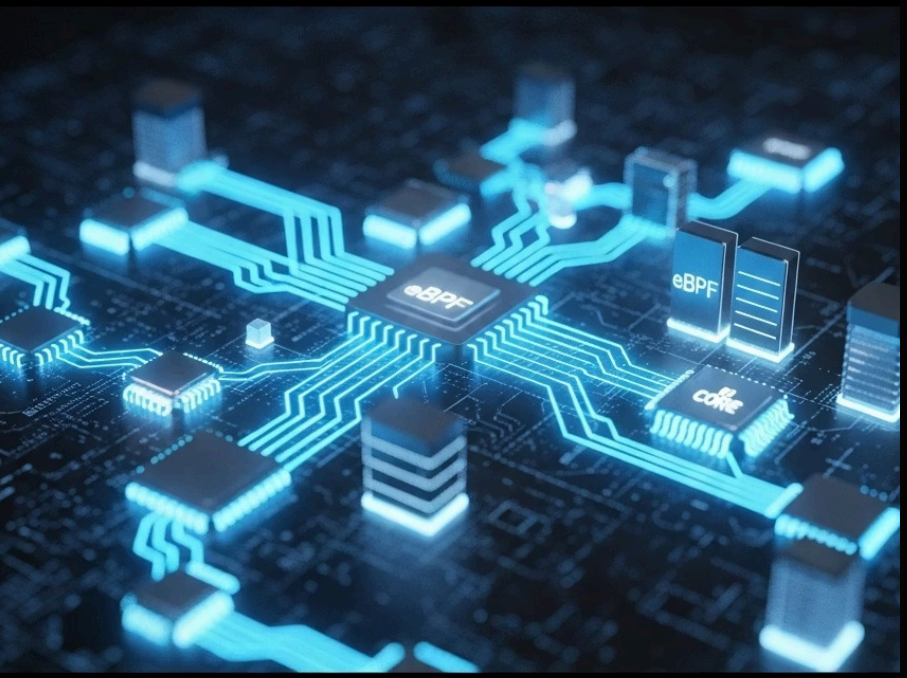


Linux Observability and Tuning using bpftrace

An intensive hands-on training for advanced Linux performance analysis, debugging, and security monitoring using the powerful eBPF tracing language

Instructor: Chandrashekar Babu <training@chandrashekar.info>

https://www.chandrashekar.info/ | **https://www.slashprog.com/**

Course Overview

Duration

5 days @ 4 hours per day

Level

Advanced

Approach

Hands-on with extensive try-out exercises

This intensive training teaches you how to leverage bpftrace for real-time system observability, performance analysis, and tuning. You'll learn to write custom scripts to monitor kernel and userspace events, diagnose latency issues, and optimise system behaviour without modifying code.

By the end of this course, you'll be able to develop sophisticated tracing solutions to troubleshoot complex performance problems in production environments.

Target Audience

SREs & DevOps Engineers

Who need to troubleshoot and optimise production systems running critical workloads

Performance Engineers

Focused on tuning Linux workloads for maximum efficiency and minimal latency

Kernel Developers

Who require lightweight tracing tools for debugging kernel issues

Security Analysts

Monitoring runtime behaviour for anomalies and potential security breaches

About me

A FOSS Technologist and Corporate Trainer with 30+ years of experience in software development, technology consulting and corporate training.

A Linux enthusiast since late 1995

Expertise in diverse technology domains in the Linux ecosystem (Linux Kernel / Device Driver development, Linux Administration and Deployment, MySQL / PostgreSQL Database systems, Apache web server administration and tuning, Web development using PHP, Rails, Flask, software architecture and design principles, agile / adaptive software methodologies).

To know more about me: kindly visit **https://www.chandrashekar.info/** | **https://www.slashprog.com/**

LinkedIn: **https://www.linkedin.com/in/chandrashekarbabu/**

YouTube: **https://www.youtube.com/@chandrashekarbabu** Udemy: **https://www.udemy.com/user/chandrashekar-babu-3/**

Your brief introductions please...

Your designation / role in your organization

You experience

Your comfort-level on Linux

Your comfort-level in Linux debugging and Instrumentation

Your specific expectation (related to this training) if any...

Prerequisites

Required Skills

- Strong Linux CLI proficiency
- Understanding of Linux system architecture (syscalls, processes, filesystems)
- Familiarity with basic performance metrics (CPU, memory, disk I/O, network)
- Experience with bash or Python scripting
- Comfort with reading C code (for kernel structures)

While we'll review key concepts, participants are expected to have working knowledge of Linux internals. This isn't an introduction to Linux — it's an advanced course for practitioners who want to take their diagnostic abilities to the next level.

Lab Environment Requirements

Linux Host or VM

- Kernel ≥ 4.9 (5.x+ recommended for full feature support)
- Root access (required for kernel probes)
- 4GB+ RAM recommended

Required Software

- bpftrace installed (via package manager)
- Linux perf tools
- BCC tools (optional for comparisons)
- Text editor of choice

Installation Commands

```
# Ubuntu/Debian
sudo apt install -y bpftrace linux-tools-common

# RHEL/CentOS/Fedora
sudo yum install -y bpftrace perf
```

Course Agenda

Module 1: eBPF & bpftrace Fundamentals

Core concepts, architecture, and basic usage patterns

Module 2: System-Wide Observability

Tracing syscalls, filesystem operations, and process events

Module 3: Performance Tuning Deep Dive

CPU profiling, memory allocation analysis, and I/O bottlenecks

Module 4: Networking & Security Observability

Monitoring network stacks, socket operations, and security events

Module 5: Advanced Scripting & Production Use

Real-world techniques and production troubleshooting

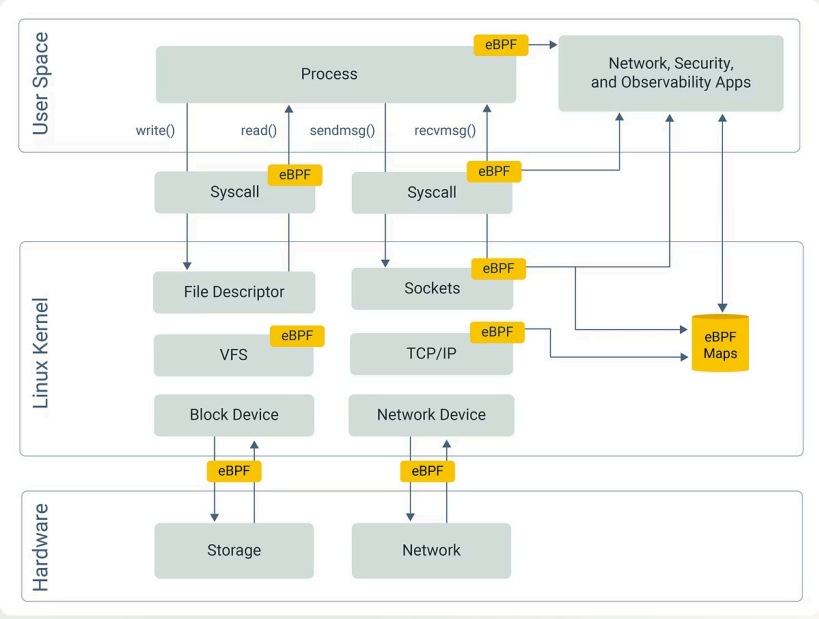
Each module includes lecture content followed by extensive hands-on exercises to solidify your understanding through practical application.

Module 1

eBPF & bpftrace Fundamentals

eBPF: Revolutionizing the Linux Kernel

A comprehensive exploration of extended Berkeley Packet Filter (eBPF) technology, its revolutionary impact on Linux kernel capabilities, and how it enables unprecedented innovation in observability, security, and networking.



What is eBPF?

eBPF (extended Berkeley Packet Filter) is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in a privileged context such as the operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring changes to kernel source code or loading kernel modules.

Safe Execution

eBPF allows sandboxed programs to run within the operating system, with safety guaranteed through verification and Just-In-Time compilation.

Runtime Extension

Application developers can add capabilities to the operating system at runtime without modifying kernel code or loading modules.

Diverse Applications

Powers networking, observability, and security functionality in modern data centers and cloud native environments.

The Evolution of eBPF

BPF originally stood for Berkeley Packet Filter, a framework that originated in the BSD UNIX ecosystem dating back to 1992. BPF was initially introduced as a mechanism for efficient network packet filtering and analysis. It allowed users to define filtering rules in a specialized bytecode, which the kernel could execute to selectively capture network packets, minimizing the need to copy unnecessary data to user space.

In 2014, the Linux kernel introduced eBPF, a significant extension of the original BPF.

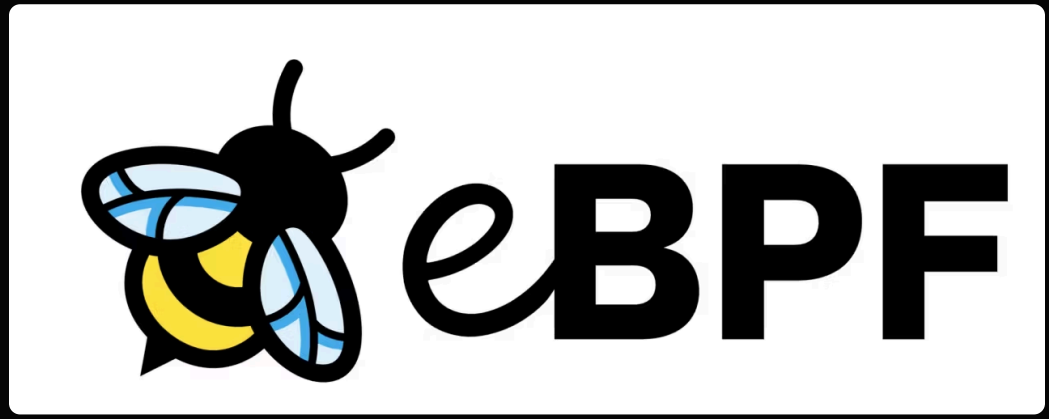
Linux but now that eBPF (extended BPF) can do so much more than packet filtering, the acronym no longer makes sense. eBPF is now considered a standalone term that doesn't stand for anything.

In the Linux source code, the term BPF persists, and in tooling and documentation, the terms BPF and eBPF are generally used interchangeably. The original BPF is sometimes referred to as cBPF (classic BPF) to distinguish it from eBPF.

Meet eBee: The eBPF Mascot

The bee is the official logo for eBPF and was originally created by Vadim Shchekoldin. At the first eBPF Summit in 2020, there was a vote taken and the bee was named eBee.

The bee symbolizes the industrious nature of eBPF - working efficiently behind the scenes to enhance the kernel's capabilities, much like how bees work tirelessly to support their ecosystem.



The official eBPF mascot: eBee

Why the Operating System Kernel Matters

Historically, the operating system has always been an ideal place to implement observability, security, and networking functionality due to the kernel's privileged ability to oversee and control the entire system.

Central Control Point

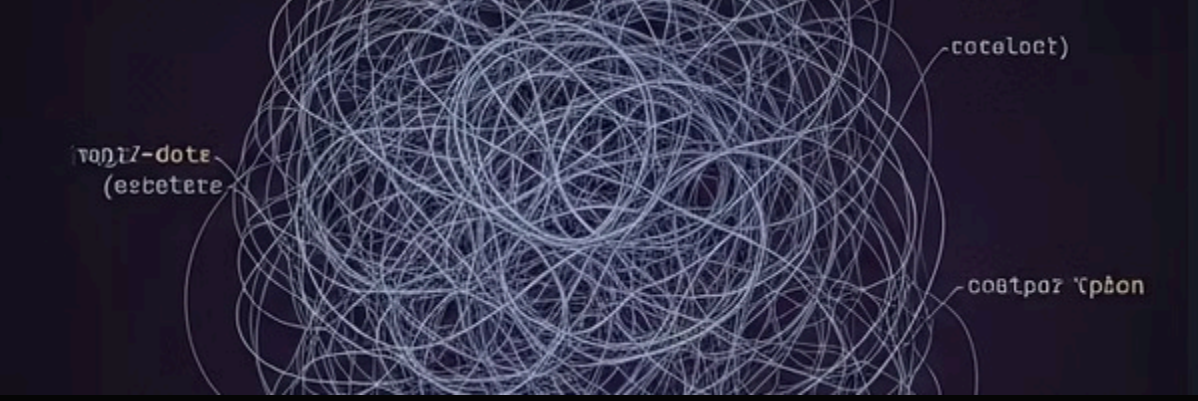
The kernel has visibility into all system activities, making it ideal for monitoring and security functions.

Stability Requirements

Kernels are hard to evolve due to their central role and high requirements for stability and security.

Innovation Challenges

The rate of innovation at the operating system level has traditionally been lower compared to functionality implemented outside of the OS.



How eBPF Changes the Game

eBPF changes the fundamental formula of kernel innovation. It allows sandboxed programs to run within the operating system, which means that application developers can run eBPF programs to add additional capabilities to the operating system at runtime.

Before eBPF

- Change kernel source code and convince the Linux kernel community
- Wait several years for the new kernel version to become a commodity
- Write a kernel module that needs regular fixes and risks corrupting the kernel

With eBPF

- Write sandboxed programs that run in privileged context
- Add capabilities without changing kernel source code
- Ensure safety through verification and JIT compilation
- Deploy innovations immediately without kernel version changes

eBPF Use Cases Today

High-Performance Networking

Providing high-performance networking and load-balancing in modern data centers and cloud native environments.

Security Observability

Extracting fine-grained security observability data at low overhead, enabling better threat detection and response.

Application Tracing

Helping application developers trace applications and providing insights for performance troubleshooting.

Container Security

Preventive application and container runtime security enforcement, protecting workloads in real-time.

The possibilities are endless, and the innovation that eBPF is unlocking has only just begun.

The eBPF.io Community

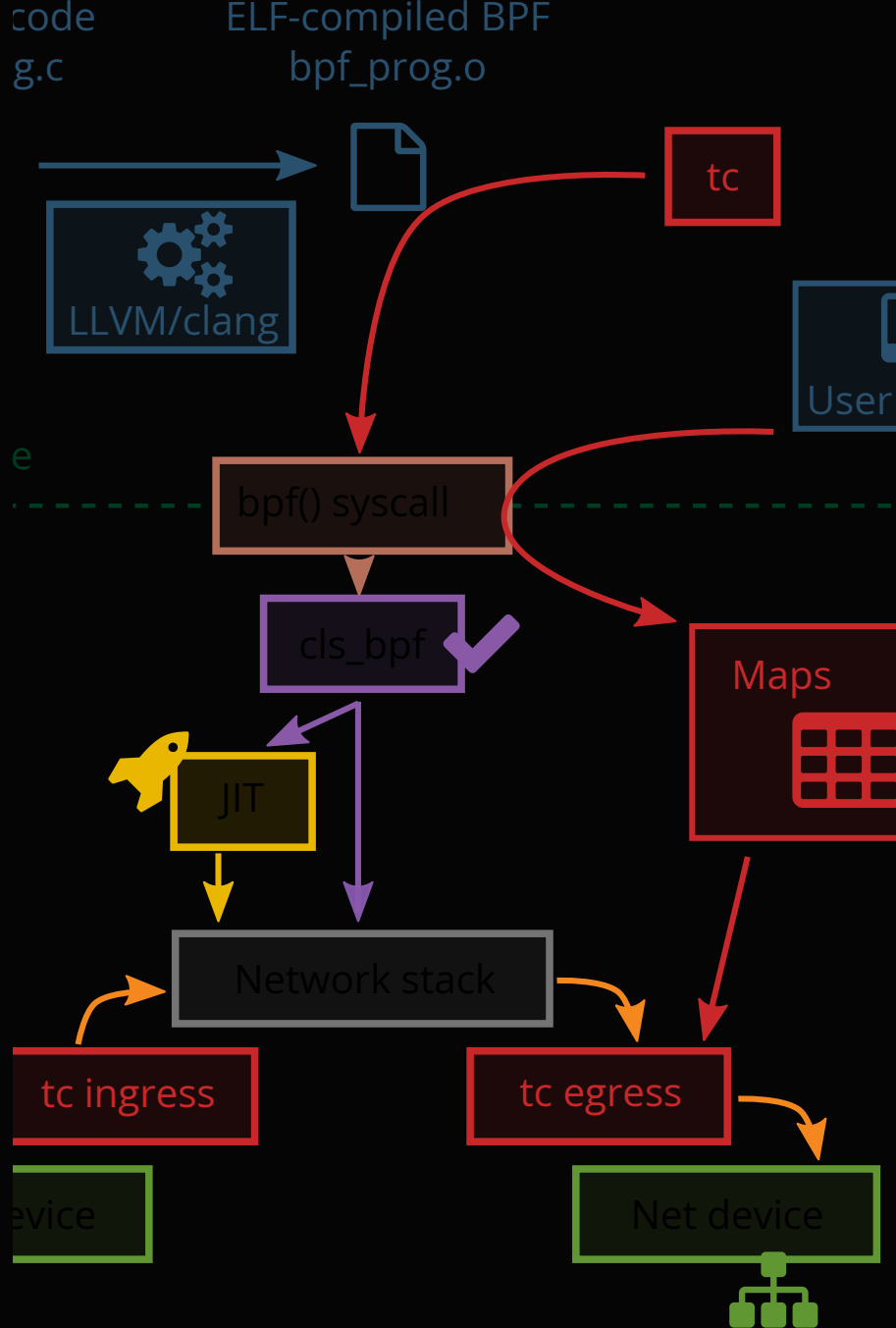
eBPF.io is a place for everybody to learn and collaborate on the topic of eBPF. eBPF is an open community and everybody can participate and share.

Whether you want to read a first introduction to eBPF, find further reading material, or make your first steps to becoming contributors to major eBPF projects, eBPF.io will help you along the way.

eBPF Architecture: Hook Overview

eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point. Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others.

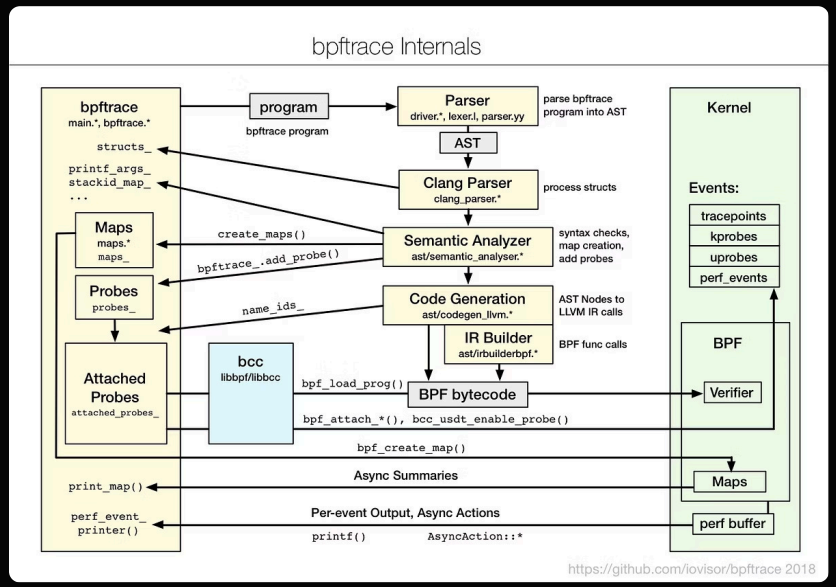
If a predefined hook does not exist for a particular need, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs almost anywhere in kernel or user applications.



System Call Hook Example

One common hook point is at the system call interface, where applications interact with the kernel. eBPF programs can intercept these calls to add functionality or gather data.

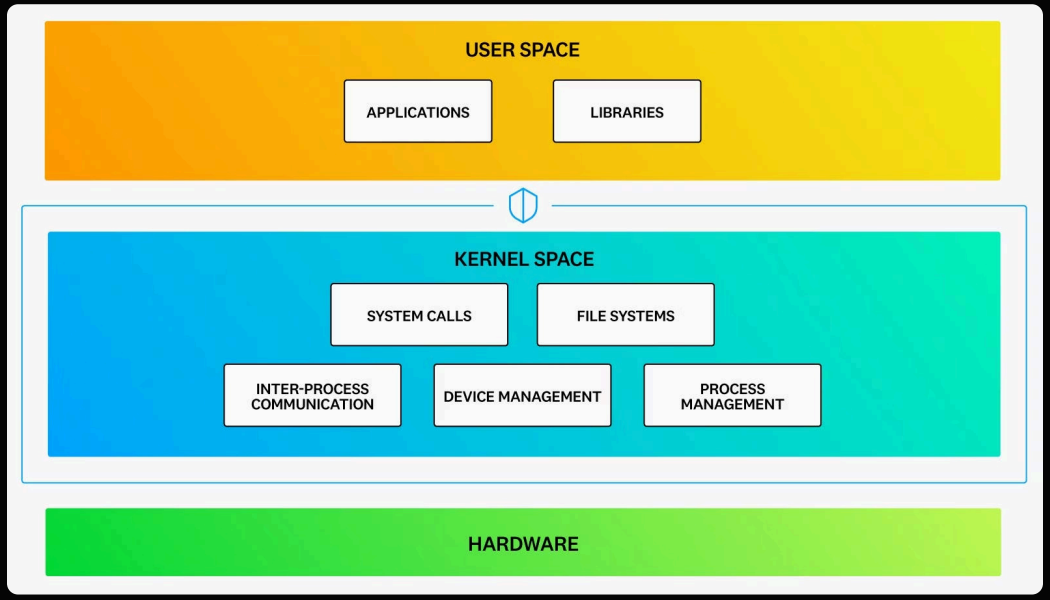
This allows for powerful capabilities like auditing system calls, enhancing security by validating parameters, or collecting performance metrics without modifying the kernel source code.



How are eBPF Programs Written?

In many scenarios, eBPF is not used directly but indirectly via projects like Cilium, bcc, or bpftrace which provide an abstraction on top of eBPF and do not require writing programs directly but instead offer the ability to specify intent-based definitions.

If no higher-level abstraction exists, programs need to be written directly. The Linux kernel expects eBPF programs to be loaded in the form of bytecode. While it is possible to write bytecode directly, the more common development practice is to leverage a compiler suite like LLVM to compile pseudo-C code into eBPF bytecode.



eBPF Program Lifecycle

Write Program

Develop eBPF program in C or using a higher-level abstraction like bcc or bpftrace.

Compile to Bytecode

Use LLVM/Clang to compile the program into eBPF bytecode.

Load into Kernel

Use the bpf system call to load the bytecode into the kernel.

Verification

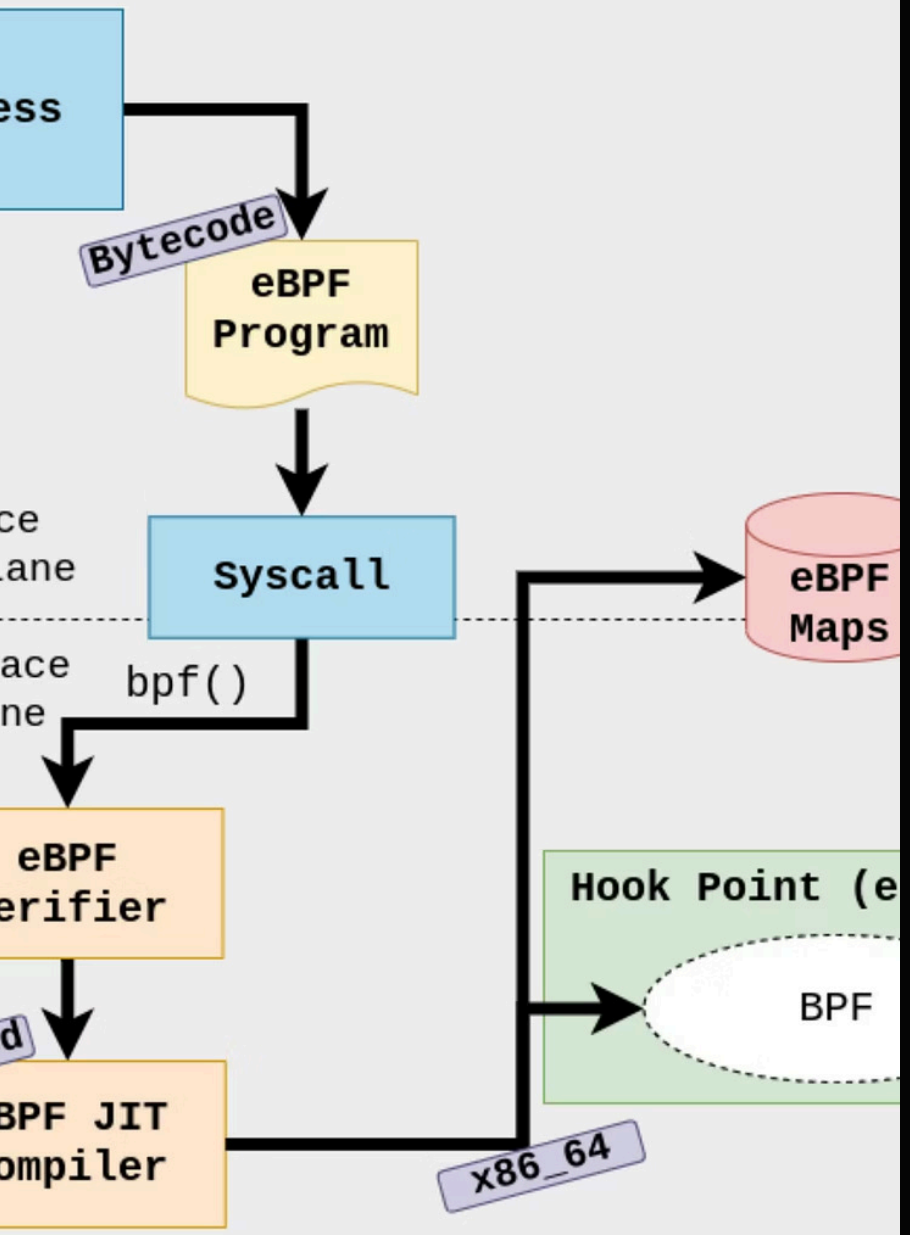
Kernel verifies the program is safe to run.

JIT Compilation

Program is compiled to native machine code for optimal performance.

Attach to Hook

Program is attached to a hook point and begins execution when triggered.



Loader & Verification Architecture

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

As the program is loaded into the Linux kernel, it passes through verification and JIT compilation before being attached to the requested hook.

Verification: Ensuring Safety

The verification step ensures that the eBPF program is safe to run. It validates that the program meets several conditions:

Privilege Check

The process loading the eBPF program holds the required capabilities (privileges). Unless unprivileged eBPF is enabled, only privileged processes can load eBPF programs.

Safety Analysis

The program does not crash or otherwise harm the system through invalid memory access or other dangerous operations.

Termination Guarantee

The program always runs to completion (i.e. the program does not sit in a loop forever, holding up further processing).

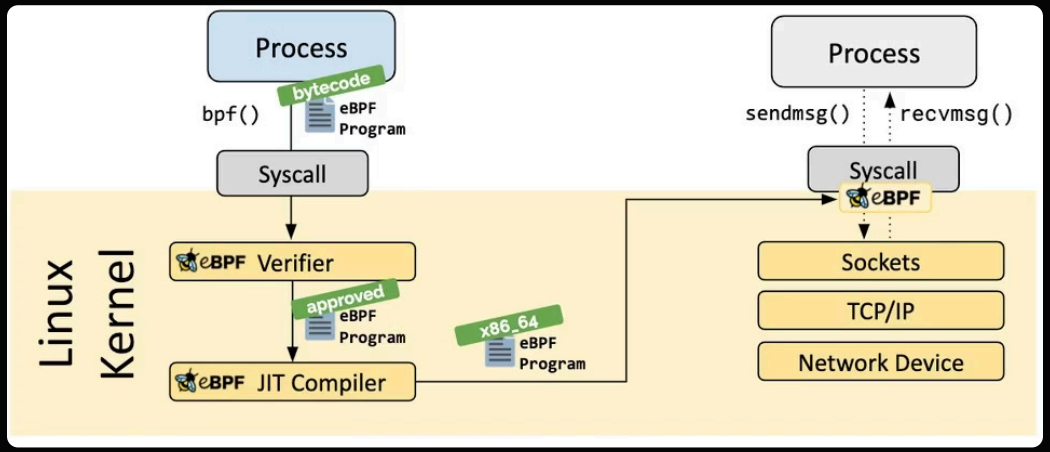
Complexity Limits

Program must have a finite complexity. The verifier will evaluate all possible execution paths and must be capable of completing the analysis within the limits of the configured upper complexity limit.

JIT Compilation: Optimizing Performance

The Just-in-Time (JIT) compilation step translates the generic bytecode of the program into the machine specific instruction set to optimize execution speed of the program.

This makes eBPF programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module.

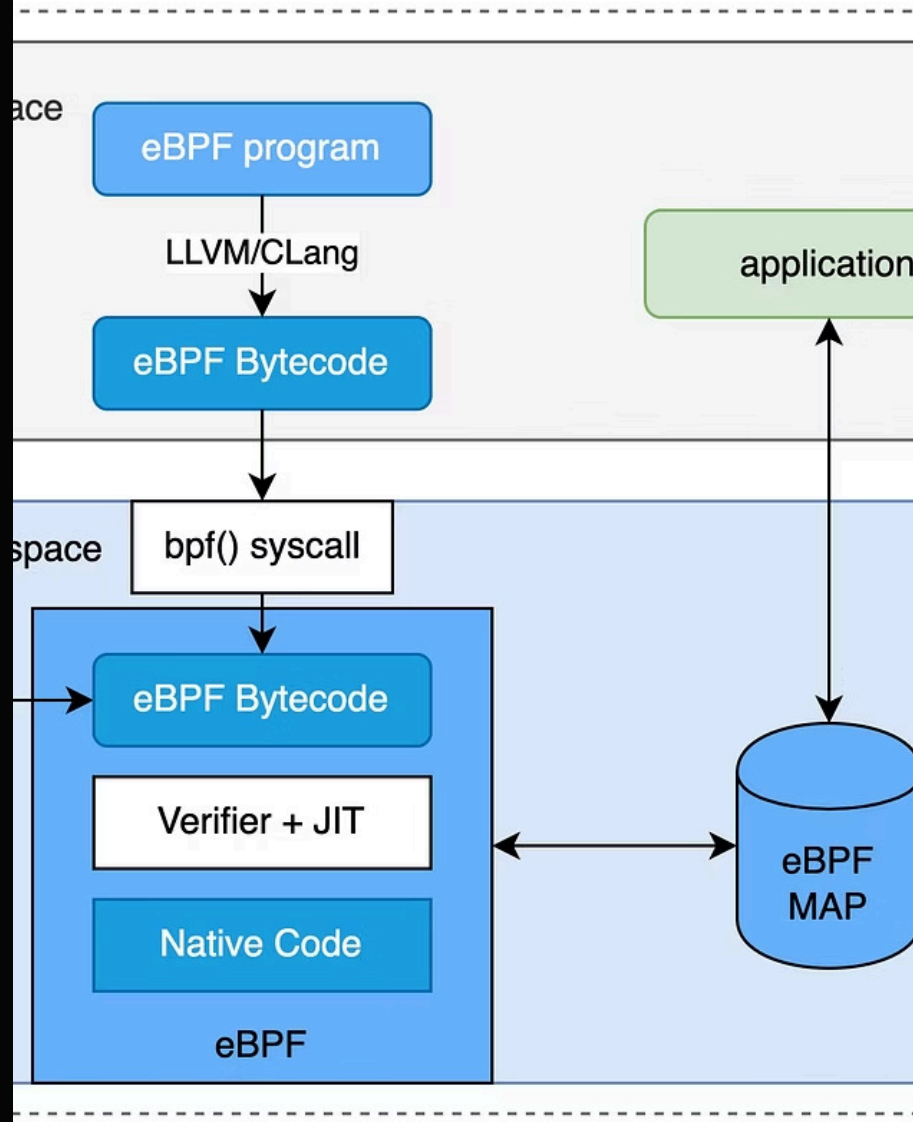


The JIT compiler ensures that eBPF programs have minimal overhead, making them suitable for performance-critical applications like networking and security.

eBPF Maps: Sharing Data

A vital aspect of eBPF programs is the ability to share collected information and to store state. For this purpose, eBPF programs can leverage the concept of eBPF maps to store and retrieve data in a wide set of data structures.

eBPF maps can be accessed from eBPF programs as well as from applications in user space via a system call, enabling efficient communication between kernel and user space.



Types of eBPF Maps

Hash Tables & Arrays

Basic data structures for key-value storage with different access patterns.

LRU (Least Recently Used)

Automatically evicts least recently used entries when capacity is reached.

Ring Buffer

Circular buffer for efficient producer-consumer communication patterns.

Stack Trace

Specialized map for storing program execution stack traces.

LPM (Longest Prefix Match)

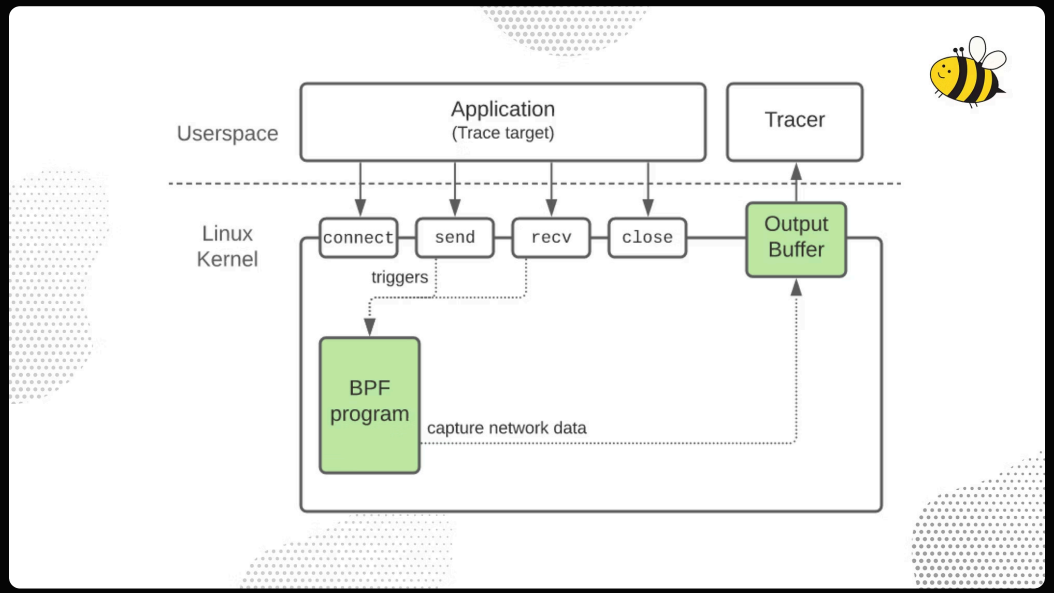
Optimized for IP routing and similar prefix-based lookups.

For various map types, both a shared and a per-CPU variation is available to optimize for different access patterns.

Helper Calls: Safe API Access

eBPF programs cannot call into arbitrary kernel functions. Allowing this would bind eBPF programs to particular kernel versions and would complicate compatibility of programs.

Instead, eBPF programs can make function calls into helper functions, a well-known and stable API offered by the kernel.



Available Helper Functions

The set of available helper calls is constantly evolving. Examples of available helper calls include:

Random Number Generation

Generate random numbers for use in eBPF programs.

Time & Date Functions

Get current time & date for timestamping or timing operations.

Map Access

eBPF map operations for storing and retrieving data.

Context Access

Get process/cgroup context for security and monitoring.

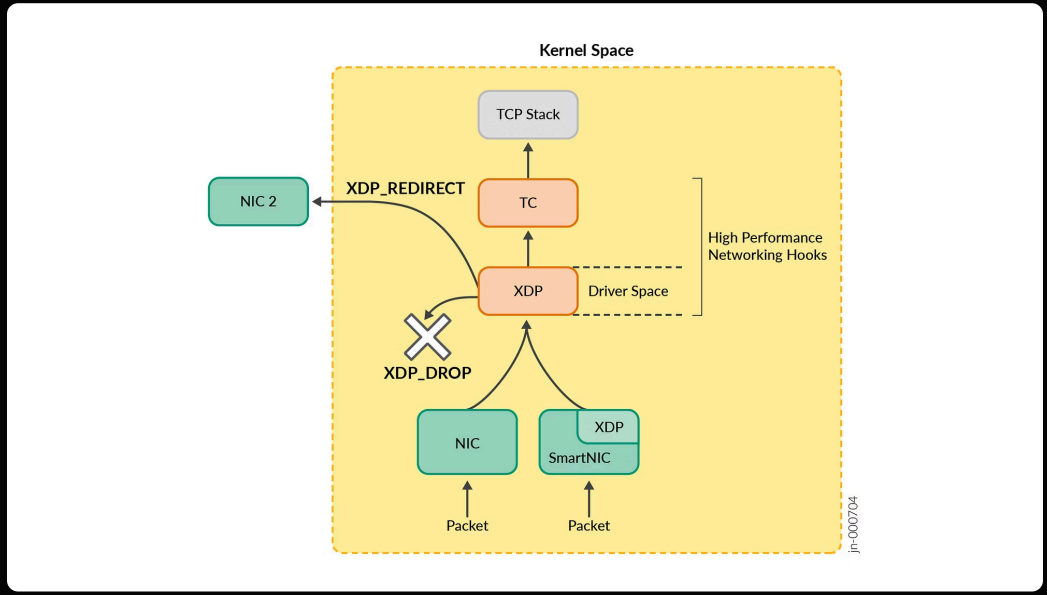
Network Packet Manipulation

Manipulate network packets and forwarding logic.

Tail & Function Calls: Composable Programs

eBPF programs are composable with the concept of tail and function calls:

- **Function calls** allow defining and calling functions within an eBPF program
- **Tail calls** can call and execute another eBPF program and replace the execution context, similar to how the execve() system call operates for regular processes



This composability allows for complex program logic to be split into manageable, reusable components.

eBPF Safety: A Critical Priority

"With great power there must also come great responsibility."

eBPF is an incredibly powerful technology and now runs at the heart of many critical software infrastructure components. During the development of eBPF, safety was the most crucial aspect when eBPF was considered for inclusion into the Linux kernel.

eBPF Safety Layers

Required Privileges

Unless unprivileged eBPF is enabled, all processes that intend to load eBPF programs into the Linux kernel must be running in privileged mode (root) or require the capability CAP_BPF.

Hardening

Upon successful verification, the eBPF program runs through a hardening process including program execution protection, Spectre mitigation, and constant blinding.

Verifier

All programs pass through the eBPF verifier, which ensures the safety of the program itself, checking for bounded loops, memory safety, and program complexity.

Abstracted Runtime Context

eBPF programs cannot access arbitrary kernel memory directly. Data structures outside the program context must be accessed via eBPF helpers.

Unprivileged eBPF

If unprivileged eBPF is enabled, unprivileged processes can load certain eBPF programs subject to a reduced functionality set and with limited access to the kernel.

This allows for broader use of eBPF while maintaining system security, but with significant restrictions compared to privileged eBPF programs.

Verifier: The Safety Guardian

The eBPF verifier ensures the safety of the program itself. This means, for example:

Termination Guarantee

Programs are validated to ensure they always run to completion. eBPF programs may contain bounded loops but the program is only accepted if the verifier can ensure that the loop contains an exit condition which is guaranteed to become true.

Memory Safety

Programs may not use any uninitialized variables or access memory out of bounds, preventing potential crashes or security vulnerabilities.

Size Limits

Programs must fit within the size requirements of the system. It is not possible to load arbitrarily large eBPF programs.

Complexity Bounds

Program must have a finite complexity. The verifier will evaluate all possible execution paths and must be capable of completing the analysis within the limits of the configured upper complexity limit.

Verifier vs. Security

Verifier Focus

The verifier is meant as a safety tool, checking that programs are safe to run. It ensures programs won't crash the kernel or get stuck in infinite loops.

Security Considerations

The verifier is not a security tool inspecting what the programs are doing. Security comes from the privilege requirements, hardening process, and abstracted runtime context.

Hardening Process

Upon successful completion of verification, the eBPF program runs through a hardening process according to whether the program is loaded from a privileged or unprivileged process.

Program Execution Protection

The kernel memory holding an eBPF program is protected and made read-only. If the eBPF program is attempted to be modified, the kernel will crash instead of allowing it to continue executing the corrupted/manipulated program.

Mitigation Against Spectre

eBPF programs mask memory access to redirect access under transient instructions to controlled areas. The verifier follows program paths accessible only under speculative execution, and the JIT compiler emits Retpolines when needed.

Constant Blinding

All constants in the code are blinded to prevent JIT spraying attacks. This prevents attackers from injecting executable code as constants which could allow an attacker to jump into the memory section of the eBPF program.

Abstracted Runtime Context

eBPF programs cannot access arbitrary kernel memory directly. Data and data structures that lie outside of the context of the program must be accessed via eBPF helpers.

This guarantees consistent data access and makes any such access subject to the privileges of the eBPF program.

An eBPF program cannot randomly modify data structures in the kernel, providing an important security boundary.

Why eBPF? The Power of Programmability

Let's start with an analogy. Do you remember GeoCities? 20 years ago, web pages used to be almost exclusively written in static markup language (HTML). A web page was basically a document with an application (browser) able to display it.

Looking at web pages today, web pages have become full-blown applications and web-based technology has replaced a vast majority of applications written in languages requiring compilation.

What enabled this evolution? The short-answer is programmability with the introduction of JavaScript. It unlocked a massive revolution resulting in browsers evolving into almost independent operating systems.

Key Aspects of Programmability Revolution

Safety

Untrusted code runs in the browser of the user. This was solved by sandboxing JavaScript programs and abstracting access to browser data.

Continuous Delivery

Evolution of program logic must be possible without requiring to constantly ship new browser versions. This was solved by providing the right low-level building blocks.

Performance

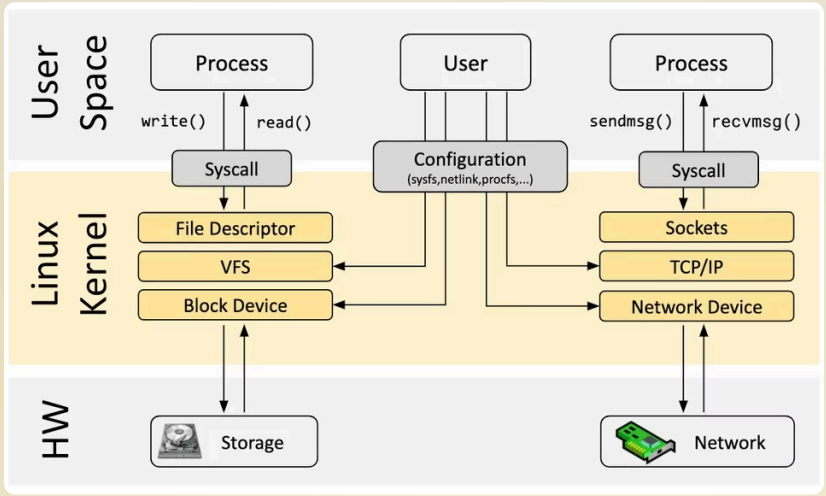
Programmability must be provided with minimal overhead. This was solved with the introduction of a Just-in-Time (JIT) compiler.

For all of the above, exact counterparts can be found in eBPF for the same reason.

eBPF's Impact on the Linux Kernel

In order to understand the programmability impact of eBPF on the Linux kernel, it helps to have a high-level understanding of the architecture of the Linux kernel and how it interacts with applications and the hardware.

The main purpose of the Linux kernel is to abstract the hardware or virtual hardware and provide a consistent API (system calls) allowing for applications to run and share the resources.



Traditional Kernel Extension Options

1

Native Support

- Change kernel source code and convince the Linux kernel community that the change is required
- Wait several years for the new kernel version to become a commodity

2

Kernel Module

- Write a kernel module
- Fix it up regularly, as every kernel release may break it
- Risk corrupting your Linux kernel due to lack of security boundaries

3

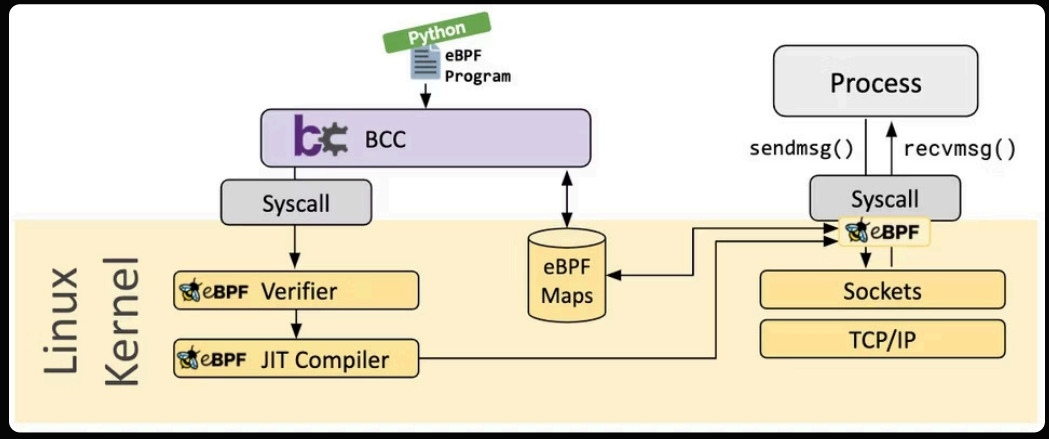
eBPF Approach

- Write a sandboxed eBPF program
- Load it into the kernel at runtime
- Benefit from safety guarantees and performance optimization

Development Toolchains: BCC

BCC is a framework that enables users to write python programs with eBPF programs embedded inside them.

The framework is primarily targeted for use cases which involve application and system profiling/tracing where an eBPF program is used to collect statistics or generate events and a counterpart in user space collects the data and displays it in a human readable form.

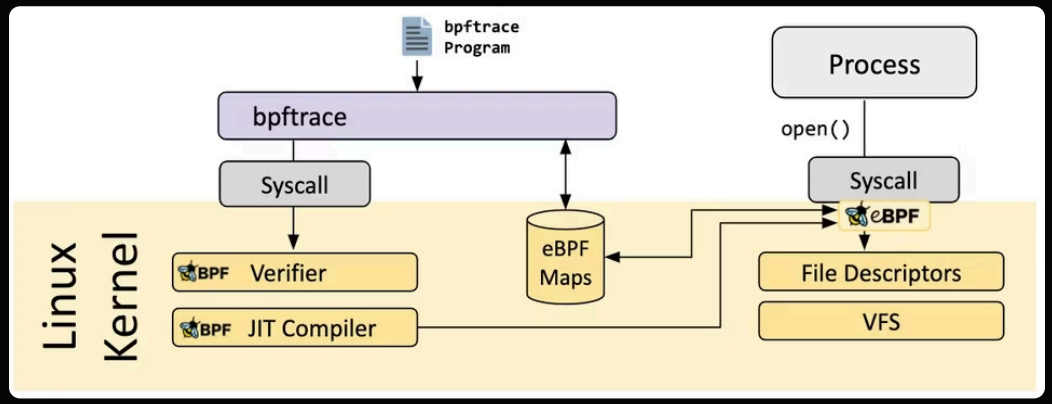


Running the python program will generate the eBPF bytecode and load it into the kernel.

Development Toolchains: bpftrace

bpftrace is a high-level tracing language for Linux eBPF and available in semi-recent Linux kernels (4.x).

bpftrace uses LLVM as a backend to compile scripts to eBPF bytecode and makes use of BCC for interacting with the Linux eBPF subsystem as well as existing Linux tracing capabilities: kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes), and tracepoints.

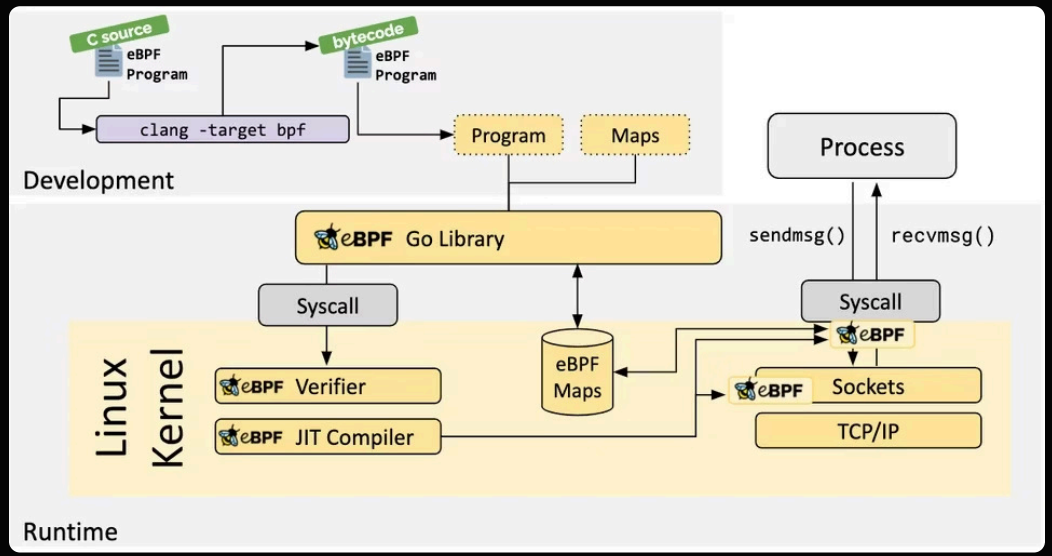


The bpftrace language is inspired by awk, C, and predecessor tracers such as DTrace and SystemTap.

Development Toolchains: eBPF Go Library

The eBPF Go library provides a generic eBPF library that decouples the process of getting to the eBPF bytecode and the loading and management of eBPF programs.

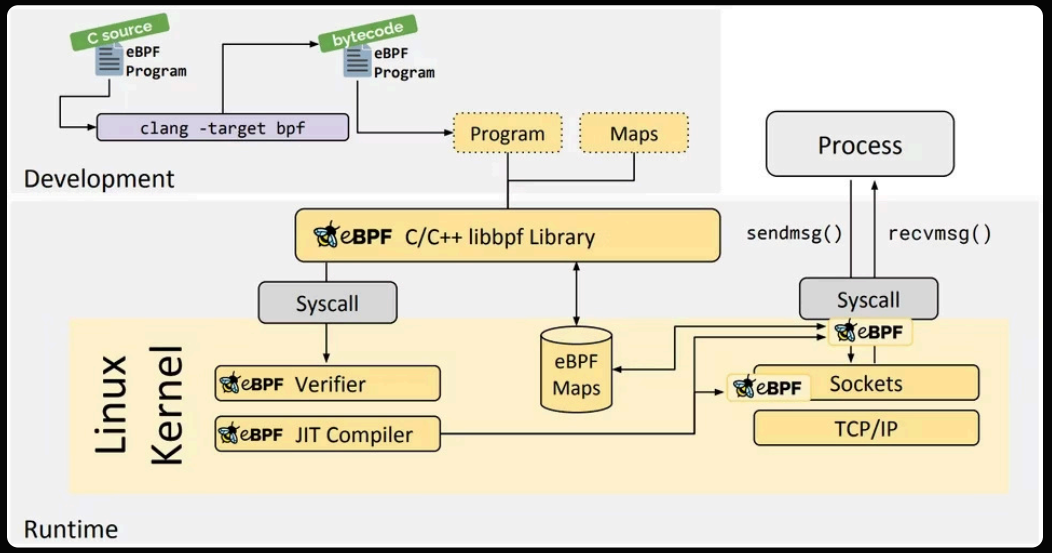
eBPF programs are typically created by writing a higher level language and then use the clang/LLVM compiler to compile to eBPF bytecode.



Development Toolchains: libbpf C/C++ Library

The libbpf library is a C/C++-based generic eBPF library which helps the loading of eBPF object files generated from the clang/LLVM compiler into the kernel.

It generally abstracts interaction with the BPF system call by providing easy to use library APIs for applications.



Major eBPF Projects

Cilium

eBPF-based networking, security, and observability for cloud native environments.

BCC

Tools for BPF-based Linux IO analysis, networking, monitoring, and more.

bpftrace

High-level tracing language for Linux eBPF.

Hubble

Network, Service & Security Observability for Kubernetes using eBPF.

These projects provide higher-level abstractions that make eBPF more accessible to users with different needs and skill levels.

Cilium: eBPF-Based Networking

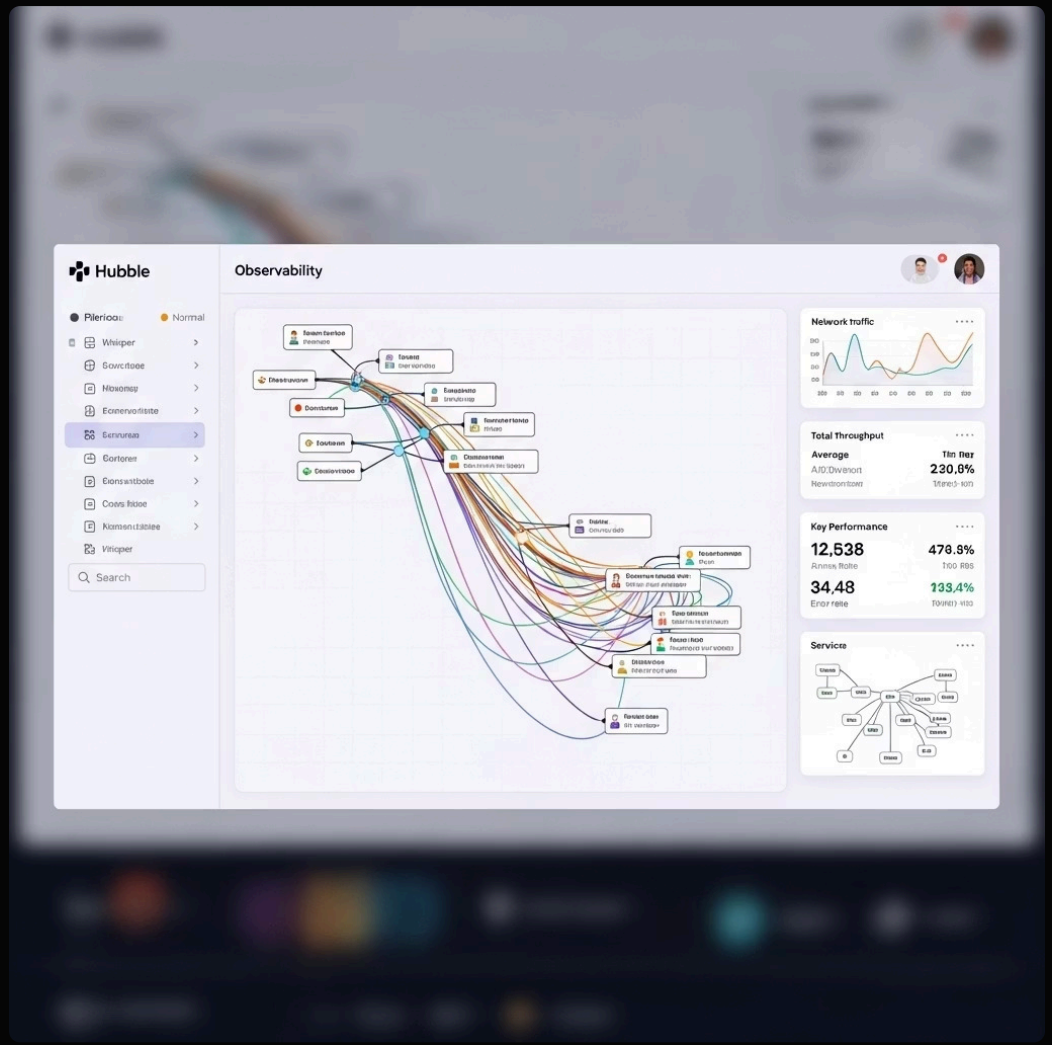
Cilium is an open source project that provides networking, security, and observability for cloud native environments such as Kubernetes clusters and other container orchestration platforms.

It leverages eBPF to provide high-performance networking, load balancing, and security policy enforcement without the overhead of traditional solutions like iptables.

Hubble: eBPF-Based Observability

Hubble is a fully distributed networking and security observability platform for cloud native workloads. It is built on top of Cilium and eBPF to enable deep visibility into the communication and behavior of services.

Hubble provides service dependency mapping, operational monitoring and alerting, and application monitoring with security observability at the network level.



Learning Resources: Documentation

eBPF Docs

Technical documentation for eBPF at docs.ebpf.io

BPF & XDP Reference Guide

Comprehensive guide in the Cilium Documentation

BPF Documentation

Official BPF Documentation in the Linux Kernel

BPF Design Q&A

FAQ for kernel-related eBPF questions

Learning Resources: Tutorials

Learn eBPF Tracing

Tutorial and Examples from Brendan Gregg's Blog

XDP Hands-On Tutorials

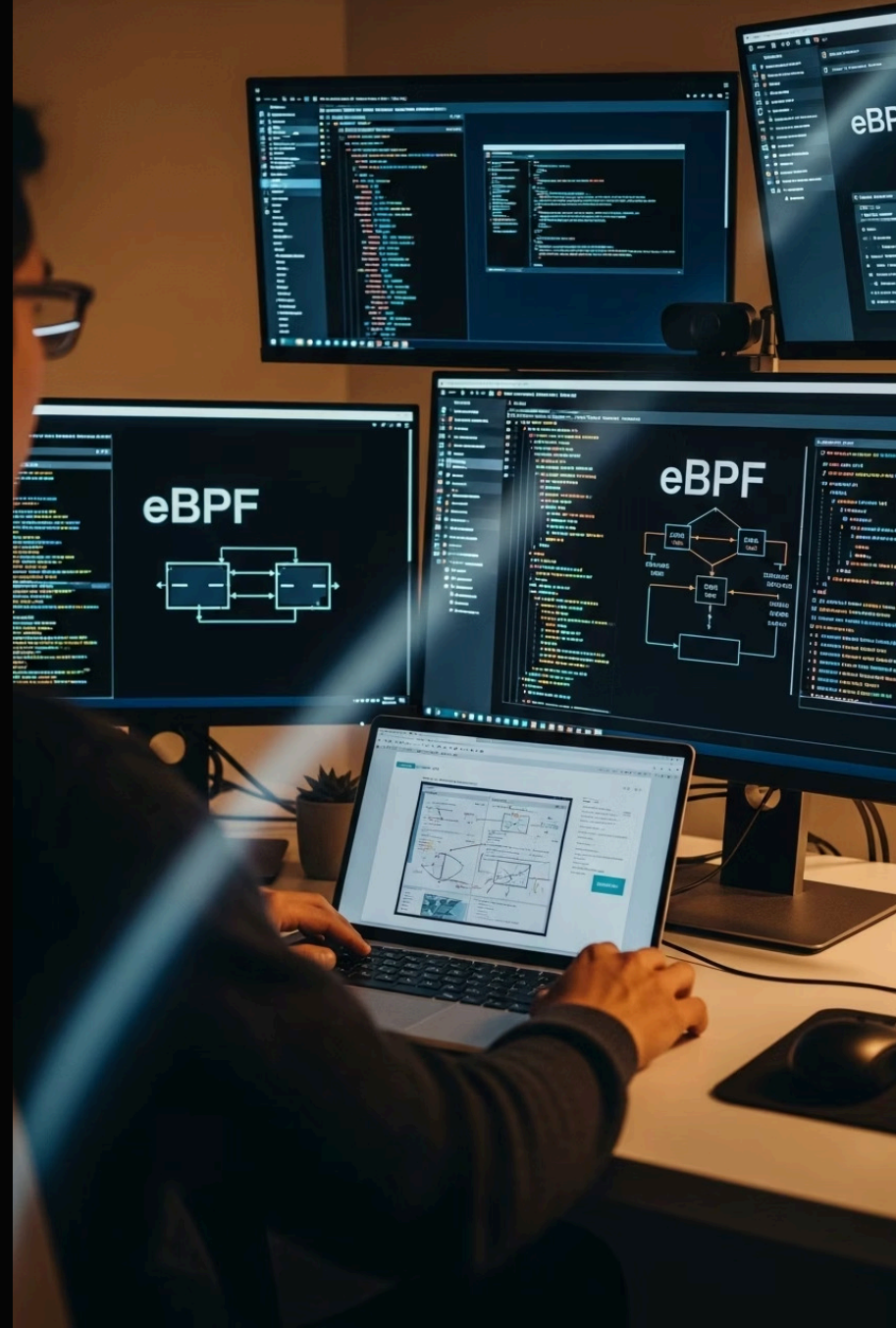
Practical tutorials for XDP (eXpress Data Path) programming

BCC, libbpf and BPF CO-RE Tutorials

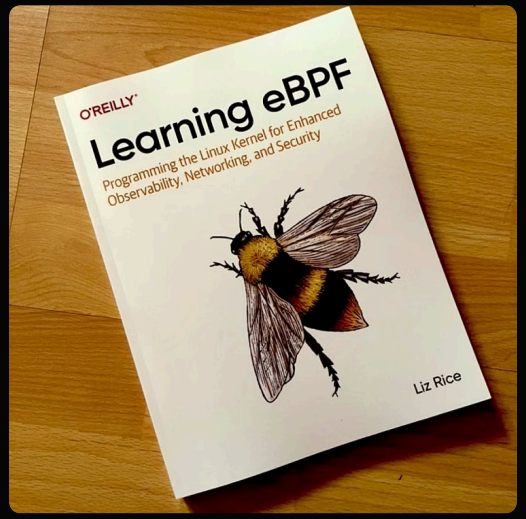
Tutorials from Facebook's BPF Blog

eBPF Tutorial

Learning eBPF Step by Step with Examples

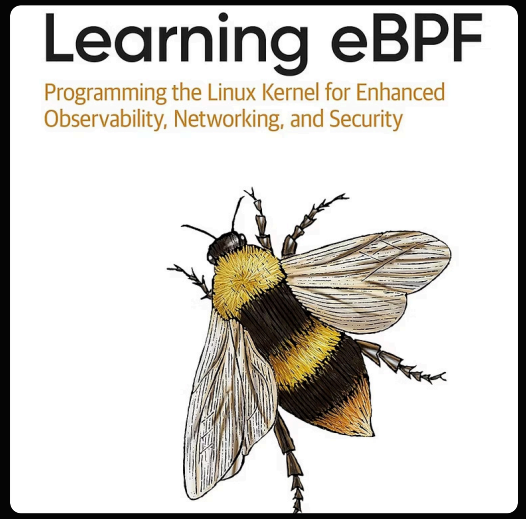


Learning Resources: Books



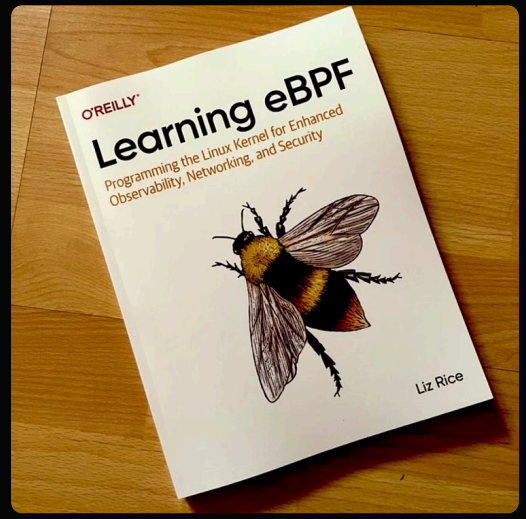
Learning eBPF

By Liz Rice, O'Reilly, 2023



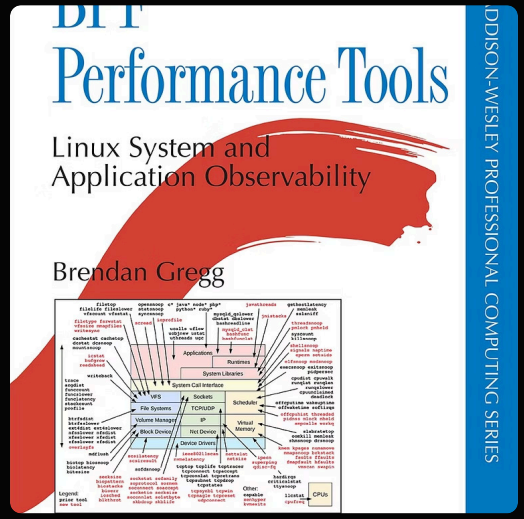
Security Observability with eBPF

By Natália Réka Ivánkó and Jed Salazar, O'Reilly, 2022



What is eBPF?

By Liz Rice, O'Reilly, 2022



BPF Performance Tools

By Brendan Gregg, Addison-Wesley Professional Computing Series, 2019

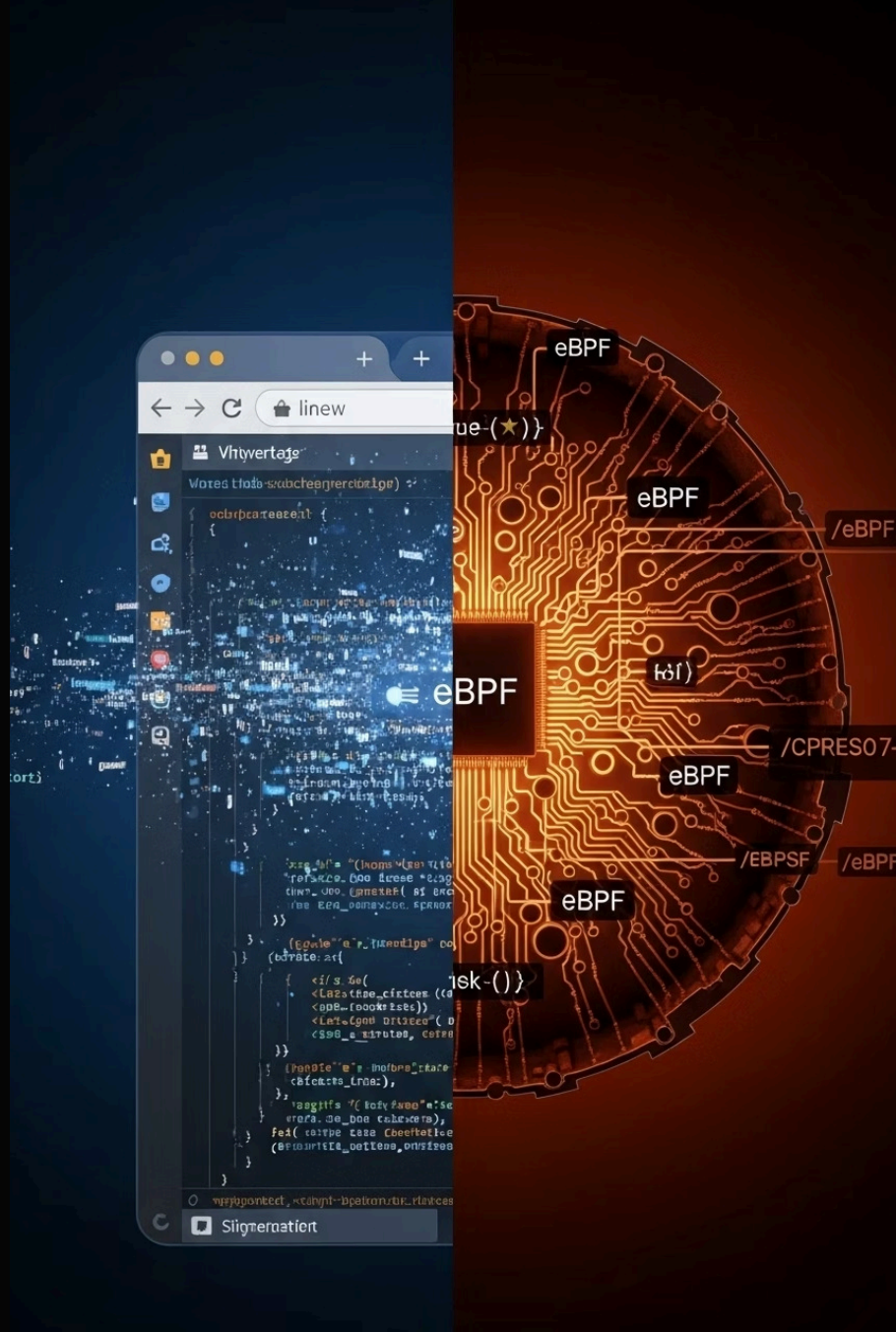
The JavaScript Analogy Revisited

JavaScript Revolution

- Enabled programmability in browsers
- Decoupled application innovation from browser release cycles
- Created a safe sandbox for untrusted code
- Used JIT compilation for performance

eBPF Revolution

- Enables programmability in the kernel
- Decouples innovation from kernel release cycles
- Creates a safe sandbox for kernel extensions
- Uses JIT compilation for performance



The Future of eBPF

eBPF is still in its early stages of adoption, with new use cases and capabilities emerging regularly. The future holds exciting possibilities:

Expanded Kernel Capabilities

More hooks and helper functions to access additional kernel subsystems

Simplified Development

Better tooling and higher-level abstractions to make eBPF more accessible

Cross-Platform Support

Efforts to bring eBPF to other operating systems beyond Linux

New Application Domains

Expansion into areas like machine learning, storage optimization, and more

bpftrace: High-Level eBPF Front-End

What is bpftrace?

bpftrace is a high-level tracing language for eBPF, designed for one-liners and short scripts. It's inspired by tools like DTrace and provides a powerful yet accessible way to utilize eBPF for system analysis.

Key Features:

- Awk-like syntax for concise expression
- Rich built-in functions and variables
- Support for maps, histograms, and statistics
- Access to kernel and userspace probe points
- Strong safety guarantees via eBPF verifier

Basic Syntax:

```
probe_type:probe_point /filter/ {
    action
}
```

Simple Example:

```
# Trace open syscall and count by processes
bpftrace -e 'tracepoint:syscalls:sys_enter_open* {
 @[comm] = count();
}'
```

bpftrace vs. Other Tracing Tools

Tool	Advantages	Limitations	Use Cases
strace	Simple to use, no kernel requirements	High overhead, syscalls only, process-specific	Quick debugging of application syscalls
perf	Sampling profiler, low overhead	Limited customisation, fixed event types	CPU profiling, standard event counting
ftrace	Built into kernel, function tracing	Clunky interface, limited data processing	Kernel function call tracing
BCC	Full eBPF power, Python front-end	More complex to write, requires Python	Complex, custom tracing tools
bpftrace	Concise syntax, powerful data processing	Not for long-running agents, less performance than BCC	Ad-hoc investigations, custom metrics

bpftrace strikes an excellent balance between ease-of-use and power, making it ideal for interactive performance analysis and debugging.

Probe Types in bpftrace

kprobe/kretprobe

Kernel function entry/return probes

```
# Trace disk read/write functions with size
bpftrace -e 'kprobe:blk_account_io_* {
    printf("%s %s %d bytes\n",
           probe, comm, arg2);
}'
```

uprobe/uretprobe

User-level function entry/return probes

```
# Trace malloc() calls in libc
bpftrace -e 'uprobe:/lib/x86_64-linux-gnu/libc.so.6:malloc
/comm != "bpftrace"/ {
    printf("malloc(%d) by %s\n", arg0, comm);
}'
```

tracepoint

Stable kernel tracing points

```
# Trace process executions
bpftrace -e 'tracepoint:syscalls:sys_enter_execve {
    printf("%s executed\n", str(args->filename));
}'
```

usdt

User-level statically defined tracepoints

```
# Trace Node.js GC events
bpftrace -e 'usdt:/usr/local/bin/node:gc__start {
 printf("Node.js GC: %s\n", comm);
}'
```

More Probe Types

profile

Timer-based sampling

```
# Sample stack traces at 99 Hz
bpftrace -e 'profile:hz:99 {
    @[kstack] = count();
}'
```

interval

Periodic events

```
# Print memory info every 5 seconds
bpftrace -e 'interval:s:5 {
    printf("Free memory: %d MB\n",
        kstack->free_pages * 4 / 1024);
}'
```

software/hardware

Software/hardware events (from perf)

```
# Count CPU cache misses by process
bpftrace -e 'hardware:cache-misses:1000000 {
    @[comm] = count();
}'
```

BEGIN/END

Program start/end special probes

```
# Setup and cleanup
bpftrace -e 'BEGIN { printf("Tracing started...\n"); }
END { printf("Tracing complete.\n"); }'
```

Built-in Variables

Variable	Type	Description	Example
pid	Integer	Process ID	`if (pid == 1234) { ... }`
tid	Integer	Thread ID	`@threads[tid] = count();`
comm	String	Process name	`printf("Process: %s\n", comm);`
nsecs	Integer	Timestamp in nanoseconds	`@start[tid] = nsecs;`
cpu	Integer	Current CPU	`@load[cpu] = count();`
args	Structure	Tracepoint arguments	`args->filename`
arg0...argN	Various	Probe arguments	`printf("size: %d", arg2);`
retval	Integer	Return value (kretprobe/uretprobe)	`if (retval < 0) { ... }`

```
# Example using multiple variables
bpftrace -e 'tracepoint:syscalls:sys_exit_read /pid == 181/ {
 printf("read() returned %d bytes by %s on CPU %d\n",
 args->ret, comm, cpu);
}'
```

Data Types and Maps

Basic Data Types:

- Integer types (32 and 64-bit)
- Strings (char arrays)
- Pointers
- Structs (for kernel data structures)

Maps:

Maps are key-value stores prefixed with @ that persist throughout program execution.

Map Operations:

- Counters: `@count++`
- Key-value: `@[key] = value`
- Multiple keys: `@[key1, key2] = value`
- Histograms: `@h = hist(value)`
- String maps: `@str[pid] = comm`

Example: Counting File Opens by Process

```
#!/usr/bin/env bpftrace

BEGIN {
    printf("Counting file opens by process...\n");
}

tracepoint:syscalls:sys_enter_openat {
    @opens[comm] = count();
}

END {
    printf("File open counts by process:\n");
    print(@opens);
}
```

Output:

```
Counting file opens by process...
^C
File open counts by process:
@opens[systemd]: 4
@opens[bash]: 7
@opens[chrome]: 132
```

Filtering and Predicates

Basic Filtering

bpftrace allows you to filter events using predicates - expressions inside forward slashes:

```
probe /predicate/ { actions }
```

Common Filter Operations:

- Equality: `==`, `!=`
- Comparison: `<`, `>`, `<=`, `>=`
- Logical: `&&`, `||`, `!`
- String comparison: `==`, `!=`
- Bitwise: `&`, `|`, `^`
- String matching: `strncmp()`, `strchr()`

Example: Trace Large Reads Only

```
#!/usr/bin/env bpftrace

// Trace read() syscalls with buffers > 1MB
tracepoint:syscalls:sys_enter_read /args->count > 1024*1024/ {
    printf("Large read: %s (%d) reading %d bytes\n",
           comm, pid, args->count);
}
```

Example: Filter by Process and Error

```
#!/usr/bin/env bpftrace

// Only trace failed opens in nginx process
tracepoint:syscalls:sys_exit_openat
/comm == "nginx" && args->ret < 0/ {
 printf("Failed open: %s (%d) error %d\n",
 comm, pid, args->ret);
 @errors[args->ret] = count();
}
```

Output Formatting Functions

printf()

Formatted output with C-style format strings

```
printf("PID: %d, Size: %d bytes\n", pid, arg2);
```

time()

Format timestamp

```
printf("Time: %s\n", time("%H:%M:%S"));
```

join()

Join array elements into a string

```
printf("Command: %s\n", join(args->argv, " "));
```

str()

Convert pointer to string

```
printf("File: %s\n", str(args->filename));
```

print()

Print map contents

```
print(@syscalls);
```

clear()

Clear a map

```
clear(@start);
```

Example Script with Formatted Output:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("%-16s %-6s %-16s %s\n", "TIME", "PID", "PROCESS", "FILE");
}

tracepoint:syscalls:sys_enter_openat {
 printf("%-16s %-6d %-16s %s\n",
 time("%H:%M:%S"), pid, comm, str(args->filename));
}
```

Lab: Your First bpftrace Scripts

Exercise 1: Hello World

```
#!/usr/bin/env bpftrace

BEGIN {
    printf("Hello from bpftrace!\n");
}

tracepoint:syscalls:sys_enter_write {
    printf("%s wrote %d bytes\n",
           comm, args->count);
}

END {
    printf("Goodbye!\n");
}
```

Exercise 2: Count Syscalls by Process

```
#!/usr/bin/env bpftrace

BEGIN {
    printf("Counting syscalls by process...\n");
}

tracepoint:raw_syscalls:sys_enter {
    @syscalls[comm] = count();
}

interval:s:5 {
    time("%H:%M:%S\n");
    print(@syscalls);
    clear(@syscalls);
}
```

Exercise 3: Monitor File Opens

```
#!/usr/bin/env bpftrace

BEGIN {
    printf("%-16s %-6s %-16s %s\n",
           "TIME", "PID", "COMM", "FILENAME");
}

tracepoint:syscalls:sys_enter_openat {
    printf("%-16s %-6d %-16s %s\n",
           time("%H:%M:%S"),
           pid,
           comm,
           str(args->filename));
}
```

Challenge: Add a Filter

Modify the file open monitor to only show:

- Files opened by a specific process
- Files with specific extensions (.log, .conf)
- Files that are in /etc/ directory