eBPF - From a Programmer's Perspective

Niclas Hedam
IT University of Copenhagen
Denmark
nhed@itu.dk

ABSTRACT

eBPF allows software developers to write programs that are executed in the kernel without requiring recompilation and system restart. These programs can collect critical performance metrics when a kernel function is invoked. In this paper, we will describe and discuss the architecture of eBPF using libbpf as well as the core components of it. We will look at key differences between eBPF programs and typical user-space C programs. Lastly, we will look into some real-world use-cases of eBPF. We will, however, not discuss performance numbers or formal proofs. This paper is merely a summary of countless hours of reading through eBPF textbooks, blog posts, eBPF samples and kernel code.

1 ACKNOWLEDGEMENTS

I would like to extend thanks to Quentin Monnet, who have contributed tremendously in verifying correctness of the paper as well providing valuable feedback.

2 INTRODUCTION

Berkeley Packet Filter or *BPF* emerged as an efficient network packet filter in 1992 [4, 13]. A network packet filter is a network security mechanism for controlling what flows from and to a network by inspecting packets as they pass through the filter. BPF was described by the authors as 20 times faster than the state of the art. BPF differed from previous systems by running programs in a virtual machine built for register-based CPUs and having per-application buffers that did not require copying all information to make a decision [4]. BPF became state of the art and was adopted as the technology of choice for network packet filtering.

Alexei Starovoitov introduced eBPF in 2014 [17, 4] as a redesign of BPF for modern hardware. The eBPF VM is faster as it resembles the contemporary processors more and thus allows eBPF instructions to be mapped closely to the hardware *instruction set architecture* (ISA) [8]. In Alexei's commit from 2014 [17], eBPF was exposed to user-space and soon after, eBPF stopped being limited to the networking stack and over time, it became much more broad and generic. eBPF makes it possible to update the behaviour of the kernel without the need to recompile it and to reboot the system, while offering a simpler and safer interface than module programming.

eBPF is built with a static verifier that ensures that a program cannot cause a kernel crash and that it will always terminate. After the program is compiled, the eBPF verifier checks that the program is safe to run [4, 5].

Before the kernel can run eBPF programs, it must know where to attach it. The execution point is defined by the eBPF program types, which will be described later in this paper. The eBPF architecture also contains maps, which are bidirectional data structures allowing eBPF programs to asynchronously share data with user-space [4].

In this paper we will look at eBPF mainly from the perspective of libbpf. While there exist other approaches, libbpf is the recommended way to go with C programs. We will first look into the high-level architecture of eBPF and libbpf and then shift towards some practical examples and practical use cases.

3 ARCHITECTURE

Listing 1 is an example of an eBPF program that attaches to the *kill* system call. It can be used for security and auditing purposes by logging and documenting when processes are not gracefully terminated. Since the eBPF program is running in the kernel, there is no way of preventing this logging without having escalated privileges on the system.

The *SEC* macro is used to tell the compiler to place the bytecode in a specified ELF section. The section name is later picked up by the loader, which then deduces the attach type. The section names are not a eBPF

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct syscalls_enter_kill_args {
    long long pad;

    long syscall_nr;
    long pid;
    long sig;
};

SEC("tracepoint/syscalls/sys_enter_kill")
int kill_example(struct
    syscalls_enter_kill_args *ctx) {

  if(ctx->sig != 9) return 0;

  char fmt[] = "PID %u is being killed!\n";
  bpf_trace_printk(fmt, sizeof(fmt), ctx->
    pid, sizeof(ctx->pid));

  return 0;
}

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

Listing 1: A kill eBPF example written in C. This example is of a tracepoint eBPF program (see section 4.2.1).

convention, but a convention of the program loading the eBPF program. In libbpf, other than the reserved keywords such as *maps*, the probes are first defined by the type and then the hook. In listing 1 for example, the program type is *tracepoint* and the hook is `syscalls/sys_enter_kill`.

The SEC macro is defined in the `bpf/bpf_helpers.h` file in libbpf. When compiling the file, the SEC macro is replaced by an *__attribute__* statement, which is a mechanism in GNU C to attach characteristics to function declarations [9].

We use `bpf_trace_printk`, which is defined in the kernel, to print out trace information to the common trace pipe [1]. This function offer *printf*-like functionality, but in kernel-space. We will describe probe types in more details in section 4.

All eBPF programs takes a context as parameter. For tracing programs, the context contains information about the information that the kernel is currently processing including registers or function parameters [4]. The context depends on the type of eBPF program as well as the location of the probe. In the listing before, the context is a `syscalls_enter_kill_args` struct, which follows the format published by the kernel[2]. The first 8 bytes are unused and should be ignored. We will describe the context parameter in more details in section 4.

In the bottom of listing 1, the license of the eBPF program is declared. Since the kernel is licensed under GPL, some eBPF programs are required to be GPL-compatible. Other programs, like networking programs, do not have to be GPL-compatible and may be under proprietary licenses. Whether or not an eBPF program must be GPL-compatible depends on, among other things, the program type and the used helpers.

3.1 Scope

eBPF programs cannot call arbitrary kernel functions [5]. This is a design choice, as it would bind the eBPF program to specific kernel versions and thus complicate compatibility. eBPF programs can, however, invoke a set of helper functions offered by the kernel.

Examples of eBPF helper functions include

- Random number generation.
- Access to current time.
- Access to eBPF maps.
- Get process/cgroup context.
- Alter network packets.

eBPF programs can in principle invoke external library functions if they adhere to the requirements of the verifier, which are described in section 6, and if the compiler is able to inline them in the code.

3.2 Compilation

The rest of this paper assumes that you have a working eBPF environment to run the examples. If you have not compiled or run an eBPF program on your system before, read appendix A.

eBPF is a low-level language, an ISA, that can be compiled from high-level languages such as Rust and C [4, 7]. In this paper, we will focus on C and compilation using *clang* and *llc*. The choice of clang over *GCC* is rooted in the maturity of eBPF in the two compilers. Clang have had a longer history with eBPF and it is

[1] `/sys/kernel/debug/tracing/trace_pipe`

[2] Published in `/sys/kernel/debug/tracing/events/syscalls/sys_enter_kill/format`

therefore regarded as the tool of reference in the eBPF community.

When using the clang/LLVM toolchain, one can compile in one or two steps. Compiling first to LLVM *Intermediate Representation* (LLVM IR) and using llc in a second step offers a finer control on the options passed to llc. Compiling listing 1 in two steps can be done by first calling the following command.

```
$ clang -target bpf -S -D __BPF_TRACING__
    -I./libbpf/src/root/usr/include/ -Wall
    -Werror -O2 -emit-llvm -c -g kill.c
```

We choose to compile with optimisation level 2 as this is a necessary level for most eBPF programs. Without it, suboptimal code with heavy stacks may be generated and some invoked functions may be referenced incorrectly. We compile with the *-S*, *-c* and *-emit-llvm* arguments to emit an LLVM IR file instead of a typical object file. We set the target architecture to BPF to avoid compiling with the native system architecture. Doing so may produce invalid code or include invalid ELF sections. We furthermore compile with the *-D* argument, which enables some functionality required by eBPF, such as *ASM_GOTO* support. *-I* includes the libbpf library and the *-Wall* and *-Werror* arguments will stop compilation if the eBPF program has any warnings. The *-g* argument will emit source-level debug information, which for example enables *bpftool* to read the contents of eBPF maps in a structured manner.

```
$ llc -march=bpf -filetype=obj -o kern.o
    kern.ll
```

When the IR file has been emitted, we convert it to an eBPF object file using *llc*. The arguments here are quite self-explanatory.

```
$ gcc -I./libbpf/src/root/usr/include/
    -L./libbpf/src/ -o ebpf-kill-example
    user.c \
    -Wl,-rpath=./libbpf/src/ -lbpf -lelf
```

Since the loader will run in user-space, we can compile this with gcc with typical arguments. We include the libbpf library as before, and we tell the linker where to look for the library at runtime using the *-Wl,-rpath* argument.

When loading eBPF code, the Just-In-Time step translates the generic eBPF byte-code instructions into instructions specific for the machine [5]. This optimises the execution speed of the program and makes it run as efficiently as natively compiled Linux code and code loaded as modules. The generic eBPF byte-code is being translated after the program is verified to avoid any overhead when executing the program [4]. The resulting machine-code is then placed at the pre-defined location next to kernel machine-code.

3.3 Loading

At a low-level, loading eBPF programs is done through the *bpf()* system call. Various languages have libraries wrapping around this call. For example, libbpf offers an interface in C to work with eBPF. It offers functions to build a struct *bpf_object* by reading the bytecode of a program and the associated metadata (map information, BTF information, etc.) from an ELF object file, and to later reuse this object to manipulate, load, and attach the eBPF programs and its related components.

In practise, loading eBPF programs can be done by invoking the `bpf_object__open_file` and `bpf_object__load` libbpf function with the name of the file. After a successful load, the program can be attached with the `bpf_program__attach` function. This takes a `bpf_program` as parameter, which can be retrieved using the `bpf_object__find_program_by_name` helper. The *by name* refers to the function name of the eBPF program. One can also find a program by title, which refers to the declared ELF section described in section 3. Furthermore, the `bpf` syscall can be used to perform commands on BPF maps or programs.

Listing 2 shows an example of a loader program, that will load the program seen in listing 1. The while loop will keep the eBPF program loaded while we listen to the trace pipe, which is located at `/sys/kernel/debug/tracing/trace_pipe`.

Put *very simply*, eBPF programs are by default unloaded when the user-space program that loaded the eBPF program terminates [18].

4 PROGRAM TYPES

In this section, we will describe a subset of the eBPF programs types. The full list of program types can be examined in appendix B.

4.1 Networking

Networking eBPF programs are used to read, modify, retransmit, redirect or drop network packets. The actions that can be performed on the packet (cloning,

```c
#include <bpf/bpf.h>
#include <bpf/libbpf.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
  char path[128];
  sprintf(path, "kill.o");

  struct bpf_object *obj;
  struct bpf_link *link = NULL;

  int err = -1;

  // Open eBPF object with the path
  obj = bpf_object__open_file(path, NULL);
  if (libbpf_get_error(obj)) {
    fprintf(stderr, "open BPF obj failed\n");
    return err;
  }

  // Find the program within the obj file
  struct bpf_program *prog =
      bpf_object__find_program_by_name(obj,
    "kill_example");
  if (!prog) {
    fprintf(stderr, "program not found\n");
    goto cleanup;
  }

  // Load the eBPF object into the kernel
  if (bpf_object__load(obj)) {
    fprintf(stderr, "loading failed\n");
    goto cleanup;
  }

  // Attach the program to the tracepoint
  link = bpf_program__attach(prog);
  if (libbpf_get_error(link)) {
    fprintf(stderr, "attach failed\n");
    link = NULL;
    goto cleanup;
  }

  err = 0;

  while(1) sleep(1);

cleanup:
  bpf_link__destroy(link);
  bpf_object__close(obj);

  return err;
}
```

Listing 2: An example of an eBPF loader program.

retransmission, redirection, ...) and the amount of data accessible from the context vary depending on the program type.

4.1.1 Socket Filter Programs. The eBPF Socket Filter type was the first type to be added to the kernel [4]. This type enables an eBPF program to attach to sockets and read packets going through the socket. It also allows truncation and dropping of packets.

4.1.2 XDP Programs. The eBPF XDP type enables eBPF programs to inspect incoming network packets early in the network stack [4]. This allows the the eBPF program to drop the packet, before the kernel has used a significant amount of time on it. Furthermore, contrary to DPDK, eBPF programs work with the kernel and can benefit from all it implements. It is also possible for some network drivers to offload XDP eBPF programs directly to network interface cards (*NIC*).

XDP programs can return *XDP_PASS* to allow it to continue to the next subsystem, *XDP_DROP* to drop it or *XDP_TX* to forward it back to the NIC that originally received it. Lastly, an XDP program can return *XDP_REDIRECT* to send the packet through a different NIC and possibly bypass the normal network stack.

XDP is very well suited for efficient low-level filtering such as a DDoS firewall.

4.2 Tracing

Tracing eBPF programs are used to debug or trace performance of either the kernel or user-space applications.

4.2.1 Tracepoint Programs. The eBPF Tracepoint type enables eBPF programs to attach to the tracepoint handler provided by the kernel [4]. Tracepoints are static marks in the kernel that can be used for tracing and debugging purposes. All tracepoints are defined in the `/sys/kernel/debug/tracing/events` directory.

When talking about tracepoints, it is important to remember that these are defined as certain marks or events in the kernel. A tracepoint can therefore not necessarily be reduced to a specific location or function in the kernel, but it tends to be much more stable between different kernel versions.

The 'kill' example from listing 1 is an example of a tracepoint program.

4.2.2 Raw Tracepoint Programs. The eBPF Raw Tracepoint type works like the *Tracepoint* type, but can access the tracepoint more directly [4]. For example, the context parameter is no longer a struct with the values, but instead a struct containing an array with pointers to the arguments. This may yield more detailed information about the kernel's current task and comes with a performance increase, as the kernel can skip argument processing.

4.2.3 Kprobe Programs. The eBPF Kprobe type enables eBPF programs to dynamically attach to any function in the kernel [4]. Kprobe programs differ from tracepoints in the section header and the context parameter. Kprobe programs are used for tracing in the situations where no suitable tracepoint exist. The important difference between kprobes and tracepoints is that tracepoints are statically defined in the kernel while kprobes can be placed in any named function in the kernel. Due to this, kprobes are also more likely to break between different kernel versions, because the functions or structs may change.

Since tracepoints are statically defined, it is much easier to extract contextual information. In listing 1 for example, we can access information about the the syscall from a struct that is passed to the eBPF program. Since Kprobe programs can hook into any kernel function, the context parameter is different from tracepoints. Instead, the parameter is a `structpt_regs`. This struct is defined in `asm/ptrace.h` and provides access to all CPU registers.

A Kprobe attaching to the `sys_exec` kernel function should set the section header (see section 3) to either `kprobe/sys_exec` or `kretprobe/sys_exec`. Setting the probe type to *kprobe* invokes the program as the first instruction of `sys_exec`, while setting the program to *kretprobe* invokes the program as the last instruction of `sys_exec`.

4.2.4 Perf Event Programs. The eBPF Perf Event type allows eBPF programs to attach to the kernels internal *Perf* profiler [4]. Perf emits performance data events for hardware and software. Low level examples of performance data are CPU cycles and CPU cache misses. Examples of more high level performance data are the number of context switches and page faults.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct {
  __uint(type, BPF_MAP_TYPE_ARRAY);
  __type(key, int);
  __type(value, int);
  __uint(max_entries, 42);
} my_map SEC(".maps");
```

Listing 3: An example of an eBPF map definition.

5 EBPF MAPS

eBPF maps offer a two-way data structure for transferring data in and out of kernel-space. Maps are the only way for an eBPF program to communicate with other eBPF program invocations and/or user-space. In the context of tracing, maps are often used to register key statistics about the current invocation. For example a networking eBPF program may store information about network latency or increment an IP address counter to keep track of popularity of remote hosts. The user-space program can at any point in time look into the maps and inspect their current state.

Maps are created by invoking the *bpf* syscall with the `BPF_MAP_CREATE` argument [4]. One can also make use the SEC attribute discussed earlier to automatically create it as shown in listing 3.

It is important to remember that eBPF maps are not built with functionality guaranteeing integrity, which means that the developer should take extra care in ensuring that data is not overwritten by accident. Furthermore, data is shared across eBPF program invocations. Lastly, all privileged user-space programs can access eBPF maps, which allows usage of debug tools such as bpftool.

An interesting property of eBPF maps is the in-kernel aggregation. If you, for example, want to compute the minimum or maximum value, you can determine this value in the eBPF program and thus not stream all values to user-space. This significantly decreases the overhead compared to systems that transfer all samples to user-space for processing.

5.1 Definition

Listing 3 shows an example of a simple eBPF map of the array-type. There exist many different map-types and the eBPF developer should consider the characteristics of each type. For example, the array-based map

```c
#include <bpf/bpf.h>
#include <bpf/libbpf.h>

/* create or update if exists */
#define BPF_ANY       0

/* create, but do no update */
#define BPF_NOEXIST   1

/* do not create, but only update */
#define BPF_EXIST     2

int bpf_map_lookup_elem(
  int fd, void *key, void *value
);

int bpf_map_update_elem(
  int fd, void *key,
  void *value, __u64 flags
);

int bpf_map_delete_elem(
  int fd, void *key
);
```

Listing 4: A list of the functions used to interact with eBPF maps in user-space.

has a fixed key-size of 4 bytes and the whole array is preallocated in memory, while a hash-based map can have any key-size and is not preallocated in memory [4]. However, an array-based map is faster than a hash-based map, since lookups do not require computing the hash of the entry. Furthermore, there exist some more complex map-types that can cover more specific use-cases. For example, one can initialise a map that is per-CPU or based on LRU-principles. A full list of eBPF map types are available in appendix C.

5.2 Usage

The user-space program can interact with maps by using the three methods shown in listing 4. These follow the typical interface for a map structure with the exception of the flag argument of update. Listing 4 also shows the update flags, which denotes whether the map should create or update, only create or only update.

The *fd* argument should be the file descriptor of the map. The file descriptor can be retrieved by first calling `bpf_object__find_map_by_name` with the `bpf_object` from the loading step and the map name. This

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

void *bpf_map_lookup_elem(
  void *map, void *key
);

int bpf_map_update_elem(
  void *map, void *key, void *value,
  unsigned long long flags
);

int bpf_map_delete_elem(
  void *map, void *key
);
```

Listing 5: A list of the functions used to interact with eBPF maps in kernel-space.

will return a `bpf_map`, which can then be passed to `bpf_map__fd` to retrieve the file descriptor.

The eBPF program can interact with the map by using the three methods shown in listing 5. This interface differs from from the user-space interface by using pointers instead of file descriptors. For example, `bpf_map_lookup_elem` returns a direct pointer to the value in kernel memory-space, while the user-space received a copy of the value.

The *map* argument should be a pointer to the struct containing the map definition.

6 EBPF VERIFIER

As described in section 2, all eBPF programs are verified before being loaded into the kernel [4, 7, 5]. There exist a set of rules to ensure the safety and stability of the kernel. Common rules include type checking of operations, a stack limit of 512 bytes, no signed division and the absence of loops [7, 4, 5]. The verifier will also ensure that the eBPF program is always terminating. The guarantee of termination is given by converting the program into a direct acyclic graph (DAG). The verifier can then check, using depth first search (DFS), that the program always finishes and does not include any dangerous paths [4]. Loops may be used if they are unrolled doing compilation or guaranteed to terminate.

While the eBPF verifier has been under scrutiny to guarantee its reliability, some critical security vulnerabilities have been found in the past. For example, CVE-2017-16995 describes a way to read and write kernel memory and bypass the eBPF verifier [14, 4].

6.1 Hardening

When an eBPF program passes verification it is run though a hardening step [5]. In this step, the kernel memory holding the eBPF program is made read-only to protect from malicious manipulation. The kernel will then crash, when an adversary or bug tries to invoke the eBPF program in an untimely manner. Constants may also be blinded such that code in constants cannot be executed. Blinded means that the memory address of the constant is randomised such that it is harder to guess. This ensures that an attacker cannot inject arbitrary code into the constant and execute it.

The verifier will also make sure that the eBPF program does not leak kernel-space memory to user-space, for example by reading any chunk of memory and forward it via eBPF maps to user-space. An example of a rule to prevent this; A register or a stack portion must always have been initialised before an eBPF program can read them

6.2 Risky Operations

One thing that differs significantly between typical user-space C programs and eBPF programs, is the safety guarantees of operations. When writing a user-space C program, invalid memory accesses are caught as segmentation faults. In eBPF programs, invalid memory accesses must not happen in any circumstances.

Programs that do not have the necessary safeguards will not be accepted by the verifier. To follow a pointer, for example, the program must use the `bpf_probe_read` function. This function will verify that the pointer is valid and copy the desired memory space before continuing the program execution.

7 PRACTICAL DIFFERENCES

In the previous sections, we have gone trough high-level descriptions of the architecture of eBPF and the differences between eBPF and typical user-space C programs. In this section, we will see a few select examples of code that works in a normal environment, but will not pass verification in the context of eBPF.

Listing 6 shows an example of an eBPF program that will hook into an arbitrary kprobe. The program will read the first parameter of the hooked function using the *rdi* entry of the context, as the first parameter is stored in the *rdi* register.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <asm/ptrace.h>

struct my_struct {
  unsigned int foo;
};

SEC("kprobe/...")
int bpf_prog(struct pt_regs *ctx) {

  struct my_struct *my_struct =
    (void *) ctx->rdi;

  char fmt[] = "Foo contains %u\n";

  bpf_trace_printk(
    fmt,
    sizeof(fmt),
    my_struct->foo,
    sizeof(my_struct->foo)
  );

  return 0;
}

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

Listing 6: An example of a risky operation that fails verification.

Bear in mind that the location of the first function parameter and the structure of *pt_regs* may differ between systems. One can use the macros defined in `bpf_tracing.h` in libbpf to increase portability by letting the host select the appropriate registers. For example, the *PT_REGS_PARM1* will expand to *ctx->rdi* on the author's system.

Compiling listing 6 succeeds, but when loading the program, you will see output like listing 7. The first lines (until line 18) shows what the eBPF verifier checked before failing verification. The hexadecimal numbers in the parenthesises denote the eBPF opcodes.

At line 19 the eBPF verifier informs that there was an *inv* on *R1*, which translates to invalid memory access on register 1, which on the author's system is equivalent to *rdi*.

Put more simply, the eBPF verifier fails verification, since the deferencing of the foo member of *my_struct* is risky. As described in section 6.2, this is due to the risk of segmentation faults as the pointer may not be valid. Section 6.2 also describes the solution to this

```
libbpf: load bpf program failed: Permission
    denied
libbpf: -- BEGIN DUMP LOG ---
libbpf:
btf_vmlinux is malformed
Unrecognized arg#0 type PTR
; (void *) ctx->rdi;
0: (79) r1 = *(u64 *)(r1 +112)
1: (18) r2 = 0xa752520736e6961
; char fmt[] = "Foo contains %u\n";
3: (7b) *(u64 *)(r10 -24) = r2
4: (18) r2 = 0x746e6f63206f6f46
6: (7b) *(u64 *)(r10 -32) = r2
7: (b7) r2 = 0
8: (73) *(u8 *)(r10 -16) = r2
last_idx 8 first_idx 0
regs=4 stack=0 before 7: (b7) r2 = 0
; bpf_trace_printk(
9: (61) r3 = *(u32 *)(r1 +0)
R1 invalid mem access 'inv'
processed 8 insns (limit 1000000)
    max_states_per_insn 0 total_states 0
    peak_states 0 mark_read 0

libbpf: -- END LOG --
```

Listing 7: The result of compiling and loading listing 6.

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <asm/ptrace.h>

SEC("kprobe/...")
int bpf_prog(struct pt_regs *ctx) {

  unsigned int tries = 0;

  while(1){
    if(bpf_get_prandom_u32() > 0) break;
    tries++;
  }

  return 0;
}

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

Listing 8: An example of a dynamic program that cannot be verified.

problem, which is to safely copy the memory space using a dedicated helper before accessing it.

The eBPF program seen in listing 8 will continuously sample random numbers. The program will terminate

```
libbpf: -- BEGIN DUMP LOG ---
libbpf:
btf_vmlinux is malformed
Unrecognized arg#0 type PTR
; if(bpf_get_prandom_u32() > 0) break;
0: (85) call bpf_get_prandom_u32#7
1: (67) r0 <<= 32
2: (77) r0 >>= 32
; if(bpf_get_prandom_u32() > 0) break;
3: (15) if r0 == 0x0 goto pc-4

from 3 to 0: R0_w=inv0 R10=fp0
; if(bpf_get_prandom_u32() > 0) break;
0: (85) call bpf_get_prandom_u32#7
1: (67) r0 <<= 32
2: (77) r0 >>= 32
; if(bpf_get_prandom_u32() > 0) break;
3: (15) if r0 == 0x0 goto pc-4

from 3 to 0: R0_w=inv0 R10=fp0
; if(bpf_get_prandom_u32() > 0) break;
0: (85) call bpf_get_prandom_u32#7
infinite loop detected at insn 1
processed 14 insns (limit 1000000)
    max_states_per_insn 0 total_states 1
    peak_states 1 mark_read 1

libbpf: -- END LOG --
```

Listing 9: The result of compiling and loading listing 8.

if and only if the sampled number is greater than zero. Since the used random number generator is providing unsigned 32-bit integers, the chance of the program not terminating immediately is negligible. Actually, the chance of the program not terminating in the first loop iteration is $\frac{1}{4294967296}$ or $\approx 0.00000002\%$.

Compiling listing 8 succeeds, but when loading the program, you will see output like listing 9. The output of the verifier states that an infinite loop was detected, although there is virtually no chance of an infinite loop occurring. This is, however, not a strong enough guarantee for loading the eBPF program.

Listing 10 shows an example of a simple program that samples a single random number and adds 42.0 to it. Floating point arithmetic is approximate by definition, due to the sheer number of values that can be represented [3]. There exist a range of issues with floating points including rounding issues, where numbers are incorrectly rounded up or down due to an approximation.

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
#include <asm/ptrace.h>

SEC("kprobe/...")
int bpf_prog(struct pt_regs *ctx) {

  float f = bpf_get_prandom_u32() + 42.0;

  char fmt[] = "Float is %f\n";

  bpf_trace_printk(fmt, sizeof(fmt), f,
    sizeof(f));

  return 0;
}

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

Listing 10: An example of an eBPF program with floating point arithmetic.

```
error: kill.c:31:13: in function
    kill_example i32 (%struct.pt_regs*): A
    call to built-in function '
    __floatunsidf' is not supported.

error: kill.c:31:35: in function
    kill_example i32 (%struct.pt_regs*): A
    call to built-in function '__adddf3' is
     not supported.

error: kill.c:31:13: in function
    kill_example i32 (%struct.pt_regs*): A
    call to built-in function '__truncdfsf2
    ' is not supported.

error: kill.c:35:38: in function
    kill_example i32 (%struct.pt_regs*): A
    call to built-in function '
    __extendsfdf2' is not supported.
```

Listing 11: The result of compiling listing 10.

It is hard, if not impossible, to guarantee the correct execution of a program when using floating points. Therefore, when compiling a program with floating points that cannot be optimised away, the compiler forcefully stops and warns that the built-in floating point arithmetical functions are not supported.

8 PRACTICAL USE CASES

In this section we will show some practical use cases of eBPF.

8.1 DDoS firewall

Cloudflare is currently transitioning into using XDP as their DDoS mitigator [1]. A clear benefit of using eBPF XDP over IPTables is the ability to match specific patterns that are not expressible using IPTables.

Cloudflare is interested in eBPF XDP for two main reasons. First, eBPF XDP offer a way to inspect packet in the lowest possible layer with a very low cost to drop packets. Second, it is possible to express firewall rules using high-level languages like C while maintaining strong guarantees about program termination and memory access.

8.2 ExtFUSE

ExtFUSE is a framework for developing extensible user file systems which enables applications to register specialised request handlers in the kernel [2]. This allows the application to meet their specific operative needs, while still having the advanced functionality of user-space programs.

ExtFUSE leverages eBPF to load and verify the user file system extensions, which enables the user to write extensions in a high-level language like C. It also guarantees the safety and stability of the file system extensions.

8.3 Cilium

Cilium is an open source system providing connectivity between applications in a secure and transparent manner [10]. Cilium works with Linux container management systems like Kubernetes, Docker and Mesos.

Cilium aims to make Linux aware of microservices, including their containers and APIs. This allows users to insert flexible and powerful security, visibility and networking control logic into the kernel using eBPF.

8.4 Katran

Katran is a high-performance XDP-based layer 4 load balancer built by Facebook Incubator [12]. Since Katran uses XDP, it is able to run packet handling routines right after packet has been received by the NIC.

8.5 bcc

bcc is a toolkit for enabling efficient kernel tracing [15]. bcc uses eBPF under the hood and as such, it enables developers to write eBPF programs more easily. BCC

is suitable for a wide amount of tasks including performance analysis and network traffic control.

8.6 bpftrace

bpftrace is a high-level tracing language based on eBPF [16]. bpftrace enables developers to write one-liners to gauge the performance of a system. It is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap.

9 COMPUTATIONAL STORAGE

NVMe, the abbreviation for Non-Volatile Memory Express, is a specialised protocol designed exclusively for solid-state drives (SSDs) that utilise PCIe interfaces. This innovative protocol offers an efficient means of accessing and transferring data between computer systems and storage devices.

With the introduction of TP 4091 in NVMe, eBPF is expected to become the standardised method for offloading programs to storage. The combination of eBPF and NVMe's TP 4091 permits the swift offloading of programs directly to storage devices with a vendor-neutral instruction set architecture.

As of this writing, TP 4091's contents remain confidential, making it challenging to predict how eBPF will be used in computational storage devices. Important issues such as state management and resource allocation have yet to be resolved.

The IT University of Copenhagen is actively experimenting with a PCIe-based computational storage processor (CSP) that utilises eBPF as part of the DAPHNE project [6]. This system, known as Delilah [11], allows user-space applications to queue up eBPF programs and execute them on the device while taking into account the underlying storage.

10 NEXT STEPS

This paper only scratches the surface of what you can do with eBPF. In this section we will briefly highlight some of the next steps an eBPF developer can take.

- *High-level inspection of eBPF objects.* An eBPF developer can use tools such as bpftool to view and debug eBPF programs and maps.
- *Low-level inspection of eBPF programs.* Part of understanding and debugging the behaviour of eBPF programs is dumping and reading the bytecode. The eBPF developer can use tools such as *llvm-objdump* to see the underlying bytecode of an eBPF program.
- *Evaluate performance of eBPF programs.* Since eBPF programs are often running in performance sensitive environments, it is valuable for an eBPF developer to understand the overhead of eBPF programs. In Linux, one can enable a flag that will collect performance metrics about loaded eBPF programs.
- *CO-RE.* One of the challenges of eBPF is the portability of code. *Compile Once - Run Everywhere* allows eBPF developers to compile code on a system to be run on any other system with varying operating systems and kernel versions.
- *.. and much more.* eBPF has a wide range of different systems and mechanisms. This includes, for example, the ability to trace eBPF programs with eBPF and using the virtual filesystem for eBPF.

11 CONCLUSION

Extended Berkeley Packet Filter or eBPF is a low-level language based on the Berkeley Packet Filter system from 1992, but with an architecture that more closely resembles contemporary processors.

eBPF programs can be compiled from several high-level languages like C and Python. However, there are key differences between typical user-space C programs and eBPF programs. This is due to the strict security guarantees of eBPF programs, which requires programs to always be predictable and stable.

In this paper we showed how to write, compile and load eBPF programs. We furthermore discussed the different types of eBPF programs available as well as a short overview of eBPF maps.

We discussed some of the differences between typical user-space C programs and eBPF C programs. We saw how these programs may seem correct and functional, but have edge cases that prevents giving strong enough safety guarantees.

We described several use cases of eBPF including a DDoS firewall and file system extensions. We showed how these project leverage eBPF and the properties of eBPF.

Lastly, we described the eBPF verifier and discussed the security of it.

REFERENCES

[1] Gilberto Bertin. "XDP in practice: integrating XDP into our DDoS mitigation pipeline". In: *Technical Conference on Linux Networking, Netdev*. Vol. 2. 2017.

[2] Ashish Bijlani and Umakishore Ramachandran. "Extension framework for file systems in user space". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 121–134.

[3] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Global Edition. Pearson, 2016. ISBN: 9781292101767.

[4] D. Calavera and L. Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, Incorporated, 2019. ISBN: 9781492050209.

[5] Cilium. *What is eBPF? An Introduction and Deep Dive into the eBPF Technology*. URL: https://ebpf.io/what-is-ebpf/ (visited on 12/14/2020).

[6] Patrick Damme et al. "DAPHNE: An Open and Extensible System Infrastructurefor Integrated Data Analysis Pipelines". en. In: *12th Annual Conference on Innovative Data Systems Research (CIDR '22)*. Santa Cruz, California, USA: CIDR, Jan. 2022. URL: https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf.

[7] Henri Maxime Demoulin et al. "Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FineLame". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, 2019, pp. 693–708. ISBN: 9781939133038. URL: https://www.usenix.org/conference/atc19/presentation/demoulin.

[8] Matt Fleming. *A thorough introduction to eBPF*. 2017. URL: https://lwn.net/Articles/740157/ (visited on 11/10/2020).

[9] Stephen J. Friedl. *Using GNU C __attribute__*. URL: http://unixwiz.net/techtips/gnu-c-attributes.html (visited on 11/03/2020).

[10] Thomas Graf. *How to Make Linux Microservice-Aware with Cilium and eBPF*. 2018. URL: https://www.youtube.com/watch?v=_Iq1xxNZOAo (visited on 12/10/2020).

[11] Niclas Hedam et al. "Delilah: eBPF-offload on Computational Storage". en. In: *19th International Workshop on Data Management on New Hardware (DaMoN '23)*. Seattle, Washington, USA: DaMoN, 2023. DOI: 10.1145/3592980.3595319. URL: https://hed.am/papers/2023-DaMoN.pdf.

[12] Facebook Incubator. *facebookincubator / katran*. 2021. URL: https://github.com/facebookincubator/katran (visited on 02/18/2021).

[13] S. Miano et al. "Creating Complex Network Services with eBPF: Experience and Lessons Learned". In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. 2018.

[14] *NVD - CVE-2017-16995*. 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-16995.

[15] IO Visor Project. *iovisor / bcc*. 2021. URL: https://github.com/iovisor/bcc (visited on 02/18/2021).

[16] IO Visor Project. *iovisor / bpftrace*. 2021. URL: https://github.com/iovisor/bpftrace (visited on 02/18/2021).

[17] Alexei Starovoitov. *daedfb22451d*. 2014. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=daedfb22451dd02b35c0549566cbb7cc06bdd53b.

[18] Alexei Starovoitov. *Lifetime of BPF objects*. 2018. URL: https://facebookmicrosites.github.io/bpf/blog/2018/08/31/object-lifetime.html (visited on 02/17/2021).

A EXAMPLE SETUP

When compiling eBPF programs, a set of helper functions and libraries are needed. In this section, we will go through all the steps necessary to compile any of the examples in this paper. Bear in mind that as the kernel, operating systems and architecture of eBPF continues to be developed, this walk-through may become outdated. The walk-through was written in May 2023 and tested on an Ubuntu 22.04 LTS with kernel 5.19 and libbpf 1.2.

To compile the examples, you will need the following dependencies.

- *build-essential, git, make* – These tools provide the necessary framework for compilation. Git will be used to clone libbpf.
- *gcc, clang, llvm* – GCC is used to compile the kernel, while clang is used to compile eBPF code.

LLVM is used to transform LLVM IR code to BPF byte-code.

- *libelf-dev, gcc-multilib* – These libraries are required by libbpf.

Start out by retrieving new lists of packages.

```
$ sudo apt-get update
```

After completion of the update, install all of the dependencies listed above.

```
$ sudo apt install -y build-essential git
    make gcc clang llvm libelf-dev
    gcc-multilib make
```

Download a copy of libbpf. We will fetch only a single commit containing the release of version 1.2.

```
$ git clone --depth 1 --single-branch
    --branch v1.2.0 \
    https://github.com/libbpf/libbpf libbpf
```

Compile libbpf and install the headers locally.

```
$ make --directory=libbpf/src all

$ DESTDIR=root make --directory=libbpf/src
    install_headers
```

Now that we have all the dependencies in place, we can continue to the eBPF-related files. Grab the loader from listing 2. We will save it as `loader.c`. We also need to grab one of the examples. Let us grab the kill-example from listing 1 and save as `kill.c`. Please be aware that copying code from a PDF may not always work as expected.

Then, we need to compile the eBPF program using clang.

```
$ clang -target bpf -S -D __BPF_TRACING__
    -I./libbpf/src/root/usr/include/ -Wall
    -Werror -O2 -emit-llvm -c -g kill.c

$ llc -march=bpf -filetype=obj -o kill.o
    kill.ll
```

Compile the loader.

```
$ gcc -I./libbpf/src/root/usr/include/
    -L./libbpf/src/ -o ebpf loader.c \
    -Wl,-rpath=./libbpf/src/ -lbpf -lelf
```

This is all we need to do to. All that is left is to actually run the loader program.

```
$ sudo ./ebpf &

$ sudo cat
    /sys/kernel/debug/tracing/trace_pipe
```

You may have to press enter after loading the program.

To stop the eBPF program, stop the cat command using CTRL + C and run *fg*. This will bring the eBPF loader back to the front and you can stop it using CTRL + C.

B EBPF PROGRAM TYPES

```
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,
};
```

Listing 12: The declaration of all eBPF program types. From bpf.h of kernel version 6.2.11.

C EBPF MAP TYPES

```
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 =
    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,
};
```

Listing 13: The declaration of all eBPF map types. From bpf.h of kernel version 6.2.11.