BPF Technical Roadmap - 2022

tinyurl.com/bpfroadmap2022

[raw notes, non-final work in progress document]

Goal

The goal of the document is to establish a long-term technical roadmap for BPF.

The BSC expects the document to steer the technical direction of various projects in the BPF landscape, encouraging new projects that are aligned with the roadmap which could potentially be funded by the foundation.

Areas

Cross platform support

Support a matrix of architectures (Intel, Arm, RISC-V, etc.) and operating systems (Linux, Windows, BSD, etc.).

Including hardware offload into various cards

Including userland ebpf program execution (at least on userland runtimes)

Work towards feature parity (incl. JITs).

Security

Integrity, security policy and signatures

- Signatory service in the ecosystem
- Reference gatekeeper implementation
- Verifier BPF callbacks

Comprehensive LSM implementation using eBPF

- SELinux security labels (?)
- Enforcement helpers?
    - Close a socket
    - Sending signals is already possible

Attestation state related to ebpf

Long term future of unprivileged eBPF

One of the use-cases for unprivileged is container observability tools

When would processors change enough

Different address space for BPF programs, sufficiently separated.

KPTI-like memory isolation.

Auditing using BPF

- Audit
    - Syslog message from BPF or write to dmesg.
    - BPF programs on Linux audit events

Auditability of eBPF programs.

How can you get a sense of “What has changed in the kernel since a BPF program has been loaded”

Something like bpfsnoop is needed. Or a way to have bpf prog attach messages go to dmesg. Load, attach, delete. Look at LSM hook.

Compiler and libraries

Feature parity between LLVM and gcc BPF backends.

Ensure both compilers are doing the same thing. Compiler correctness test suite.

Avoid a situation where one compiler generated byte-code passes the verifier and the other does not.

Is it even practical?

selftests is the bar.

Trigger self tests on compiler patches.

Libraries

Expectations of a BPF loader.

Move to guidelines: Don’t make more loaders if there is already one

Section names are libbpf specific.

ELF format standardization.
CO-RE format standardization

Core

Verifier

Formal model of the entire verifier logic

Verifier fuzzing tests

Establish the verifier behaviors and document them for better cross-platform support.

Comprehensive verifier & JIT test suites and continued formal verification and audit effort.

Formal verification of ALU operations.

TNUM is (mostly) formally verified.

Simplify / refactor verifier code e.g. type handling.

Comprehensive BTF test suite and security audit.

Make verifier errors more user-accessible. Take hints from Rust on how to make good error messages?

Tutorial on how to get over verifier errors. (example of tutorial for verifier errors on windows here)

Verifier analysis framework to be able to reason about behavior (e.g. pruning decisions).

Code coverage for self tests

Concepts of verifier
Tinyurls for error message explanations

JIT compilers

Support for various hardware architectures and register calling conventions

JIT feature parity among architectures.

Optimizations of existing JITs.

Pack allocator (Linux) for multi-arch.

Feasibility check -

Windows uses uBPF JIT

Specification

Architecture specification and related conformance test suite that could be shared between platforms.

Optimizations

Reconsider hashing algorithms under the hood for map implementations

BPF CI

Kernel/OS

Multi-architecture testing (x86, s390x, arm64, riscv, ppc, etc).

Networking

XDP

Establish baseline performance reports like in https://core.dpdk.org/perf-reports/ to ensure we are not regressing with new kernel versions and to have vendors compete more for improving their drivers around XDP. → eBPF foundation

Compliance tests for XDP.

Also lab with different HW for XDP testing might be interesting e.g. for technical projects and development spanning across drivers (e.g. multi-buffer XDP). → eBPF foundation

XDP feature parity for major drivers & documentation of requirements from BSC side.

Developer Excellence

Documentation

Explain the various eBPF program and map types, explain their usage in easily accessible documentation with sample code so that one does not really need to check out the kernel to develop BPF programs.

Libraries

Easy to consume code examples for getting started from the eBPF libraries side (libbpf, cilium/ebpf, etc).

BPF standard library, aka C headers with common definitions (cls / xdp abstraction, BPF helper definitions, CO-RE macros, etc.)

Tooling

Complexity analysis for eBPF program development

BPF debuggers (e.g. edb)

IDE Support

Helper function name autocomplete

Guardrails to avoid pitfalls (possibly related to observability, which functions should not be traced frequently for example)

LSP / Language server support

Observability

Uprobes speedup

Heap tracing

LBR/frame-pointer/ORC walk merging

Moar tracepoints! Look at commonly used kprobes in bcc/bpftrace tools.

- VFS tracepoints

Documentation

Needs guidance for what is good/bad targets for tracing, what is a best fit for BPF versus other instrumentation frameworks (/proc, perf_events, Ftrace, etc.)

Relationship to tracing mechanisms (lttng, etw, trace logging, syslog, etc.)

Perf analysis of ebpf programs

Tooling

Standardization

Documentation under eBPF foundation / BSC of:

- current instructions
- verifier behavior
- ELF layout
- CO-RE format
- BTF
- helpers
- program types