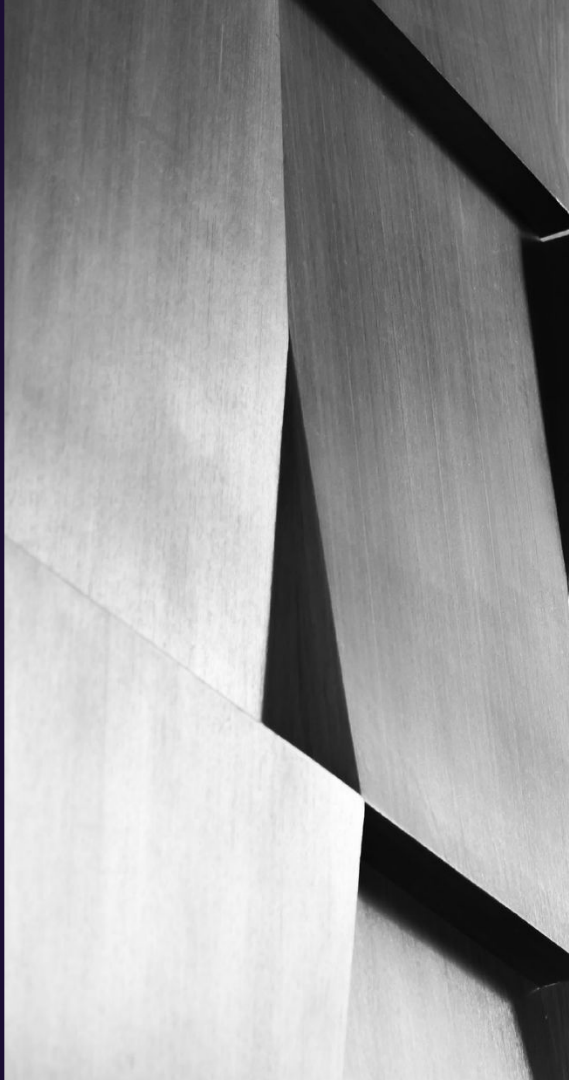


eBPF

PB173 Kernel Development Learning Pipeline

Viktor Malík
Principal Software Engineer

Red Hat

What we will cover:

- General overview

  Evolution from classic BPF, basic principles, applications

- eBPF for system observability

  Typical workflow, attach points, events

- How to write eBPF programs

  Available frameworks (libbpf, bpftrace)

- Underlying concepts

  Verifier, maps, helpers/kfuncs

- BPF Type Format (BTF)

  Compact debugging information to boost eBPF capabilities

General overview

What is eBPF?

Extended Berkeley Packet Filter

- **In-kernel virtual machine** allowing to run custom (sandboxed) programs
- No need to modify kernel source code or load modules
- Programs are written using **BPF instructions**
  - JIT-compiled to machine instructions
- Safety is assured by **BPF verifier**

What is eBPF?

Extended Berkeley Packet Filter

- **In-kernel virtual machine** allowing to run custom (sandboxed) programs
- No need to modify kernel source code or load modules
- Programs are written using **BPF instructions**
  - JIT-compiled to machine instructions
- Safety is assured by **BPF verifier**

- eBPF is one of the most exciting and developed areas of the Linux ecosystem these days
  - 5-10 new commits per day over the last 5 years
- *It's like putting JavaScript into the Linux Kernel - Brendan Gregg*

Evolution from classic BPF

A little bit of history

- **Berkeley Packet Filter (BPF)**
  - Developed in 1990s for fast packet filtering.
  - Two registers, few instructions, very limited memory (512B).
  - Now referred to as **classic BPF (cBPF)**
  - Not used anymore (implemented by eBPF)

Evolution from classic BPF

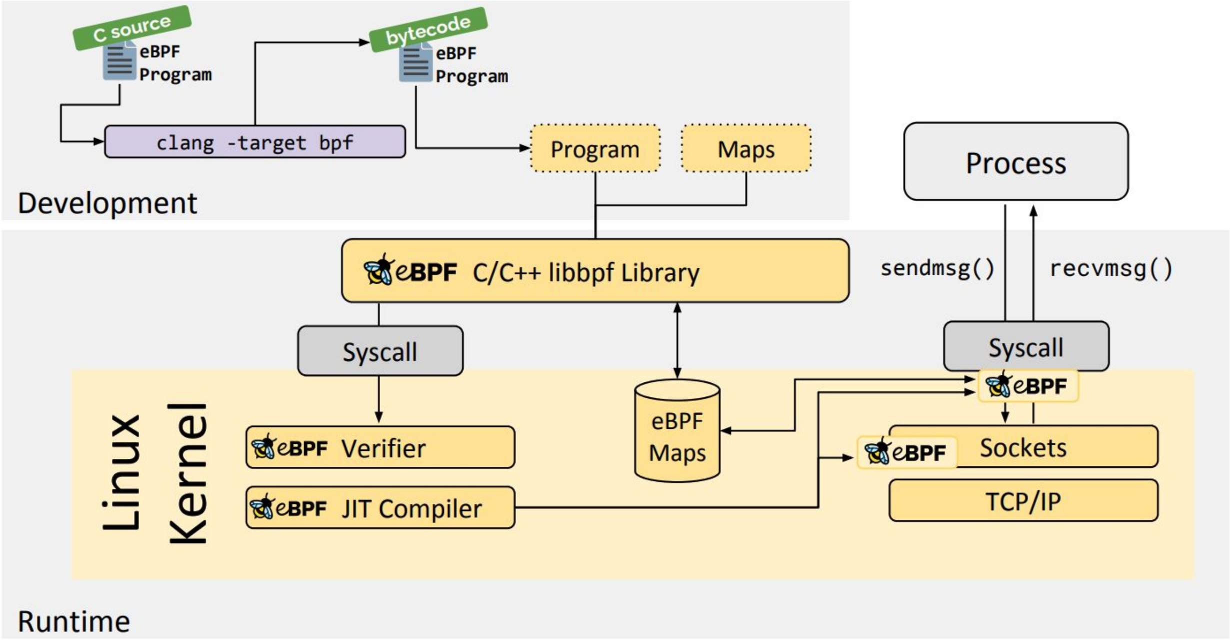
A little bit of history

- **Berkeley Packet Filter (BPF)**
  - Developed in 1990s for fast packet filtering.
  - Two registers, few instructions, very limited memory (512B).
  - Now referred to as **classic BPF (cBPF)**
  - Not used anymore (implemented by eBPF)
- **Extended BPF (eBPF)**
  - Introduced in 2014 into the Linux kernel.
  - More registers (11), instructions, more memory, verification.
  - Today **BPF = eBPF** (abbreviation no longer translated).

Applications

- **System observability (tracing)**
  - Attaching eBPF programs to various events in the kernel/userspace
  - **Collecting and processing** information about the events
- **Networking**
  - Fast packet processing (XDP – eXpress Data Path)
  - The packets can be processed before it reaches the kernel
- **Security (BPF LSM)**
  - Using BPF programs to implement Mandatory Access Control (MAC) using Linux Security Modules (LSM)
- **Scheduling (sched_ext)**
  - As of kernel v6.13, it is possible to implement custom schedulers using eBPF
- **More in development** (e.g. OOM handling)

BPF architecture



Source: https://ebpf.io/what-is-ebpf

eBPF for system observability

Typical workflow of a BPF tracing program

- Typical workflow:
  a. eBPF program is **attached to an event** in the system
  b. When the event fires, the attached BPF program **is executed**
  c. The BPF program **collects data** about the event and sends it to userspace
  d. In userspace, data is post-processed, cleaned, and **presented to the user**

Typical workflow of a BPF tracing program

- Typical workflow:
  a. eBPF program is **attached to an event** in the system
  b. When the event fires, the attached BPF program **is executed**
  c. The BPF program **collects data** about the event and sends it to userspace
  d. In userspace, data is post-processed, cleaned, and **presented to the user**

- BPF allows to do a lot of aggregation directly in the BPF program (i.e. kernel)
  - Less data needs to be sent to userspace – **significant performance gain**
  - Important advantage over other observability tools

Available events

- **Kernel**
  - Static built-in **tracepoints**
  - Dynamic function events
    - **k(ret)probes** – legacy interface, available for any instruction
    - **BPF trampolines** (fentry/fexit) – preferred interface, only available for function boundaries
  - **Memory watchpoints** – an event is generated whenever the observed memory location is accessed (can distinguish read/write/execute access).

Available events

- **Kernel**
  - Static built-in **tracepoints**
  - Dynamic function events
    - **k(ret)probes** – legacy interface, available for any instruction
    - **BPF trampolines** (fentry/fexit) – preferred interface, only available for function boundaries
  - **Memory watchpoints** – an event is generated whenever the observed memory location is accessed (can distinguish read/write/execute access).
- **Userspace**
  - Static **USDT** (user-level dynamic tracing) tracepoints
  - Dynamic **uprobes** (analogy of kprobes but for userspace)

How to write eBPF applications

Most popular frameworks

- **libbpf**
  - **canonical way** of writing BPF programs
  - both kernel and userspace parts written in C
  - many features (CO-RE, global variables) with **first-class support from kernel**
- BCC (BPF Compiler Collection)
  - userspace part written in Python, BPF part written in C, embedded as a string
  - simpler parsing and presentation of collected data (histograms, etc.)
  - uses libbpf to load programs
- **bpftrace**
  - custom **high-level language** (similar to SystemTap, DTrace)
  - great for fast prototyping
  - doesn’t require deep knowledge of eBPF

libbpf

- The default **userspace library** for interacting with the kernel
- A bit complicated to use, especially for newcomers
- Good starting point: https://github.com/libbpf/libbpf-bootstrap
- Both the BPF and the userspace parts are **written in C**
  - BPF part is compiled into a so-called *BPF skeleton* and embedded in the userspace part
- Provides a lot of macros and helpers for writing BPF code

libbpf

BPF (kernel) part

- read.bpf.c:

```
SEC("kprobe/vfs_read") ← attach points defined by special macros
int bpf_prog(struct pt_regs *ctx) ← context depends on probe type
{
    __u32 size = (__u32) PT_REGS_PARM3(ctx); ← access to the function argument

    bpf_printk("Reading %d bytes.", size); ← write to special output

    return 0;
}
```

libbpf

Userspace part

- read.c:

```
#include "read.skel.h" ← include the skeleton (BPF program)
...
int main() {
    struct read_bpf *skel = read_bpf__open();
    int err = read_bpf__load(skel);
    err = read_bpf__attach(skel);
    for (;;) {
      sleep(1);
    }
cleanup:
    read_bpf__destroy(skel);
    return -err;
}
```

bpftrace

- High-level **tracing language** for Linux using eBPF under the hood
- A single bpftrace program to create **both the BPF and the userspace** part
- More information: https://bpftrace.org/
- Allows to write powerful one-liners, great for **fast prototyping**

bpftrace

Examples

- List all opened files (system wide) by thread name:

```
# bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args.filename)); }'
Attaching 1 probe...
ptyxis-agent /proc/1435631/cmdline
Chrome_IOThread /dev/shm/.org.chromium.Chromium.ogaC8I
Chrome_IOThread /dev/shm/.org.chromium.Chromium.ogaC8I
Chrome_IOThread /dev/shm/.org.chromium.Chromium.syZXS4
mongod /var/lib/mongo/journal
[...]
```

bpftrace

Examples

- Show numbers of VFS (virtual filesystem) operations over 1 second:

```
# bpftrace -e 'kprobe:vfs_* { @[func] = count(); } interval:s:1 { exit(); }'
Attaching 2 probes...

@[vfs_statx_path]: 9
@[vfs_statx]: 124
@[vfs_open]: 197
@[vfs_fstat]: 240
@[vfs_getattr_nosec]: 249
@[vfs_write]: 315
@[vfs_read]: 1189
```

bpftrace

Examples

- Read size distribution by thread name:

```
# bpftrace -e 'tracepoint:syscalls:sys_exit_read { @[comm] = hist(args.ret); }'
Attaching 1 probe…
[...]
@[thunderbird]:
[1]                   15 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@               |
[2, 4)                 0 |                                                    |
[4, 8)                 0 |                                                    |
[8, 16)                0 |                                                    |
[16, 32)               0 |                                                    |
[32, 64)               0 |                                                    |
[64, 128)              0 |                                                    |
[128, 256)             0 |                                                    |
[256, 512)             0 |                                                    |
[512, 1K)              0 |                                                    |
[1K, 2K)               9 |@@@@@@@@@@@@@@@@@@@@@@                              |
[2K, 4K)              21 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[...]
```

bpftrace

Prewritten tools

- bpftrace also comes with a set of **prewritten tools**
- For example bashreadline.bt:

```
# cat /usr/share/bpftrace/tools/bashreadline.bt
[...]
uretprobe:/bin/bash:readline
{
        time("%H:%M:%S  ");
        printf("%-6d %s\n", pid, str(retval));
}

# /usr/share/bpftrace/tools/bashreadline.bt
Attaching 2 probes...
Tracing bash commands... Hit Ctrl-C to end.
TIME      PID     COMMAND
10:06:59  1436892 cat /etc/passwd
```

bpftool

- The reference CLI tool for **inspection** and **management** of eBPF objects
- Uses libbpf under the hood
- Provides CLI for
  - listing, dumping, loading, attaching **BPF programs**,
  - listing, dumping, creating, manipulating **BPF maps**,
  - generating **BPF skeletons**,
  - inspecting **BPF Type Information (BTF)**,
  - showing **BPF features** available on the system,
  - and much more!

bpftool

Managing eBPF programs

- List running eBPF programs

```
# bpftool prog list
[...]
44874: cgroup_device  name sd_devices  tag 89ca0032d27da4fb  gpl
       loaded_at 2025-11-10T07:47:16+0100  uid 0
       xlated 1952B  jited 1201B  memlock 4096B
       pids systemd(1)         ← modern systemd loads BPF programs
44917: tracepoint  name tracepoint_syscalls_sys_enter_openat_1  tag 562fe9f18ea5d08f  gpl
       loaded_at 2025-11-10T11:22:28+0100  uid 0
       xlated 1208B  jited 648B  memlock 4096B  map_ids 33395,33396,33391,33392,33394
       btf_id 52803
       pids bpftrace(549407)   ← our bpftrace program
```

bpftool

Managing eBPF programs

- Show instructions of a BPF program

```
# bpftool prog dump xlated id 44917
int64 tracepoint_syscalls_sys_enter_openat_1(int8 * ctx):
;
   0: (bf) r7 = r1                                ← register assignment
   1: (18) r1 = map[id:33416][0]+0                ← map creation
   3: (79) r8 = *(u64 *)(r1 +0)                   ← memory access
   4: (b7) r0 = -1837465548
   5: (bf) r0 = &(void __percpu *)(r0)
   6: (61) r0 = *(u32 *)(r0 +0)
   [...]
   17: (85) call bpf_probe_read_kernel#-123088    ← function (BPF helper) call
   18: (7b) *(u64 *)(r6 +0) = r9
   19: (7b) *(u64 *)(r10 -32) = r9
   20: (7b) *(u64 *)(r10 -40) = r9                ← stack access (r10 is frame pointer)
   [...]
```

bpftool

Managing eBPF maps

- List created eBPF maps

```
# bpftool map list
[...]
33449: percpu_hash  name AT_  flags 0x0
       key 8B  value 8B  max_entries 4096  memlock 722112B    ← key and value types
       btf_id 52913
       pids bpftrace(550454)
[...]
```

- Dump map contents

```
# bpftool map dump id 33449
[{
        "key": -1887304311,
        "values": [{
                "cpu": 0,
                "value": 0
            },{
[...]
```

Underlying concepts

eBPF Verifier

- The most important component of the system – **every program must pass** the verifier
- Properties checked:
  - **Memory access safety**
    - BPF program cannot access memory outside of its context.
    - Pointer dereferencing must use special helper functions.
  - **Stack and register access safety** – no reading uninitialized stack/register values
  - **Instruction reachability** – no dead or unreachable instructions
  - **Termination**
    - eBPF programs must terminate
    - At most 1 million instructions are allowed
  - ...and several others

eBPF maps

- Generic key-value storage accessible from **both kernel and userspace**
- Typically used for multiple purposes:
  - **Passing collected data** from BPF programs to userspace
  - **Sharing state** between (instances of) BPF programs

    Example with bpftrace:

```
kprobe:vfs_* { @timestamp[tid] = nsecs }
kretprobe:vfs_* { printf("%s ran for %d ns\n", func, nsecs - @timestamp[tid]); }
```

- Various map kinds available: (per-cpu) hash maps and arrays, queues, stacks, …

eBPF helpers and kernel functions

- Due to safety reasons, eBPF programs **cannot call arbitrary kernel functions**
- There are lists of functions which are safe (safety is ensured by the verifier) to run:
  - accessing information about the **running process** – name, PID, TID, curtask, stacktrace, …,
  - map manipulation (e.g. inserting, finding, and deleting elements),
  - **accessing memory**,
  - **iteration**,
  - … and many many more.
- These functions are of 2 kinds:
  - BPF **helpers** – stable, legacy, not added anymore.
  - BPF kernel functions (**kfuncs**) – “unstable”, preferred, new are added all the time.

Looping/iteration

General description

- Problem: eBPF programs **must terminate**, otherwise the system would hang.
- In the first version, the verifier prohibited loops completely.
- Then, **bounded loops** were added.
- Still, some **problems remained**:
  - It is still hard for the verifier to prove that a loop is bounded.
  - It is desirable to iterate over collections which are finite but the number of elements is not known beforehand.

Looping/iteration

Current state

- These days, there are many options for using loops in BPF programs:
  - **`bpf_loop` helper** allows to execute a function a number of times
    - Simple verification (if the function terminates, the loop terminates)
  - **eBPF iterators** are special program types that allow executing code for entry from some collection of kernel objects (running tasks, virtual memory areas, TCP sockets, ...)

```
SEC("iter/task")
int iter_task(struct bpf_iter__task_file *ctx) {
        struct task_struct *task = ctx->task;
        // Do something with the task pointer
}
```

  - **Open-coded iterators** allow to iterate these collections from within other BPF programs

```
bpf_for_each(task, task_ptr, NULL, BPF_TASK_ITER_ALL_PROCS) {
        // Do something with the task pointer
}
```

Accessing memory

- eBPF programs **cannot access arbitrary memory**
  - Possible **out-of-bounds access** (can be checked by the verifier)
  - Possible **page faults**
- For accessing potentially unsafe memory, special helpers must be used:

```
SEC("kprobe/vfs_read")
int bpf_prog(struct pt_regs *ctx)
{
    struct task_struct *current = (struct task_struct *)bpf_get_current_task();
    struct task_struct *parent;
    bpf_probe_read_kernel(&parent, sizeof(parent), &current->real_parent);
}
```

BPF Type Format (BTF)

BPF Type Format (BTF)

- Problem: a lot of information useful for tracing is in the debugging information
  - Names of variables, parameters, struct fields, etc.
  - But DWARF is too large (kernel-debuginfo has 4.4 GB on Fedora)
- BTF is a **compact format for kernel debugging information**
  - Contains definitions of all kernel functions
  - Generated from DWARF by deduplication (4.5 MB BTF vs 195 MB DWARF)
- Thanks to the small size, BTF is **embedded** in most kernels by default
  - See for yourself (`/sys/kernel/btf/vmlinux`)

Compile-Once, Run-Everywhere (BTF)

Features enabled by BTF

- Problem: **layout of kernel structures can change** between versions (no stable ABI)
- Normally, eBPF programs would have to be recompiled for each kernel version
- CO-RE uses BTF to **dynamically adjust** the BPF program to the current kernel upon loading
- Example using libbpf:

```
SEC("kprobe/vfs_read")
int bpf_prog(struct pt_regs *ctx)
{
    struct task_struct *current = (struct task_struct *)bpf_get_current_task();
    int ppid = BPF_CORE_READ(current, real_parent, tgid);
}
```

BPF trampolines (fentry/fexit)

Features enabled by BTF

- New probe type for attaching BPF programs to function entries/exits.
- Advantages:
  - Practically no overhead (use special nop instructions)
  - Have access to **function arguments by name** (thanks to BTF)
  - **Direct dereferencing** is possible
- Example using libbpf:

```
SEC("fentry/vfs_open")
int BPF_PROG(vfs_open, const struct path *path, struct file *file)
{
    ... path->dentry->d_name.name ...
}
```

vmlinux.h

Features enabled by BTF

- When using kernel types, BPF programs need to **include kernel headers**

```
#include <linux/sched.h>
SEC("kprobe/vfs_read")
int bpf_prog(struct pt_regs *ctx)
{
    struct task_struct *current = (struct task_struct *)bpf_get_current_task();
    ... current->tid ...
}
```

vmlinux.h

Features enabled by BTF

- When using kernel types, BPF programs need to **include kernel headers**

```
#include <linux/sched.h>
SEC("kprobe/vfs_read")
int bpf_prog(struct pt_regs *ctx)
{
    struct task_struct *current = (struct task_struct *)bpf_get_current_task();
    ... current->tid ...
}
```

- BTF has all the types so we can use it to generate **vmlinux.h - the header with all kernel types**

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

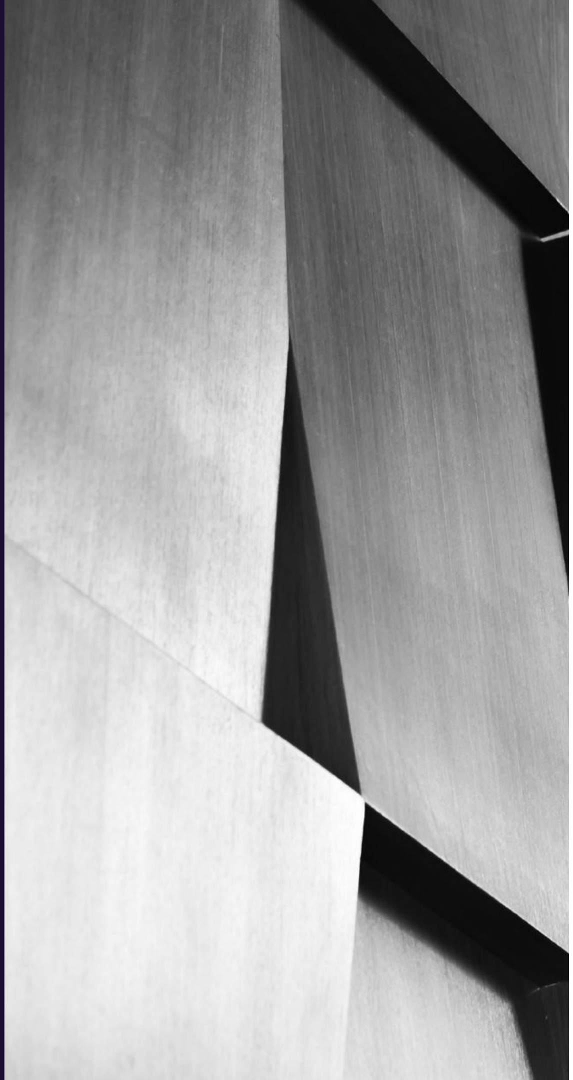
- Then, all BPF programs just need to include vmlinux.h, nothing else

```
#include "vmlinux.h"
SEC("kprobe/vfs_read")
...
```

Conclusion

Conclusion

- **eBPF is an exciting technology** which is getting a lot of traction these days.
- One (but not the only one!) of the use-cases is for **system observability**.
- eBPF completely redefined the way Linux kernel can be extended at runtime.
- More resources:
  - eBPF website: https://ebpf.io/
  - eBPF docs: https://docs.ebpf.io/
  - eBPF documentary: https://ebpfdocumentary.com/



Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

facebook.com/redhatinc

youtube.com/user/RedHatVideos

twitter.com/RedHat