BUILD.
OBSERVE.
SECURE.
PERFORM.

eBPF for SYSTEMS ENGINEERS

FROM FIRST PRINCIPLES TO PRODUCTION DEPLOYMENTS

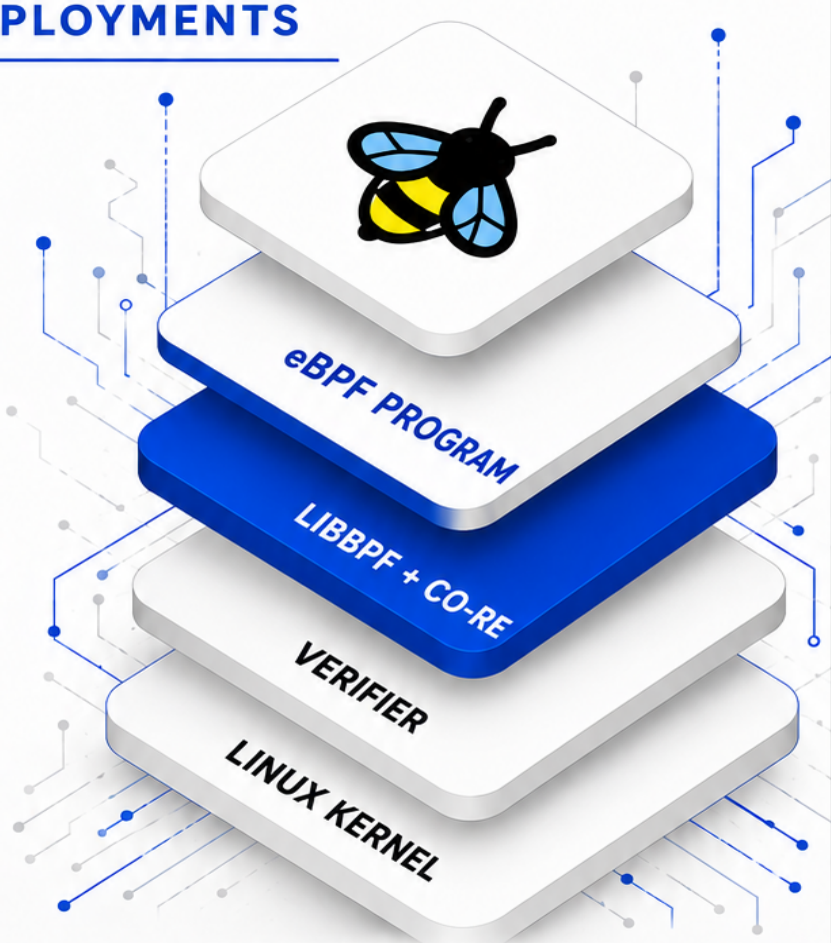
Learn to build, debug, and deploy eBPF programs that power observability, networking, security, and performance—safely in the Linux kernel.

OBSERVE
Gain deep insights with eBPF

CONNECT
Build high-performance networking solutions

SECURE
Enforce policy and detect threats

PERFORM
Optimize systems without compromise



BENJAMIN CARTER

eBPF for Systems Engineers

From First Principles to Production Deployments

Steve Publications

This book is available at https://leanpub.com/ebpfforsystemsengineers

This version was published on 2026-08-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

From First Principles to Production Deployments

This book teaches you everything you need to know about eBPF, from the foundational concepts to advanced production deployments. You will learn how to write, compile, load, debug, and operate eBPF programs that run safely in the Linux kernel to provide observability, networking, security, and performance capabilities without modifying kernel source code or loading traditional kernel modules. By the end of this book, you will be able to design and implement production-grade eBPF solutions using modern libbpf and CO-RE techniques that work across different kernel versions and environments.

Introduction: The eBPF Revolution

The Observability Problem in Modern Infrastructure

Imagine this scenario. You are on call at 2 AM. A production service is experiencing intermittent latency spikes that affect customers. The application logs show nothing unusual. CPU usage looks normal. Memory is fine. Network throughput appears stable. Yet something is clearly wrong, and you need to find it before more customers are impacted.

Five years ago, your options were limited. You could add more logging to the application and redeploy, hoping the issue reproduces. You could run perf or strace on a subset of processes, adding overhead that might change the behavior you are trying to diagnose. You could examine kernel logs and hope something relevant appeared there. Each approach was slow, invasive, and often inconclusive.

Now imagine having the ability to see exactly what is happening inside every process and every system call that touches your service, without modifying the application code, without restarting anything, and with minimal performance impact. You can trace kernel function calls to understand where time is being spent. You can monitor network packet flows at line rate. You can track file access patterns across thousands of processes simultaneously. You can build custom security policies that run directly in the kernel and enforce them dynamically.

This is what eBPF makes possible.

Extended Berkeley Packet Filter, or eBPF, has become the most significant advancement in Linux systems engineering in over a decade. It provides a safe, efficient mechanism to run sandboxed programs inside the kernel in response to events, enabling unprecedented visibility and control over system behavior without modifying kernel source code or loading traditional kernel modules.

eBPF is not just another monitoring tool. It is a fundamental shift in how we interact with the operating system kernel. Where kernel modules required complete trust and could crash or compromise the system if buggy, eBPF

programs are verified for safety before they ever execute. Where adding instrumentation meant recompiling and redeploying applications, eBPF lets you trace any function in the kernel or user space dynamically. Where networking decisions were fixed at compile time or managed through static iptables rules, eBPF enables programmable packet processing that runs at line rate inside the network stack.

Major technology companies have already made eBPF central to their infrastructure. Meta uses eBPF for load balancing across its entire fleet, handling billions of requests per second. Cloudflare relies on eBPF for DDoS protection and traffic management. Google uses it for security monitoring and policy enforcement. Kubernetes projects like Cilium have replaced traditional iptables-based networking with eBPF, achieving better performance and richer capabilities in the process.

The technology has matured rapidly since its introduction into the Linux kernel in 2014. What started as an extension of packet filtering has grown into a comprehensive platform for observability, networking, security, and even custom CPU scheduling. The eBPF instruction set has been standardized as RFC 9669, and the technology has expanded beyond Linux to Windows and other platforms.

If you work with Linux systems at any level, understanding eBPF is no longer optional. It is the tool that will define how we observe, secure, and optimize infrastructure for the next generation of distributed systems.

Before eBPF: What We Used to Do

To appreciate what eBPF enables, it helps to understand what engineers did before it existed. The problems eBPF solves are not new. Systems engineers have always needed visibility into running systems and the ability to modify behavior without downtime. But the tools available were constrained by fundamental limitations in how kernels were designed.

Kernel modules provided one path for extending kernel functionality, but they came with serious drawbacks. A kernel module runs with full privileges and direct access to all kernel memory and data structures. A bug in a module can crash the entire system, corrupt data, or create security vulnerabilities. Loading a module typically requires root access and often a system reboot. The development cycle is slow and risky.

For tracing and observability, engineers relied on a patchwork of tools. strace provided visibility into system calls but added significant overhead and could only trace a single process at a time. perf offered CPU and hardware counter profiling but required careful configuration and produced output that was difficult to correlate with application behavior. ftrace and kprobes existed in the kernel but had steep learning curves and limited programmability. Network monitoring depended on tcpdump and iptables, which were powerful but static and could not easily adapt to dynamic conditions.

Adding instrumentation to applications meant modifying source code, recompiling, and redeploying. In large organizations with thousands of services, this was impractical for many use cases. Application developers had to anticipate every possible monitoring need upfront, leading to either excessive logging that created noise and storage costs or insufficient visibility that left teams blind when problems occurred.

The limitations became especially acute as systems grew more complex. Microservices architectures multiplied the number of processes and network connections to monitor. Container orchestration introduced dynamic environments where workloads appeared and disappeared constantly. Cloud-native applications required observability at scale that traditional tools could not provide efficiently.

What was needed was a way to run custom code in the kernel that was safe, flexible, and efficient. The code had to be dynamically loadable without rebooting. It had to be able to access kernel data structures and respond to events with minimal overhead. And it had to be programmable enough to handle complex logic while being constrained enough that a bug could not bring down the system.

That is exactly what eBPF delivers.

What eBPF Gives You

At its core, eBPF provides three capabilities that transform systems engineering:

First, safe kernel extension. eBPF programs run inside the kernel but are sandboxed by a sophisticated verifier that performs static analysis before any code executes. The verifier guarantees that programs will not crash the kernel, access memory they should not, or run indefinitely. This means you can

load eBPF programs from untrusted sources with confidence that they cannot compromise system stability. Programs are dynamically loaded and unloaded without affecting the running kernel.

Second, deep visibility into system behavior. eBPF programs attach to hooks throughout the kernel and user space, executing when specific events occur. You can trace any kernel function, monitor system calls across all processes simultaneously, inspect network packets at multiple points in the stack, track file operations, profile CPU usage with microsecond precision, and much more. The data collected is rich and contextual, enabling you to understand not just what happened but why it happened.

Third, programmable infrastructure. eBPF moves decisions that were previously static or managed by complex external systems into the kernel itself. Network policies can be enforced at line rate without copying packets to user space. Security checks can run synchronously as part of system call processing. Load balancing can happen in the kernel with full awareness of connection state and application semantics. This reduces latency, simplifies architecture, and enables capabilities that were previously impractical.

eBPF achieves these goals through several key technical innovations. A specialized virtual machine executes programs using a simple, well-defined instruction set that maps efficiently to native hardware. A just-in-time compiler translates bytecode to optimized machine code at load time, delivering performance close to hand-written kernel code. Maps provide shared data structures for communication between kernel and user space with minimal overhead. Helper functions give controlled access to kernel services without exposing the full complexity of kernel internals. And the CO-RE mechanism ensures that programs compiled once can run across different kernel versions by automatically adjusting to structural differences at load time.

These capabilities combine to make eBPF uniquely powerful for modern infrastructure challenges:

- For observability, eBPF enables comprehensive monitoring of applications, containers, and infrastructure with minimal overhead and no code changes to instrumented services.
- For networking, eBPF provides programmable packet processing that can implement custom routing, load balancing, firewalling, and traffic analysis at wire speed.

- For security, eBPF allows dynamic policy enforcement and real-time threat detection based on actual system behavior rather than static rules or signatures.
- For performance engineering, eBPF offers fine-grained profiling and tracing that pinpoints bottlenecks and enables data-driven optimization decisions.

Who Should Read This Book

This book is written for systems engineers who want to understand and use eBPF in production environments. You should be comfortable with Linux fundamentals, including the command line, basic C programming concepts, and understanding of processes, networking, and filesystems. You do not need prior experience with eBPF or kernel development.

The book will be particularly valuable if you are:

- A DevOps engineer or SRE responsible for monitoring and troubleshooting complex distributed systems
- A platform or cloud infrastructure engineer building and maintaining the foundations that applications run on
- A network engineer looking to move beyond static configurations toward programmable, adaptive networking
- A security engineer seeking deeper visibility into system behavior and more flexible policy enforcement mechanisms
- A backend developer curious about what happens inside the kernel when your code runs
- Anyone who has ever been frustrated by the limitations of traditional monitoring and debugging tools

If you work with Kubernetes, container orchestration, or cloud-native infrastructure, eBPF is especially relevant. Many of the leading projects in these spaces, including Cilium, Falco, Tetragon, and others, are built on eBPF. Understanding the technology will help you use these tools more effectively and build custom solutions when off-the-shelf options fall short.

How This Book Is Organized

The book is structured to take you from first principles through advanced production deployments in a logical progression. Each chapter builds on the previous ones, so reading in order is recommended, though you may want to jump ahead to specific topics once the foundations are clear.

The Introduction sets up why eBPF matters and what problems it solves. Chapter 1 covers the history and evolution of BPF and eBPF, providing context for design decisions and constraints you will encounter. Chapter 2 establishes the Linux kernel architecture knowledge needed to understand where and how eBPF programs run.

Chapters 3 through 6 form the technical core of how eBPF works internally. You will learn about the execution model and instruction set, write your first programs using the modern toolchain, understand the verifier that ensures safety, and see how JIT compilation delivers performance. These chapters provide the foundation for everything that follows.

Chapters 7 and 8 cover maps and helper functions, the mechanisms by which eBPF programs store data and interact with kernel services. These are essential building blocks used in almost every eBPF program.

Chapter 9 explains BTF and CO-RE, the mechanisms that make eBPF programs portable across different kernel versions. This is critical knowledge for production deployments where you cannot control the exact kernel version running on each system.

Chapters 10 and 11 cover tracing mechanisms in depth: kprobes, uprobes, tracepoints, and the newer fentry/fexit approaches. These are the primary hooks used to observe and instrument system behavior.

Chapters 12 through 14 focus on networking and security. You will learn about XDP for high-performance packet processing, traffic control and socket filtering, cgroup-based policies, and Linux Security Module integration for deep kernel-level security enforcement.

Chapters 15 and 16 address production use cases for observability and Kubernetes. You will see how to build custom monitoring tools, integrate with Prometheus and OpenTelemetry, trace containerized workloads, and leverage eBPF in cloud-native environments.

Chapter 17 covers the practical aspects of deploying, operating, and securing eBPF programs in production. This includes debugging techniques, performance tuning, version compatibility management, and security hardening.

The Conclusion synthesizes the material and looks at where eBPF is headed. Throughout the book, code examples use modern libbpf and CO-RE techniques that are production-ready and work across a wide range of kernel versions. Every example is complete and self-contained, with build instructions and explanations for every significant line of code.

By the end of this book, you will not just understand what eBPF can do. You will know how to use it effectively to solve real problems in your infrastructure, and you will have the knowledge to evaluate new eBPF capabilities as they emerge and determine when and how to apply them.

Chapter 1: History and Evolution of BPF

Understanding where eBPF came from is not academic trivia. The design decisions made decades ago still shape how the technology works today. Constraints inherited from early packet filtering implementations explain why eBPF has a fixed-size stack, why programs must terminate deterministically, and why helper functions are the only way eBPF code talks to the kernel. This chapter traces the evolution from the original Berkeley Packet Filter to modern eBPF, showing how each step was driven by real engineering needs and technical trade-offs.

The Original Berkeley Packet Filter

The story begins in 1992 at Lawrence Berkeley National Laboratory. Steven McCanne and Van Jacobson were working on network measurement tools and ran into a fundamental problem. When capturing network packets for analysis, the operating system copied every packet from kernel space to user space, even if the application only wanted a small subset of traffic. For high-speed networks, this meant most of the work was wasted effort copying data that would immediately be discarded.

Their solution was elegant in its simplicity. Instead of filtering packets in user space after copying them, why not run the filter program inside the kernel and only copy packets that matched? This is exactly what they implemented, and they called it the Berkeley Packet Filter, or BPF.

The original BPF was a domain-specific virtual machine designed for one purpose: efficient packet filtering. It had a very simple architecture. There were two 32-bit registers, an accumulator A and an index register X. Instructions operated on these registers and a small scratch memory area. The instruction set supported basic arithmetic, loads from packet data, conditional jumps, and return values indicating whether to accept or drop a packet.

This design was constrained by the technology of the time. Processors had limited registers, and the virtual machine needed to be simple enough to implement efficiently in kernel code. The stack-based accumulator model was familiar from early computing architectures and mapped reasonably well to available hardware.

BPF was incorporated into BSD Unix and quickly adopted by other operating systems. It became the foundation for tcpdump, the ubiquitous packet capture tool, and libpcap, the library that provides raw socket access on Unix-like systems. For decades, BPF did exactly what it was designed to do: filter network packets efficiently in the kernel.

But as systems evolved, the limitations of this simple design became apparent. The two-register architecture made complex logic difficult to express. Conditional jumps could not support efficient loops, limiting the kinds of filtering rules that could be implemented. There were no data structures for maintaining state across packet processing. And most importantly, BPF was designed exclusively for networking. Its architecture made no sense for other use cases like system tracing or security policy enforcement.

Linux Socket Filters: The First Step

Linux adopted BPF in kernel version 2.1.75 in 1997, implementing it as socket filters. These allowed applications to attach filter programs to sockets, controlling which packets were delivered to user space. This was useful for network monitoring tools and applications that wanted to reduce the overhead of receiving unwanted packets.

Linux socket filters used the classic BPF instruction set and preserved the original design constraints. They worked well for their intended purpose but shared all the same limitations as the BSD implementation. The architecture was fundamentally designed around packet filtering, and extending it to other domains would have required significant changes that conflicted with the existing semantics.

During this period, the Linux kernel also developed its own tracing infrastructure. kprobes, introduced in kernel 2.6, allowed dynamic instrumentation of almost any kernel instruction by replacing it with a breakpoint and running a handler when hit. This was powerful but had limitations. Handlers ran in

whatever context the probed code was executing in, which could be problematic for complex operations. There was no sandboxing mechanism, so buggy handlers could crash the system. And the interface was not programmable in a way that made it easy to build sophisticated tracing applications.

By the late 2000s and early 2010s, several factors converged to create demand for something better than what either classic BPF or kprobes could provide. Virtualization and cloud computing were creating massive, dynamic infrastructures that needed deep visibility. Network functions like load balancing and firewalling were becoming bottlenecks when implemented in user space. Security teams wanted more flexible mechanisms for monitoring and enforcing policies without deploying kernel modules on every system.

The Birth of cBPF and eBPF

The term cBPF, or classic BPF, came into use to distinguish the original architecture from what was coming next. The extended Berkeley Packet Filter, eBPF, was conceived at PLUMgrid, a networking startup founded by engineers who had worked on virtualization and cloud infrastructure. Alexei Starovoitov led the development effort, with significant contributions from Daniel Borkmann and others.

The goal was ambitious: create a general-purpose execution environment inside the kernel that could safely run untrusted code, support rich functionality beyond packet filtering, and deliver performance close to native kernel code. The approach was to take the concept of an in-kernel virtual machine from classic BPF and redesign it from the ground up for modern hardware and broader use cases.

The first major change was the instruction set. eBPF moved from a stack-based architecture with two registers to a register-based design with ten 64-bit general-purpose registers plus a frame pointer. This made the virtual machine much more similar to real processors like x86 and ARM, enabling better code generation from high-level languages and more efficient JIT compilation. The expanded register file meant that function calls could pass arguments in registers rather than through memory, just like native code.

The second major change was the addition of a sophisticated verifier. Classic BPF programs were trusted; there was no mechanism to check whether they were safe before execution. eBPF introduced static analysis that examines

every possible execution path of a program and guarantees properties like termination, memory safety, and bounded resource usage before the program is allowed to run. This was critical for enabling eBPF to be used with untrusted code and for justifying JIT compilation of dynamically loaded programs.

The third major change was the introduction of maps. Classic BPF had no way to maintain state between invocations. eBPF maps provide key-value storage that persists across program executions and can be shared between multiple programs and user space. This enabled use cases like connection tracking, rate limiting, and aggregation of statistics that were impossible with the original architecture.

The fourth change was the helper function mechanism. Rather than giving eBPF programs direct access to arbitrary kernel code, eBPF provides a white-listed set of helper functions for specific operations like reading process information, accessing network data, or getting timestamps. This maintains the safety boundary while enabling rich functionality.

Alexei Starovoitov and the eBPF Project

Alexei Starovoitov's background shaped the direction of eBPF in fundamental ways. He had worked on virtualization and networking at PLUMgrid, where he encountered practical problems that existing tools could not solve efficiently. The company needed to implement sophisticated network policies and load balancing in virtualized environments, and the options available were unsatisfactory.

Kernel modules provided the performance needed but came with deployment and safety concerns. User-space proxies added latency and complexity. Classic BPF was too limited for the logic required. Starovoitov recognized that what was needed was a programmable execution environment inside the kernel that combined the flexibility of user-space code with the efficiency and visibility of kernel code, while maintaining safety guarantees that made deployment practical at scale.

The decision to frame eBPF as an extension of BPF rather than an entirely new technology was strategic. As Daniel Borkmann advised, positioning it as an evolution of existing packet filtering infrastructure made it easier to get accepted into the kernel networking subsystem. The name preserved continuity even though the underlying architecture was fundamentally different.

The development process was iterative and collaborative. Early versions were submitted to the Linux kernel mailing list, where they went through rigorous review by kernel maintainers and contributors. Features were added incrementally based on demonstrated use cases and careful consideration of security implications. This gradual approach, while slower than shipping a complete system at once, resulted in technology that integrated well with the rest of the kernel and earned the trust of the community.

Major Milestones: From 3.18 to 6.x

eBPF was merged into the Linux kernel in version 3.18, released in March 2014. The initial implementation included the basic infrastructure: the bpf syscall for loading programs and creating maps, the verifier, support for socket filters using the new instruction set, and a small set of helper functions. It was a foundation rather than a complete system, but it proved the concept worked.

The pace of development accelerated rapidly over the following years. Here are the key milestones that shaped modern eBPF:

- Linux 4.1 (January 2015): kprobe support enabled eBPF programs to attach to kernel functions for tracing, opening up observability use cases far beyond networking. The bpf_trace_printk helper was added for debugging output. Traffic control integration allowed eBPF programs to classify and act on packets in the TC subsystem.
- Linux 4.3 (July 2015): cgroup support enabled attaching eBPF programs to control groups, making it possible to implement policies based on process membership rather than just network interfaces. This was critical for container-aware networking and security.
- Linux 4.8 (July 2016): XDP, the eXpress Data Path, introduced high-performance packet processing at the driver level before packets entered the kernel networking stack. This enabled line-rate filtering, load balancing, and DDoS protection that was previously only possible with specialized hardware or user-space frameworks like DPDK.
- Linux 4.9 (October 2016): uprobes extended tracing to user space, allowing eBPF programs to instrument application functions without modifying or recompiling the code. This made it practical to trace any running process for observability and debugging.

- Linux 4.17 (January 2018): Sleepable eBPF programs were added, allowing programs to block and wait for operations like I/O instead of having to complete immediately. This expanded the range of operations eBPF could perform but required careful handling to avoid deadlocks and priority inversions.
- Linux 5.4 (February 2019): BTF, the BPF Type Format, was standardized as a way to embed type information in kernel and eBPF binaries. This enabled better debugging, introspection, and the CO-RE mechanism that would follow.
- Linux 5.5 (March 2019): fentry and fexit probes provided a more efficient alternative to kprobes for tracing kernel functions, using the existing ftrace infrastructure to achieve lower overhead.
- Linux 5.7 (July 2019): LSM hooks allowed eBPF programs to attach to the same security decision points used by SELinux and AppArmor, enabling dynamic security policy enforcement without writing kernel modules.
- Linux 5.8 (October 2019): The ring buffer map type was introduced as a more efficient alternative to perf buffers for streaming data from kernel to user space, with better performance and guaranteed event ordering.
- Linux 5.15 (January 2021): The SYSCALL program type and improved syscall tracing capabilities enhanced security monitoring and auditing.
- Linux 6.x series: Continued expansion of capabilities including kfuncs (direct calls to kernel functions from eBPF), sched_ext for custom CPU scheduling, arena maps for shared memory, and numerous improvements to the verifier, JIT compilers, and helper function set.

Each milestone added capabilities that enabled new use cases and made existing ones more practical. The technology evolved from a specialized networking tool into a comprehensive platform for kernel extension. By the mid-2020s, eBPF had become central to how many organizations approach observability, networking, and security in their infrastructure.

The standardization of the eBPF instruction set as RFC 9669 in October 2024 marked another important milestone. This formal specification enables implementations beyond Linux and provides a stable reference for tooling and cross-platform development. The eBPF Foundation, established under the Linux Foundation, now governs the technology's continued evolution with participation from major industry players.

Understanding this history matters because it explains why eBPF works the way it does. The verifier exists because loading untrusted code into the kernel

requires safety guarantees. Maps exist because observability and networking use cases need shared state. Helper functions exist to provide controlled access to kernel capabilities. CO-RE exists because production environments run many different kernel versions. Each feature was added to solve a real problem encountered by people building real systems, and that pragmatic approach continues to drive development today.

Chapter 2: Linux Kernel Architecture for eBPF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Kernel Space vs User Space: The Boundary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

System Calls and Context Switching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Kernel Hooks and Tracing Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Memory Management Basics for eBPF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The BPF Virtual Machine Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

How Programs Attach to the Kernel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 3: The eBPF Execution Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The eBPF Instruction Set Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Registers and Calling Conventions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The Fixed-Size Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Control Flow: Loops and Conditionals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Execution Contexts and Program Inputs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Deterministic Termination Requirements

Chapter 4: Writing Your First eBPF Programs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Setting Up the Development Environment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The LLVM/Clang Compilation Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

A Hello World Tracepoint Program

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Building with libbpf Skeletons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Running and Reading Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Understanding the Generated Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 5: The eBPF Verifier

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Why the Verifier Exists

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Safety Guarantees: Memory, Execution Time, Privileges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

How the Verifier Works: State Machine Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Common Verification Failures and Fixes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Loop Unrolling and Complexity Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Understanding Verifier Logs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 6: JIT Compilation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Why JIT Compilation Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The JIT Compilation Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Architecture-Specific JITs (x86, ARM, RISC-V)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

JIT vs Interpreter Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Enabling, Disabling, and Tuning JIT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

When Native Performance Is Critical

Chapter 7: eBPF Maps — Data Structures Between Kernel and User Space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

What Maps Are and Why They Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Hash Maps: The Workhorse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Array Maps and Perf Arrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Ring Buffers for High-Throughput Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

LRU and Per-CPU Variants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Map Performance and Capacity Planning

Chapter 8: Helper Functions — Talking to the Kernel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The Helper Function Calling Convention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Core Helpers: Time, Task, and Map Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Networking Helpers for Packets and Sockets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Tracing Helpers for kprobes and Tracepoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Security and Permission Helpers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Choosing the Right Helper

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 9: BTF and CO-RE — Kernel-Agnostic eBPF Programs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The Kernel Version Compatibility Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

What BTF Is: Type Information in the Kernel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

How CO-RE Relocation Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Writing CO-RE-Compatible Programs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

libbpf's Role in CO-RE

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Debugging CO-RE Relocations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 10: Tracing with kprobes, uprobes, and Tracepoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

kprobes: Instrumenting Kernel Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Finding Safe Probe Locations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

uprobes for User-Space Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Tracepoints: Stable Instrumentation Points

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Raw Tracepoints for Maximum Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Choosing the Right Tracing Mechanism

Chapter 11: fentry, fexit, and Modern Function Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

How fentry/fexit Differ from kprobes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Performance Advantages of fentry/fexit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Attaching to Kernel Functions Safely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Return Value Tracing with fexit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Limitations and Gotchas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Migration Guide: From kprobes to fentry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 12: XDP — Extreme Packet Processing at Line Rate

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The Networking Performance Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

XDP Architecture and Execution Points

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Driver, Hardware, and Generic Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Writing Your First XDP Program

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Packet Manipulation and Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

XDP in Production: Load Balancing and DDoS Protection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 13: Traffic Control, Socket Filters, and cgroups Programs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Traffic Control (TC) Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Socket Filters for Application Traffic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

cgroup Attach Points: v1 vs v2

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Socket Operations Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Container-Aware Networking with cgroups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Combining Multiple Network Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 14: Security Monitoring with eBPF and LSM Hooks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Why Traditional Security Tools Fall Short

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Syscall Monitoring and Auditing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

File System Access Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Process Lifecycle Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

LSM Hooks: Deep Kernel Security Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Building a Security Monitoring Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 15: Observability, Profiling, and Performance Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

The Three Pillars: Metrics, Traces, Logs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Building Custom Observability Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

CPU Profiling with eBPF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Latency Analysis and Histograms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Integration with Prometheus and OpenTelemetry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Production Monitoring Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 16: Kubernetes and Container Visibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Containers as a Tracing Challenge

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Namespace-Aware eBPF Programs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Pod and Container Identification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Service Mesh Integration (Istio, Cilium)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Runtime Security in Kubernetes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Multi-Tenant Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Chapter 17: Production Operations — Deploying, Securing, and Maintaining eBPF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Security Model and Privilege Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Deployment Strategies and Automation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Debugging Production eBPF Programs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Performance Tuning and Resource Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Version Compatibility and Rollout Planning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Monitoring Your eBPF Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Conclusion: The Future of eBPF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Where eBPF Has Been

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Emerging Capabilities and Active Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Beyond Linux: Cross-Platform Prospects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Skills That Will Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

Final Thoughts on Systems Engineering with eBPF

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ebpfforsystemsengineers.