing lightweight eBPF development frameworks, exploring their potential beyond Linux and integrating them with emerging technologies. She has spoken at major conferences, including KubeCon, eBPF Summit, and the Linux Plumbers Conference. When not immersed in code, she shares her insights and learnings on her blogs and enjoys collaborating on innovative projects with fellow developers and researchers.

- **Hudson Coutinho:** Working as a DevOps engineer since 2018, he has participated in strategic projects for large national and international companies, dedicating myself to building and improving robust and scalable DevOps architectures.

  Throughout their career, they have executed several migrations to the cloud, created environments in Kubernetes, created micro-services in Docker, implemented complete CI/CD pipelines, and optimized internal processes, always with a keen eye for efficiency and security.

  As head of DevOps, Hudson has led DevOps teams and multidisciplinary teams, guiding them to achieve results that are always based on agile principles and focused on automation, seeking continuous improvement.

Acknowledgement

First and foremost, I extend my heartfelt appreciation to my family and friends for their unwavering support and encouragement throughout this journey. Their love and encouragement have been a constant source of motivation.

I am immensely grateful to BPB Publications and the team for their guidance and expertise in bringing this book to fruition. Their support and assistance were invaluable in navigating the complexities of the publishing process.

I would also like to acknowledge the reviewers, technical experts, and editors who provided valuable feedback and contributed to the refinement of this manuscript. Their insights and suggestions have significantly enhanced the quality of the book.

Last but not least, I want to express our gratitude to the readers who have shown interest in this book. Your support and encouragement have been deeply appreciated.

Thank you to everyone who has played a part in making this book a reality.

Preface

In today's complex computing environment, having a deep understanding of system behavior is a necessity. Maintaining performance, efficiency, security, and reliability, demands deep insights into the inner workings of our infrastructure. Traditional monitoring and debugging tools, while useful, are often unhelpful when faced with the scale and intricacy of modern systems. These tools often operate at a high level, providing aggregate metrics that do not provide information about individual processes, network interactions, or kernel events. They can also introduce significant overhead, impacting the very systems they are supposed to observe. This is where **extended Berkeley Packet Filter** (**eBPF**) fills these gaps, offering a revolutionary approach to system observability.

eBPF is the evolution of the **classic Berkeley Packet Filter** (**cBPF**) which was originally created in the 1990's. It allows for user-defined, sandboxed bytecode to be executed by the kernel. eBPF represents a paradigm shift in how we can interact with and use the kernel, opening up unprecedented possibilities for innovation in areas such as observability, networking, and security. This book will move beyond theoretical concepts and demonstrate the practical applications of eBPF, providing concrete examples of how to leverage eBPF for a wide range of applications.

Divided into ten chapters, this book provides a complete guide to the eBPF ecosystem. We start by exploring the foundations of classic BPF, understanding its history, and examining its core architecture. From there, we observe the evolution of eBPF, detailing its history, features, and the key advancements that distinguish it from its predecessor.

Chapter 3 provides a deep dive into eBPF programming concepts, covering essential elements such as the bpf() system call, program types, attach types, maps, and helper functions. Chapter 4 then explores the diverse array of libraries and frameworks available for

eBPF development, ranging from libbpf and BCC to language-specific options like ebpf-go and Aya.

Chapter 5 guides you through writing your first eBPF programs using different programming languages and frameworks, providing hands-on experience and practical examples. Chapter 6 explains the crucial aspects of eBPF portability and deployment, introducing **BPF Type Format** (**BTF**) and **Compile Once, Run Everywhere** (**CO-RE**) for building and deploying eBPF programs at scale.

The next three chapters take the theoretical knowledge that has been learned so far and demonstrate the practical applications of eBPF. Chapter 7 focuses on eBPF observability, focusing on tracing and analyzing system behavior with minimal overhead. Chapter 8 explores eBPF networking, covering a wide range of program types and their use cases in multiple levels of the Linux network stack including load-balancing, traffic shaping, and socket filtering. Chapter 9 examines the security applications of eBPF, discussing how it can be leveraged for implementing controls and monitoring system activity.

Finally, Chapter 10 provides a look into the future of eBPF, detailing the growing open-source ecosystem, standardization efforts, and emerging trends.

Whether you are a software engineer, a network engineer, a security professional, or simply curious about this emerging technology, This book will be a valuable resource for anyone seeking to understand, explore, and master the eBPF ecosystem.

Chapter 1: Classic Berkeley Packet Filter - This chapter provides a comprehensive introduction to the **classic Berkeley Packet Filter** (**cBPF**), tracing its evolution from its origins in the 1990s as a high-performance network filtering tool. It explores the architecture of cBPF, including its components like the network tap, packet filter, and pseudo-machine, and how they interact to efficiently process network packets. The chapter also delves into the implementation of cBPF in Linux, providing code examples and demonstrating its original uses. It also discusses early applications of cBPF such as tcpdump, and the

modernization efforts like JIT compilation and seccomp-bpf that laid the groundwork for the emergence of **extended BPF** (**eBPF**).

Chapter 2: Extended Berkeley Packet Filter - This chapter explores the evolution of the **Berkeley Packet Filter** (**BPF**) to eBPF. It details the history of eBPF, highlighting the challenges and motivations behind its development. The chapter explores the key features of eBPF, emphasizing its efficiency, versatility, and safety advantages over traditional kernel modules. It also provides a comparative analysis of eBPF and cBPF architectures, outlining the advancements in the instruction set, register size, and program loading capabilities. The chapter concludes by examining the differences in virtual machine implementations, the role of the eBPF verifier and JIT compiler, and the functionalities of eBPF helpers and maps.

Chapter 3: eBPF Programming Concepts - This chapter provides an overview of eBPF programming concepts, focusing on the key elements involved in writing eBPF programs. It begins by introducing the bpf() system call, the primary interface for user-space interaction with the eBPF subsystem. It then explains each eBPF program type, exploring the different categories and their specific purposes, along with the corresponding attach types that determine where these programs hook into the kernel. The chapter also examines eBPF maps, detailing their role as efficient key-value stores for inter-process communication, and the various map types available. Additionally, it explores eBPF helper functions and discusses other program primitives such as loops, tail calls, and return codes.

Chapter 4: eBPF Programming Libraries and Frameworks - This chapter explores various libraries and frameworks available for eBPF program development. It discusses the advantages and disadvantages of writing raw BPF bytecode and then introduces libbpf, a core library offering high-level and low-level APIs for interacting with eBPF programs and maps. The chapter also covers the integration of eBPF with the perf profiling tool, enabling advanced tracing capabilities. Furthermore, it examines BCC, a popular framework with a rich collection of tools and examples, and bpftrace, which provides a high-level language for simplified eBPF program creation. Additionally,

the chapter explores several Go and Rust libraries like gobpf, ebpf-go, libbpfgo, libbpf-rs, and Aya, offering diverse options for eBPF development in different programming languages. Finally, it touches upon eBPF for Windows, demonstrating the growing cross-platform support for eBPF.

Chapter 5: Writing Your First eBPF Program - This chapter provides a practical guide to writing your first eBPF programs using various programming languages and frameworks. It starts by outlining the necessary steps to set up your development environment, including configuring kernel settings for eBPF functionality. Then, it dives into programming with BCC in Python, demonstrating how to write a simple "Hello World" program using kprobesand a more complex example utilizing maps and helper functions to count syscalls. The chapter also explores writing eBPF programs in C with libbpf, showcasing the process of creating, loading, and attaching BPF programs, along with using maps and helpers. Furthermore, it provides examples of eBPF development in Go using ebpf-go and in Rust using libbpf-rs, highlighting the unique features and functionalities of each framework. Finally, the chapter concludes with a discussion on best practices for writing efficient and safe eBPF programs.

Chapter 6: eBPF Portability and Deploying - This chapter focuses on the practical aspects of deploying eBPF programs in production environments, with a particular emphasis on portability and scalability. It introduces **BPF Type Format** (**BTF**), a metadata format crucial for program introspection and portability, and **Compile Once, Run Everywhere** (**CO-RE**), a technology that allows eBPF programs to be compiled once and run across different kernel versions without recompilation. The chapter also provides an overview of bpftool, a command-line utility for managing and interacting with eBPF programs and maps. It then delves into different deployment approaches, contrasting the naive method using BCC with the more robust CO-RE-based approach for production systems. Finally, the chapter discusses deployment frameworks like systemd and bpfman, highlighting their features and capabilities for managing eBPF programs at scale, and

concludes by emphasizing the importance of feature compatibility, privilege management, unit testing, and staggered deployments for successful production implementation.

Chapter 7: eBPF Observability - This chapter explores the use of eBPF for observability, detailing how its high-performance, low-overhead characteristics enable deep insights into kernel and application behavior. It introduces various eBPF program types designed for observability, including kprobes, uprobes, tracepoints, and perf events, each with its own strengths and use cases. The chapter also examines libbpf tracing macros that simplify eBPF program development and discusses the advantages and disadvantages of using eBPF for tracing compared to other methods. Finally, it provides guidance on selecting the appropriate eBPF program type based on specific needs and performance considerations.

Chapter 8: eBPF Networking - This chapter provides a comprehensive overview of eBPF's applications in networking, showcasing its versatility and capabilities in enhancing network functionality and performance. It explores 18 different eBPF program types, each designed to address specific networking tasks, ranging from socket filtering and traffic control to XDP programming and segment routing offering practical guidance on their usage. It also provides code examples and further references to aid in understanding and implementing these program types effectively.

Chapter 9: eBPF Security - This chapter focuses on the application of eBPF for security purposes, exploring its ability to provide comprehensive system visibility and implement security controls. It introduces various eBPF program types designed for security monitoring and enforcement, such as controlling cgroup device access, managing sysctl parameters, filtering network traffic within cgroups, and implementing **mandatory access control** (**MAC**) policies. The chapter also examines the strengths of eBPF as a security tool and discusses popular open-source eBPF security projects like Falco, Tetragon, Suricata-eBPF, and Pulsar, highlighting their functionalities and contributions to enhancing system security.

Chapter 10: eBPF Open Source Projects and the Future of eBPF - This chapter explores the growing open-source landscape surrounding eBPF, highlighting key projects and future trends. It begins by examining various language-specific projects that facilitate eBPF program development in C, Go, Python, Rust, and WebAssembly. The chapter then delves into notable open-source projects that leverage eBPF for observability, networking, and security, showcasing the versatility and impact of eBPF across different domains. Finally, it discusses the future of eBPF, including standardization efforts, security enhancements, and the expansion of eBPF to new platforms like Windows, emphasizing the continued growth and evolution of the eBPF ecosystem.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/jbp4kb7

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Learning-eBPF**.
In case there's an update to the code, it will be updated on the existing GitHub repository.
We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :
errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Classic Berkeley Packet Filter 1

Introduction 1

Structure 2

Objectives 2

Introduction to BPF 2

Before BPF, a history *2*

BPF architecture 3

Network tap *4*

Filter *4*

libpcap *6*

BPF pseudo-machine *6*

Instruction set and addressing modes *7*

Features of BPF 9

BPF on Linux 10

Early BPF usage 14

Modernizing BPF before eBPF 15

JIT *15*

Seccomp-BPF *15*

BPF+ *18*

Conclusion 20

2. Extended Berkeley Packet Filter 21

Introduction 21

Structure 21

Objectives 22

Introducing eBPF 22

History of eBPF 22

Need of eBPF 24

Features of eBPF 25
eBPF use cases 26
eBPF concepts 26
eBPF architecture 28
Differences between BPF and eBPF 29
Virtual machine *30*
eBPF verifier *32*
eBPF JIT compiler *33*
eBPF helpers *33*
eBPF maps *33*
Conclusion 34

3. eBPF Programming Concepts 35
Introduction 35
Structure 36
Objectives 36
bpf() system call 36
int cmd *37*
*union bpf_attr *attr* *40*
unsigned int size *41*
Return values *41*
Error numbers *42*
eBPF program types 43
BPF attach types 47
Map types 49
BPF_MAP_TYPE_HASH *50*
BPF_MAP_TYPE_ARRAY *51*
BPF_MAP_TYPE_PROG_ARRAY *51*
BPF_MAP_TYPE_PERF_EVENT_ARRAY *51*
BPF_MAP_TYPE_PERCPU_HASH *51*
BPF_MAP_TYPE_PERCPU_ARRAY *51*

BPF_MAP_TYPE_STACK_TRACE 52
BPF_MAP_TYPE_CGROUP_ARRAY 52
BPF_MAP_TYPE_LRU_HASH 52
BPF_MAP_TYPE_LRU_PERCPU_HASH 52
BPF_MAP_TYPE_LPM_TRIE 52
BPF_MAP_TYPE_ARRAY_OF_MAPS 52
BPF_MAP_TYPE_HASH_OF_MAPS 53
BPF_MAP_TYPE_DEVMAP 53
BPF_MAP_TYPE_SOCKMAP 53
BPF_MAP_TYPE_CPUMAP 53
BPF_MAP_TYPE_XSKMAP 53
BPF_MAP_TYPE_SOCKHASH 54
BPF_MAP_TYPE_CGROUP_STORAGE 54
BPF_MAP_TYPE_REUSEPORT_SOCKARRAY 54
BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE 54
BPF_MAP_TYPE_QUEUE 54
BPF_MAP_TYPE_STACK 54
BPF_MAP_TYPE_SK_STORAGE 55
BPF_MAP_TYPE_DEVMAP_HASH 55
BPF_MAP_TYPE_STRUCT_OPS 55
BPF_MAP_TYPE_RINGBUF 55
BPF_MAP_TYPE_INODE_STORAGE 55
BPF_MAP_TYPE_TASK_STORAGE 55
BPF_MAP_TYPE_BLOOM_FILTER 56
BPF_MAP_TYPE_USER_RINGBUF 56
Map-specific helpers 56
Other BPF helpers 57
Program arguments 59
Loops 59
Tail calls 60
Sleepable programs 60

Program return codes 61
bpftool 61
Conclusion 62

4. eBPF Programming Libraries and Frameworks 63
Introduction 63
Structure 63
Objectives 64
BPF bytecode 64
C and libbpf 65
Perf 66
BCC 67
bpftrace 67
ply 68
gobpf 68
ebpf-go 69
libbpfgo 69
eBPF for Windows 70
libbpf-rs 70
Aya 71
BumbleBee 71
eunomia-bpf 71
bpftime 72
Conclusion 72

5. Writing Your First eBPF Program 73
Introduction 73
Structure 73
Objectives 74
Setting up your development environment 74
Programming in Python with BCC 75

Installing prerequisites *75*
Programming with BCC *76*
Displaying data *76*
Writing your first program with BCC *77*
Using maps and helpers with BCC *78*
Writing eBPF programs with C and libbpf 80
Installing prerequisites *80*
Writing your first program with libbpf *81*
Using maps and helpers with libbpf *83*
Writing eBPF Go programs with ebpf-go 87
Install prerequisites *87*
Writing your first program *88*
Using maps and helpers with ebpf-go *90*
Installing prerequisites *92*
Writing your first libbpf-rs program *92*
Using maps and helpers with libbpf-rs *94*
BPF headers 96
Best practices 97
Conclusion 97

6. eBPF Portability and Deploying 99
Introduction 99
Structure 99
Objectives 100
BPF Type Format 100
CO-RE 101
Reading data in a BPF CO-RE application *102*
Handling kernel changes and feature mismatch *103*
BPFtool 103
Loading and managing BPF programs *104*
Querying BPF programs *105*

Inspecting BPF maps ... *106*
Verifying BPF programs ... *107*
Dumping BPF program disassembly ... *107*
Attaching BPF programs ... *108*
BPF deployment approaches ... 108
The naive method ... *108*
A better way ... *109*
BPF deployment frameworks ... 109
systemd ... *110*
RestrictFileSystems ... *110*
RestrictNetworkInterfaces ... *110*
IPIngressFilterPath and IPEgressFilterPath ... *110*
DeviceAllow ... *111*
BPFProgram ... *111*
Deploying your application with bpftool and systemd ... *111*
bpfman ... *112*
greggd ... *112*
Notes for fleet-wide deployment ... 113
Feature compatibility ... *113*
Privileges and compatibility ... *113*
CAP_BPF ... *114*
BPF program types and capatabilities ... *114*
Unit testing ... *116*
Staggered deploys ... *121*
Conclusion ... 122

7. eBPF Observability ... 123
Introduction ... 123
Structure ... 123
Objectives ... 124
Introduction to eBPF observability ... 124

Observability program types 125
Using eBPF for tracing 125
libbpf tracing macros 126
BPF_PROG 126
BPF_PROG2 127
BPF_KPROBE 127
BPF_KRETPROBE 127
BPF_KSYSCALL and BPF_KPROBE_SYSCALL 128
BPF_UPROBE 128
BPF_URETPROBE 128
BCC macros 128
Kprobe programs 128
Kprobe/Kretprobe 129
ksyscall/kretsyscall 132
uprobe/uretprobe 132
USDTs 133
kprobemulti/ kretprobemulti 135
Tracepoint programs 135
Raw tracepoint programs 138
Raw tracepoint writeable programs 139
Perf event programs 140
Picking the right program type 142
Conclusion 142

8. eBPF Networking 143
Introduction 143
Structure 143
Objectives 144
Introduction to eBPF network programmability 144
Socket filter programs 146
Traffic Classifier programs 148

Traffic classifier action programs 151
Writing programs *152*
XDP programs 153
cgroup socket programs 157
Lightweight tunnel programs 158
Segment routing programs 161
Socket option programs 163
Socket SKB programs 165
Socket message programs 168
cgroup socket address programs 170
Socket reuseport programs 172
Flow dissector programs 174
cgroup socket option programs 176
Socket lookup programs 177
Netfilter programs 180
Conclusion 182

9. eBPF Security 183
Introduction 183
Structure 183
Objectives 184
Introduction to eBPF security tooling 184
eBPF security program types 185
cgroup device controls 185
Monitoring cgroup sysctl controls 187
Firewalls for container networks 190
BPF for Linux Security Modules 191
Open source eBPF security projects 193
Conclusion 194

10. eBPF Open Source Projects and the Future of eBPF 195

Introduction..... 195

Structure..... 195

Objectives..... 196

Introduction to eBPF's open source projects 196

Language projects..... 196

Notable open source projects..... 198

Observability 198

Networking 199

Security..... 199

The future of eBPF..... 200

eBPF foundation 200

Standardization (IETF) 201

Signing programs 201

BPF firewalls 202

eBPF security..... 203

eBPF on Windows 203

Conclusion..... 204

Index 205-210

Chapter 1
Classic Berkeley Packet Filter

Introduction

Berkeley Packet Filter (**BPF**) (originally known as **Berkeley Software Distribution** (**BSD**) Packet Filter) is a framework originally built around high-performance network filtering and packet capture. It utilizes a **central processing unit** (**CPU**) register-based filter evaluator that runs as a pseudo-machine inside a Unix-based kernel.

Today, the **extended Berkeley Packet Filter** (**eBPF**) is much more than that. It provides low-level observability and high-performance network capabilities on Linux, macOS, and Windows systems, becoming an essential tool for network administrators, security researchers, and software developers. Before we jump into eBPF, it is important to understand the underlying mechanisms behind eBPF, and its evolution over time.

This chapter will introduce BPF, examine the events that led up to its creation in 1992, and explain the architecture of how it works. In this section, we will exclusively evaluate the original implementation of BPF, now known as the **classic Berkeley Packet Filter** (**cBPF**).

Structure

In this chapter, we will go through the following topics:

- Introduction to BPF
- Before BPF, a history
- BPF architecture
- BPF on Linux
- Early BPF usage
- Modernizing BPF before eBPF

Objectives

In this chapter, you will learn the beginnings of cBPF and why it was created, as the architecture of the original BPF **virtual machine** (**VM**). Then you will learn how to write basic cBPF packet filter programs on Linux as well as write basic Seccomp-BPF programs. Finally, you will learn about the efforts to modernize cBPF before the eventual creation of eBPF.

Introduction to BPF

If you think you are new to cBPF, you may not realize that you have likely used cBPF before in the form of `tcpdump` filters. The BPF specification provides a raw interface (as opposed to copying packets across kernel / user-space boundaries) to data link layers (Layer 2 of the OSI model) in a protocol-independent fashion. All packets on the (L2) network, even those destined for other hosts, are accessible through this mechanism. cBPF allows running user space code (filters) against raw network interfaces inside a sanity-checking virtual machine.

BPF was first implemented in BSD as the BSD Packet Filter, it was later implemented in Linux which was known as Linux Packet Filter or Berkeley Packet Filter. Once BPF was optimized and extended in 2014, the original implementation became known as classic BPF, or cBPF for short.

Before BPF, a history

As *Steven McCanne* and *Van Jacobson*, the creators of BPF note in their 1992 paper *A New Architecture for User-level Packet Capture* (**https://www.tcpdump.org/papers/bpf-usenix93.pdf**), until the creation of

BPF, each flavor of Unix (NIT, SunOS, Uiltrix, SGI were popular at the time) provided different facilities for kernel packet filtering. Even with the significantly slower network speeds in the early 1990s, these implementations were still considered sub-optimal and warranted improvements.

McCanne and *Van Jacobson's* USENIX paper in 1992 described BSD Packet Filter which was a new kernel architecture for packet capture. BPF offered significant performance improvements over the existing packet capture facilities (see *Figure 8* of the paper).

The original BPF implementation (known as BSD Packet Filter) was implemented in BSD Unix, starting in version 4.3 and SunOS in version 3.5. It was not until 1999, in Linux Kernel version 2.1.75, that the Linux Socket Filter, aka BPF, was released.

BPF architecture

This section covers the implementation of BPF as described in *McCanne* and *Van Jacobson's* USENIX paper. As we will go on to discuss, various platforms have implemented BPF with some variety.

BPF works by a user defining a filter (that is program) which is converted to BPF bytecode and then passed to the BPF virtual machine in kernel space for execution by an interpreter as seen in *Figure 1.1.* This allows the filter to be run in kernel-space safely which removes the need for copying all packets across from kernel-space to user-space.

***Figure 1.1**: The BPF filter lifecycle*

BPF has three main components (displayed in *Figure 1.2*):

- **Network tap**: Collects copies of packets from the network device drivers and delivers them to listening applications
- **Packet filter**: Decides if a packet should be accepted and how to copy it to the application
- **The BPF pseudo-machine**: The in-kernel VM that runs the BPF filter (i.e. program)

Figure 1.2 illustrates the original BPF architecture and the division between user-space, kernel-space, and the network:

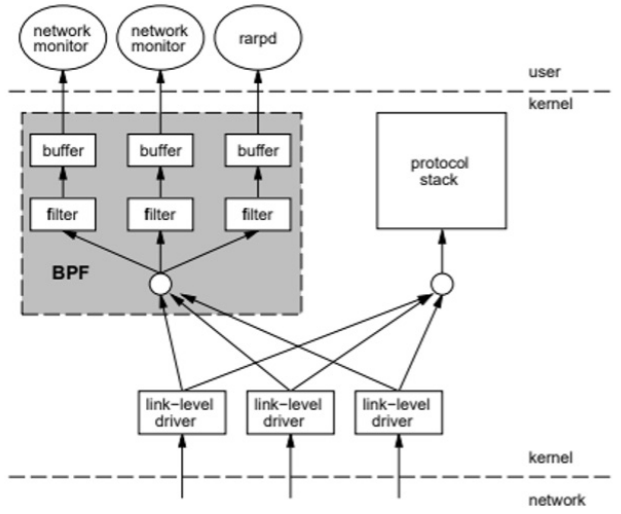

***Figure 1.2**: The BPF architecture (source: BSD Packet Filter)*

Network tap

While a BPF filter (program) is running and the packet arrives at the network interface, instead of being processed via the unix protocol stack (e.g. TCP or UDP), it is first processed by the BPF pseudo-machine. BPF feeds the packet to the user-defined filter(s), aka the BPF filter, where if the filter accepts the packet, it is copied over to the buffer associated with that filter. This is an optimization over all packets having to be copied from kernel-space to user-space and then filtered.

The program listening in user-space reads the packets in the buffer and processes/displays them in the manner that the user-space application defines. The packet then proceeds to be processed by the network stack as normal.

Filter

Until the creation of BPF, there were several filter models in existence: **CMU/ Stanford Packet Filter** (**CSPF**), NIT in SunOS, Ultrix Packet Filter in DEC Ultrix and snoop in SGI IRIX. CSPF was one of the more prevalent filter models.

BPF utilizes a **control flow graph** (**CFG**) filter displayed in *Figure 1.3*, which removes redundant evaluations that are present in the CSPF

filter. Each node in the graph is a comparison predicate with two final targets (true/false). This is easier to model on registers. There is a one-time overhead to order/optimize the graph.

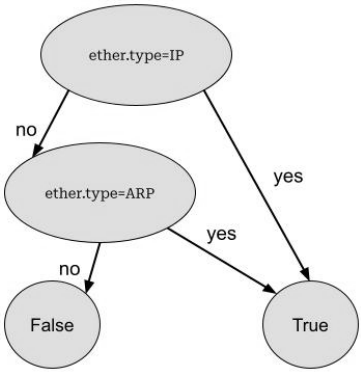

***Figure 1.3**: Simple CFG of a ip or arp filter (source: BSD Packet Filter)*

While the above example is simple, you can see in this more complex filter example, shown in *Figure 1.4*.

You can see in this example, there is a maximum of 5 evaluation operations.

If this same filter were to be written using CSPF, there would be a minimum (and maximum) of 7 evaluation operations (as shown in *Figure 1.4*):

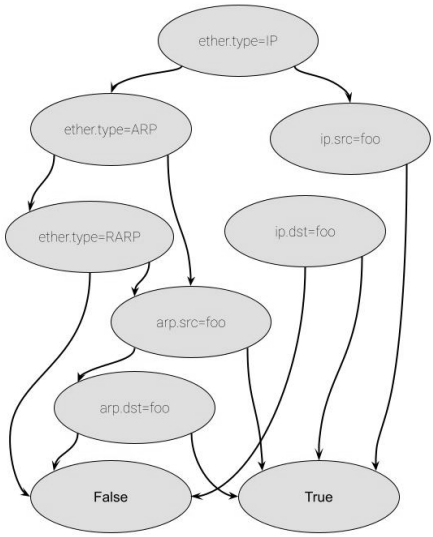

***Figure 1.4**: CFG of a host foo filter (source: BSD Packet Filter)*