CS 60038: Advances in Operating Systems Design

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Extended Berkeley Packet Filter (eBPF)

Introduction

Revolutionizes Production Process across Industries

Google

Google uses eBPF for security auditing, packet processing, and performance monitoring.

NETFLIX

Netflix uses eBPF at scale for network insights.

android

Android uses eBPF to monitor network usage, power, and memory profiling.

S&P Global

S&P Global uses eBPF through Cilium for networking across multiple clouds and on-prem.

shopify

Shopify uses eBPF through Falco for intrusion detection.

CLOUDFLARE

Cloudflare uses eBPF for network security, performance monitoring, and network observability.

- GKE Dataplane V2, an opinionated dataplane, to make Linux kernel Kubernetes-aware
- Android to monitor network usage, power, and memory profiling.
- Cloudflare for Enhanced Network Security, Performance Monitoring, and Observability.
- Netflix Unveils Flow Exporter
- Apple for kernel security monitoring.
- Facebook for Enforcing encryption at scale, for Traffic Optimization Systems for a Fast and Reliable User Access.

LKM vs eBPF

- To overcome the limitation of user space code using system calls to access the kernel, Linux provides **kernel modules (LKM)** that can be loaded into the kernel at run time.
- To provide sandboxing of these LKMs, Linux provided **BPF (Berkeley Packet Filter),** originally in BSD, as a way of running in kernel programs.
  - Examples of these include utilities such as tcpdump.
- Originally, **Berkeley Packet Filter (BPF)** was designed for capturing and filtering **network packets**.
  - However, its scope has vastly expanded, and it now encompasses a wide range of use cases beyond networking.
- eBPF provides a **virtual machine-like** environment within the Linux kernel, allowing users to write, compile, and run programs that can interact with various kernel subsystems.

eBPF Background

What ?

eBPF allows developers to safely inject code into a kernel sandbox, functioning as a privileged virtual machine within the kernel space.

Why ?

By leveraging eBPF, developers can enhance the functionalities of the OS without the need for modifying kernel code or loading a kernel module.

Also optimizes current kernel capabilities required for specific business use case.

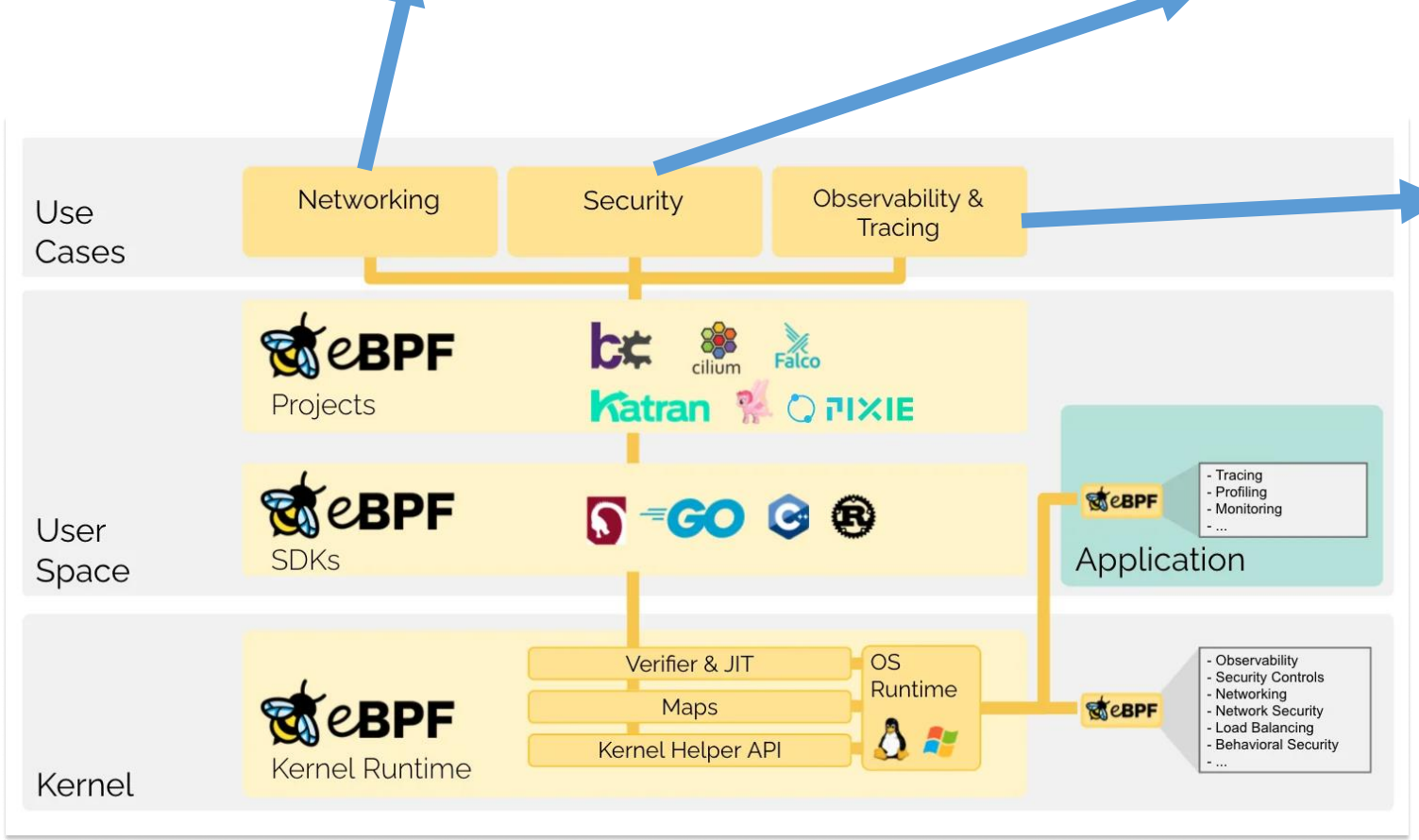
How ?

The programs are event-driven, executing when the kernel or an application encounters a hook point. Numerous pre-defined hook points are available, including system calls, function entry/exit, kernel tracepoints, and network events.

Usage of eBPF

To implement custom network functions such as load balancing, packet filtering, and routing, all within the kernel, providing high performance and flexibility.

Tp enforce security policies by monitoring system calls, network activity, and other kernel events, detecting and preventing unauthorized actions.



Enables system & application monitoring by attaching probes to various kernel subsystems, allowing developers and administrators to gain deep insights into the performance and behaviour of their systems.

Help identify performance bottlenecks and optimize system behaviour by modifying kernel data structures and functions on-the-fly.

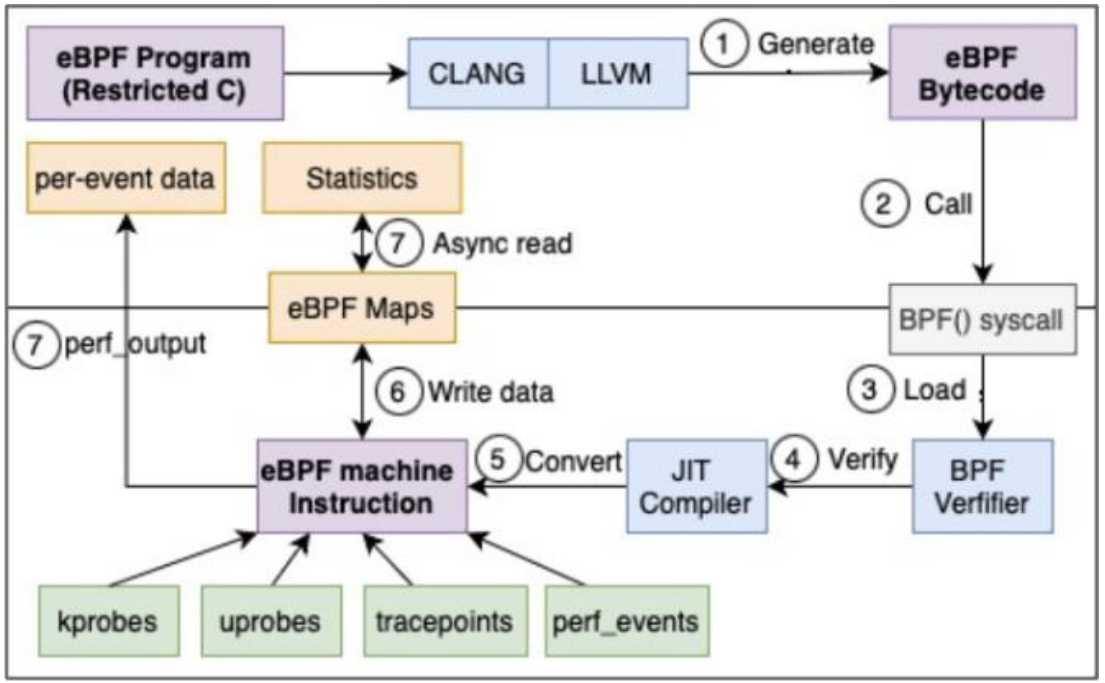
How eBPF Works

1. Writing eBPF Programs:

- Users write eBPF programs using a restricted C-like language.
- These programs are event-driven and are triggered by specific events such as system calls, network packets, or tracepoints.
- eBPF programs can be written using the LLVM (Low-Level Virtual Machine) compiler collection with the BPF backend, which supports the eBPF instruction set.

2. Compiling and Loading:

- Once written, the eBPF program is compiled into bytecode, which can be loaded into the kernel using the bpf() system call.
- This bytecode is platform-independent and can be executed on any Linux system with eBPF support.



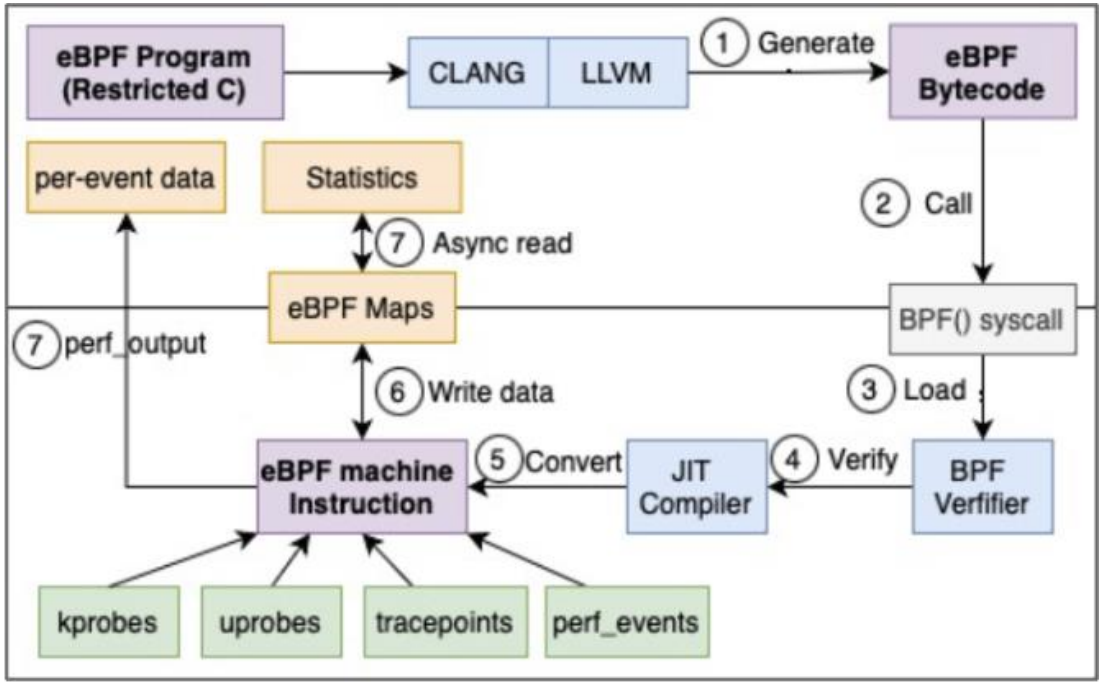
How eBPF Works

3. Verification:

- To ensure safety, the kernel verifies the loaded bytecode using a verifier component.
- The verifier checks for several conditions, such as illegal memory access, infinite loops, and proper resource usage, preventing potentially harmful or poorly written programs from running.

4. Just-In-Time (JIT) Compilation:

- After verification, the bytecode is translated into native machine code for the target platform using a Just-In-Time (JIT) compiler.
- This step significantly improves performance by avoiding interpretation overhead during execution.



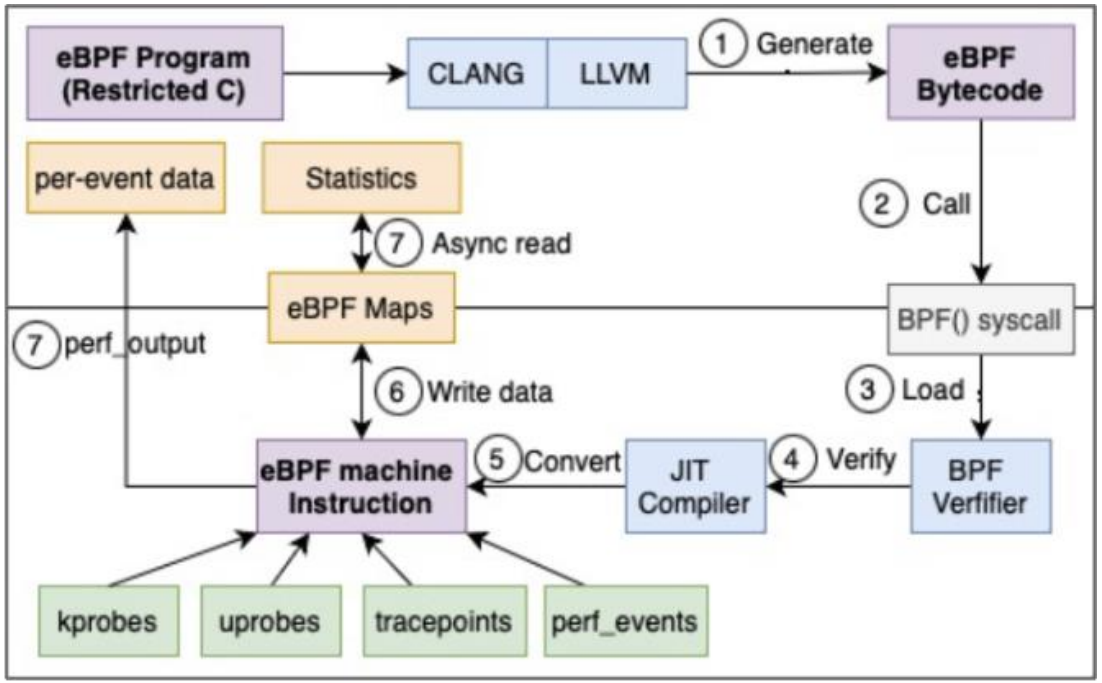
How eBPF Works

5. Execution:

- The eBPF program is executed by the in-kernel eBPF virtual machine in response to specific events.
- The eBPF VM ensures isolation and safety while providing kernel data structures and functions access.

6. Interaction with User-space:

- eBPF programs can communicate with user-space applications using eBPF maps, key-value data structures shared between the kernel and User-space.
- This allows for efficient and flexible data exchange between the two domains.



eBPF Components

- Programs are written in restricted C. eBPF backend for LLVM/Clang
  - `sudo clang -g -O2 -emit-llvm -c testProg.bpf.c -o - | llc -march=bpf -filetype=obj -o testProg.o`
- eBPF Verifier
  - Verified to finish (no loops), no unreachable instructions, reads to uninitialized registers, or memory access to arbitrary pointers restricted kernel function calls and data structure access
- eBPF Maps / Perf Event Ring Buffer
  - Memory Mapped, bi-directional data structures for storage.
  - Allow sharing of data between eBPF Kernel programs and also between kernel and use-space program
- Helper Functions
  - Kernel functions exposed to eBPF Programs

eBPF Program Types

- An eBPF program can be attached at multiple different places within the kernel
  - For performance engineering, the hooks are
    - Kprobes/Uprobes
    - Tracepoints
    - USDT
    - PerfEvents

```
enum bpf_prog_type {
        BPF_PROG_TYPE_UNSPEC,
        BPF_PROG_TYPE_SOCKET_FILTER,
        BPF_PROG_TYPE_KPROBE,
        BPF_PROG_TYPE_SCHED_CLS,
        BPF_PROG_TYPE_SCHED_ACT,
        BPF_PROG_TYPE_TRACEPOINT,
        BPF_PROG_TYPE_XDP,
        BPF_PROG_TYPE_PERF_EVENT,
        BPF_PROG_TYPE_CGROUP_SKB,
        BPF_PROG_TYPE_CGROUP_SOCK,
        BPF_PROG_TYPE_LWT_IN,
        BPF_PROG_TYPE_LWT_OUT,
        BPF_PROG_TYPE_LWT_XMIT,
        BPF_PROG_TYPE_SOCK_OPS,
        BPF_PROG_TYPE_SK_SKB,
        BPF_PROG_TYPE_CGROUP_DEVICE,
        BPF_PROG_TYPE_SK_MSG,
        BPF_PROG_TYPE_RAW_TRACEPOINT,
        BPF_PROG_TYPE_CGROUP_SOCK_ADDR,
        BPF_PROG_TYPE_LWT_SEG6LOCAL,
        BPF_PROG_TYPE_LIRC_MODE2,
        BPF_PROG_TYPE_SK_REUSEPORT,
        BPF_PROG_TYPE_FLOW_DISSECTOR,
        BPF_PROG_TYPE_CGROUP_SYSCTL,
        BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE,
        BPF_PROG_TYPE_CGROUP_SOCKOPT,
        BPF_PROG_TYPE_TRACING,
        BPF_PROG_TYPE_STRUCT_OPS,
        BPF_PROG_TYPE_EXT,
        BPF_PROG_TYPE_LSM,
        BPF_PROG_TYPE_SK_LOOKUP,
        BPF_PROG_TYPE_SYSCALL, /* a program that can execute syscalls */
        BPF_PROG_TYPE_NETFILTER,
};
```

https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h

eBPF Maps

- https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#maps

```
enum bpf_map_type {
        BPF_MAP_TYPE_UNSPEC,
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        BPF_MAP_TYPE_PERCPU_HASH,
        BPF_MAP_TYPE_PERCPU_ARRAY,
        BPF_MAP_TYPE_STACK_TRACE,
        BPF_MAP_TYPE_CGROUP_ARRAY,
        BPF_MAP_TYPE_LRU_HASH,
        BPF_MAP_TYPE_LRU_PERCPU_HASH,
        BPF_MAP_TYPE_LPM_TRIE,
        BPF_MAP_TYPE_ARRAY_OF_MAPS,
        BPF_MAP_TYPE_HASH_OF_MAPS,
        BPF_MAP_TYPE_DEVMAP,
        BPF_MAP_TYPE_SOCKMAP,
        BPF_MAP_TYPE_CPUMAP,
        BPF_MAP_TYPE_XSKMAP,
        BPF_MAP_TYPE_SOCKHASH,
        BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED,
        /* BPF_MAP_TYPE_CGROUP_STORAGE is available to bpf programs attaching
         * to a cgroup. The newer BPF_MAP_TYPE_CGRP_STORAGE is available to
         * both cgroup-attached and other progs and supports all functionality
         * provided by BPF_MAP_TYPE_CGROUP_STORAGE. So mark
         * BPF_MAP_TYPE_CGROUP_STORAGE deprecated.
         */
        BPF_MAP_TYPE_CGROUP_STORAGE = BPF_MAP_TYPE_CGROUP_STORAGE_DEPRECATED,
        BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
        BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,
        BPF_MAP_TYPE_QUEUE,
        BPF_MAP_TYPE_STACK,
        BPF_MAP_TYPE_SK_STORAGE,
        BPF_MAP_TYPE_DEVMAP_HASH,
        BPF_MAP_TYPE_STRUCT_OPS,
        BPF_MAP_TYPE_RINGBUF,
        BPF_MAP_TYPE_INODE_STORAGE,
        BPF_MAP_TYPE_TASK_STORAGE,
        BPF_MAP_TYPE_BLOOM_FILTER,
        BPF_MAP_TYPE_USER_RINGBUF,
        BPF_MAP_TYPE_CGRP_STORAGE,
};
```

https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h

eBPF Maps

- **BPF_MAP_CREATE**
  - Create a map with the desired type and attributes in attr:
- **BPF_MAP_LOOKUP_ELEM**
  - Lookup key in a given map using
  - `attr->map_fd, attr->key, attr->value`.
  - Returns zero and stores found element into `attr->value` on success, or negative error on failure.
- **BPF_MAP_UPDATE_ELEM**
- **BPF_MAP_DELETE_ELEM**

```
int fd;
union bpf_attr attr = {
    .map_type = BPF_MAP_TYPE_ARRAY; /* mandatory */
    .key_size = sizeof(__u32); /* mandatory */
    .value_size = sizeof(__u32); /* mandatory */
    .max_entries = 256; /* mandatory */
    .map_flags = BPF_F_MMAPABLE;
    .map_name = "example_array"; };

fd = bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
```

```
int *val= bpf_map_lookup_elem(& example_array,
&value);
```

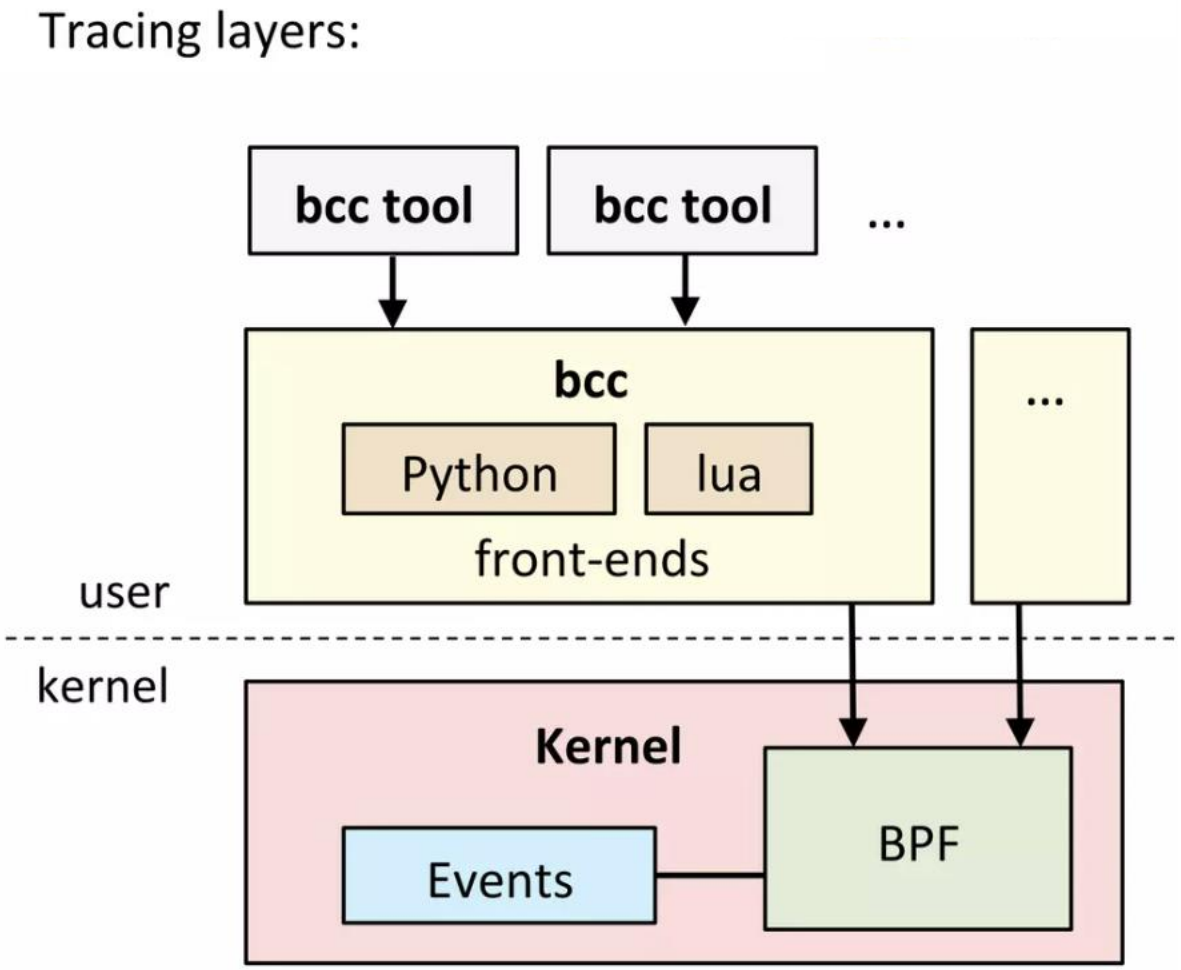
https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h

eBPF Development Tools

- eBPF uses the **BPF Compiler Collection (BCC),** a toolkit to create efficient and simple kernel tracing programs.
- A range of useful examples is provided with the BCC, which requires Linux 4.1 or above. BCC provides a C wrapper around LLVM and provides Python and Lua based front ends.

bcc tools

- BPF Compiler Collection
  - https://github.com/iovisor/bcc
  - Lead Developer: Brenden Blanco
- It includes tracing tools
- Provide BPF Front-ends:
  - Python
  - Lua
  - C++
  - C helper libraries
  - Golang (gobpf)



bcc tool Installation

- Go to: https://github.com/iovisor/bcc/blob/master/INSTALL.md
- Kernel Configuration:
  - In general, to use these features, a Linux kernel version 4.1 or newer is required.
  - In addition, the kernel should have been compiled with the following flags set
  - **cat /boot/config-’uname -r’ | grep 'BPF'**

```
root@utkalika-ibm4:/usr/share/bcc/tools# cat /boot/config-6.2.0-33-generic | grep 'BPF'
CONFIG_BPF=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_ARCH_WANT_DEFAULT_BPF_JIT=y
# BPF subsystem
CONFIG_BPF_SYSCALL=y
CONFIG_BPF_JIT=y
CONFIG_BPF_JIT_ALWAYS_ON=y
CONFIG_BPF_JIT_DEFAULT_ON=y
CONFIG_BPF_UNPRIV_DEFAULT_OFF=y
# CONFIG_BPF_PRELOAD is not set
CONFIG_BPF_LSM=y
# end of BPF subsystem
CONFIG_CGROUP_BPF=y
CONFIG_IPV6_SEG6_BPF=y
CONFIG_NETFILTER_XT_MATCH_BPF=m
CONFIG_BPFILTER=y
CONFIG_BPFILTER_UMH=m
CONFIG_NET_CLS_BPF=m
CONFIG_NET_ACT_BPF=m
CONFIG_BPF_STREAM_PARSER=y
CONFIG_LWTUNNEL_BPF=y
CONFIG_BPF_EVENTS=y
CONFIG_BPF_KPROBE_OVERRIDE=y
CONFIG_TEST_BPF=m
```

```
cat /boot/config-6.2.0-33-generic | grep 'BPF'
```

- Install bcc from package or from the source
- Tools will be installed under /usr/share/bcc/tools.

bcc/Python (C & Python)

```
#!/usr/bin/python
#
# This is a Hello World example that formats output as
fields.

from bcc import BPF
from bcc.utils import printb

# define BPF program
prog = """
int hello(void *ctx) {
bpf_trace_printk("Hello, World!\\n");
return 0;
}
"""
```

```
# load BPF program
b = BPF(text=prog)
b.attach_kprobe(event=b.get_syscall_fnname("clone"),
fn_name="hello")

# header
print("%-18s %-16s %-6s %s" % ("TIME(s)", "COMM", "PID",
"MESSAGE"))

# format output
while 1:
try:
(task, pid, cpu, flags, ts, msg) = b.trace_fields()
except ValueError:
continue
except KeyboardInterrupt:
exit()
printb(b"%-18.9f %-16s %-6d %s" % (ts, task, pid, msg))
```

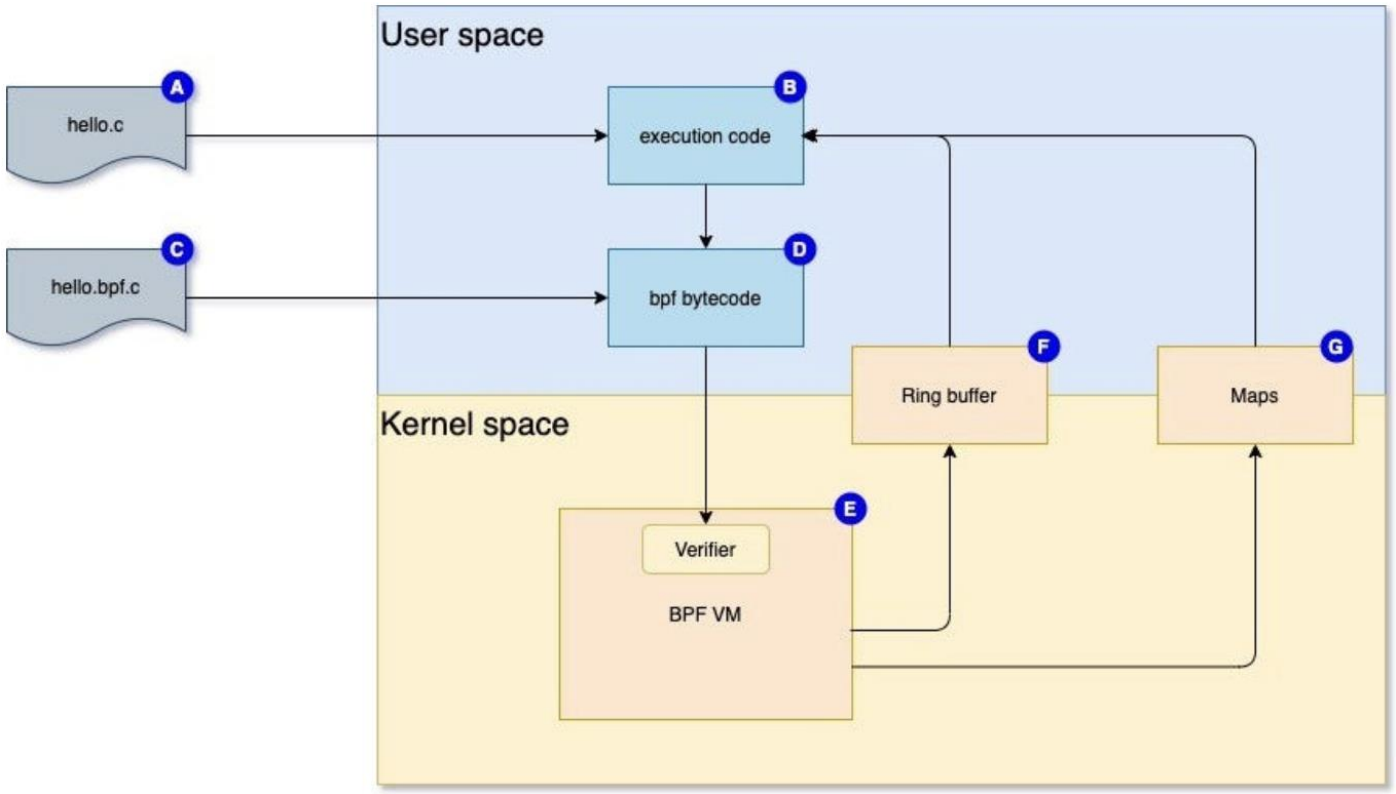

Full Code: https://github.com/iovisor/bcc/tree/master/examples/tracing/hello_fields.py

eBPF Programs - BPF CO-RE vs libbpf-bootstrap

- Writing bpf program in C
  - These small programs are portable and also know as BPF CO-RE.
  - *Compile Once Run Everywhere*
- Libbpf is a user-space library / API for loading and interacting with bpf programs.
- Libbpf handles the execution and is delivered a header files, which contain C function declarations and more.

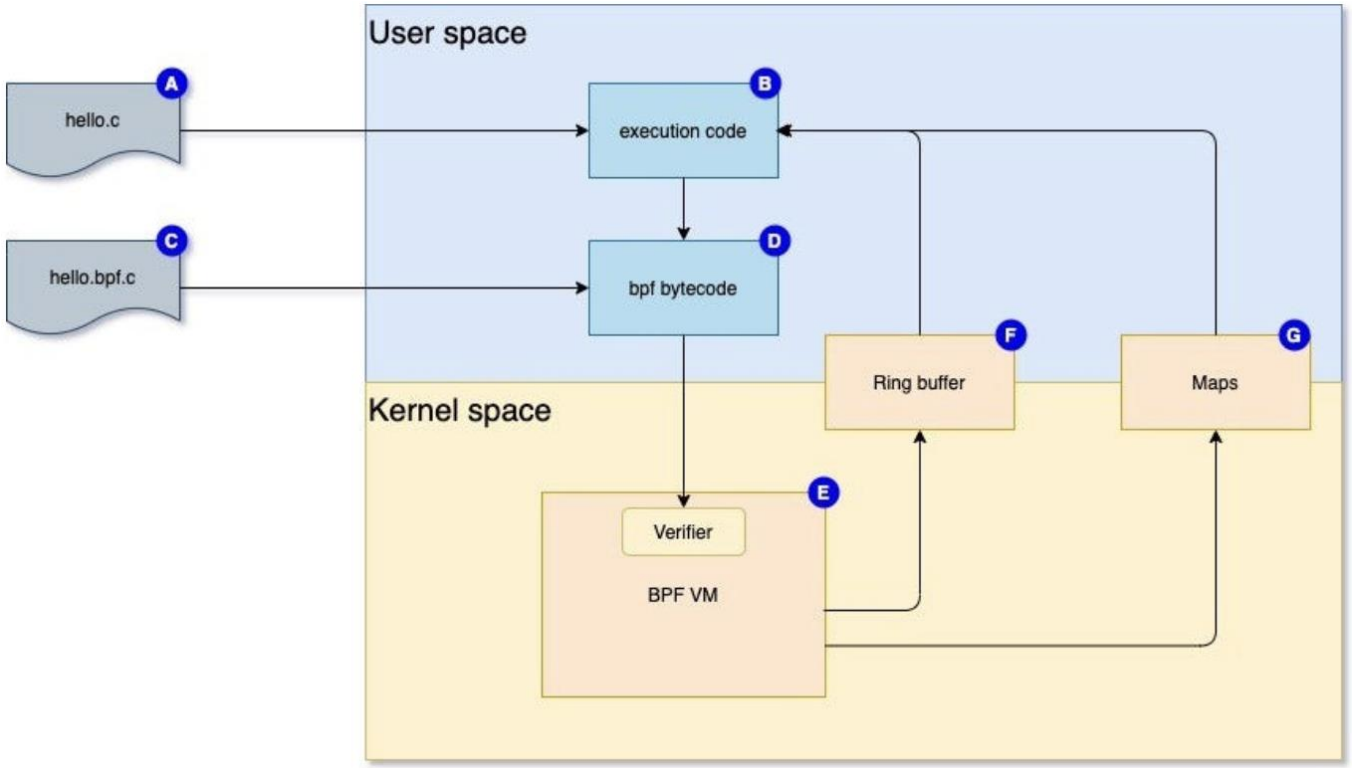
eBPF Programs - BPF CORE vs libbpf-bootstrap

- Most minimal programs require at least two pieces of code (A & C), which are actually the code for execution (B) and the actual bpf program to execute as bytecode (D) in the BPF VM (E), which is actually an in-kernel VM.
- Before execution there is always a validation done by the verifier.
- When the actual bpf program is executed it will use the ring buffer (F) to prepare and send data to the execution code.



eBPF Programs - BPF CORE vs libbpf-bootstrap

- Maps (G) are the actual storage containers that deliver abstracted data containers
  - Support multiple data types and share data with the user space executed program.
- The ring buffer uses the maps to store the prepared data.



Installation – Pre-requisites

- Linux Operating system with the latest kernel (*uname -r*).
- Ensure that the Linux kernel headers are installed and available.
- Ensure that you have the latest version of Clang/LLVM installed. Clang will be used by our Makefile to compile the actual C program.
- For compiling everything you must have the libbpf sources available at libbpf-bootstrap/libbpf. Or just customize the Makefile for this.

eBPF CO-RE Application – maps.bpf.c

```
#include "vmlinux.h"
#include <bpf_helpers.h>
#include "maps.h"

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 128);
    __type(key, pid_t);
    __type(value, struct event);
} execs SEC(".maps");

SEC("tracepoint/syscalls/sys_enter_execve")
int tracepoint__syscalls__sys_enter_execve(struct
trace_event_raw_sys_enter *ctx){
    struct event *event;
    pid_t pid;
    u64 id;
    uid_t uid = (u32) bpf_get_current_uid_gid();
    id = bpf_get_current_pid_tgid();
    pid = (pid_t)id;
```

```
    if (bpf_map_update_elem(&execs, &pid,
&((struct event){}), 1)) {
        return 0;
    }
    event = bpf_map_lookup_elem(&execs, &pid);
    if (!event) {
        return 0;
    }
    event->pid = pid;
    event->uid = uid;
    bpf_get_current_comm(&event->comm,
                         sizeof(event->comm));
    return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

Check /sys/kernel/debug/tracing/events

eBPF CO-RE Application – maps.c Snippet (loading)

```
. . .
int main(int argc, char **argv) {
        struct maps_bpf *obj;
        int err = 0;
        struct rlimit rlim = {
                .rlim_cur = 512UL << 20,
                .rlim_max = 512UL << 20,
        };
        const struct argp argp = {
                .options = opts,
                .parser = parse_arg,
                .doc = argp_program_doc,
        };
        int fd;

        err = setrlimit(RLIMIT_MEMLOCK, &rlim);
        if (err) {
                fprintf(stderr, "failed to change rlimit\n");
                return 1;
        }
```

eBPF CO-RE Application – maps.c Snippet (loading)

```
. . .
    obj = maps_bpf__open();
    if (!obj) {
          fprintf(stderr, "failed to open and/or load BPF object\n");
         return 1;
    }
    err = maps_bpf__load(obj);
    if (err) {
         fprintf(stderr, "failed to load BPF object %d\n", err);
         goto cleanup;
    }
    err = maps_bpf__attach(obj);
    if (err) {
         fprintf(stderr, "failed to attach BPF programs\n");
         goto cleanup;
    }
    fd = bpf_map__fd(obj->maps.execs);
    printf("printing executed commands\n");
    while (1) {
        print_execs(fd);
        fd = bpf_map__fd(obj->maps.execs);
    }
. . .
```

Refer for Full Code https://github.com/sartura/ebpf-hello-world/

Compilation

- We will need a static version of libbpf:

```
$ git clone https://github.com/libbpf/libbpf && cd libbpf/src/
$ make BUILD_STATIC_ONLY=1 OBJDIR=../build/libbpf DESTDIR=../build INCLUDEDIR=
 LIBDIR= UAPIDIR= install
```

- To build steps is provided here:

```
$ bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
$ clang -g -O2 -target bpf -D__TARGET_ARCH_x86_64 -I . -c maps.bpf.c -o maps.bpf.o
$ bpftool gen skeleton maps.bpf.o > maps.skel.h
$ clang -g -O2 -Wall -I . -c maps.c -o maps.o
$ clang -Wall -O2 -g maps.o libbpf/build/libbpf.a -lelf -lz -o hello
$ sudo ./maps
```

Further Readings

- https://arnold-van-wijnbergen.medium.com/introduction-b914e6e976f3 - libbpf code
- https://arthurchiao.art/blog/firewalling-with-bpf-xdp/
- https://www.slideshare.net/brendangregg/velocity-2017-performance-analysis-superpowers-with-linux-ebpf
- https://www.slideshare.net/RayJenkins1/understanding-ebpf-in-a-hurry-149197981
- https://www.vamsitalkstech.com/cloud/an-introduction-to-ebpf-architecture-part-2/
- https://github.com/gojue/ebpf-slide
- https://unzip.dev/0x00c-ebpf/
- https://www.sartura.hr/blog/simple-ebpf-core-application/

Happy Learning