eBPF - From a Programmer's Perspective

Niclas Hedam
IT University of Copenhagen
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 or alter user-space system contexts when a kernel function is invoked. In this paper, we will describe and discuss the architecture of eBPF 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.

1 INTRODUCTION

Berkeley Packet Filter or *BPF* emerged as an efficient network packet filter in 1992 [4, 10]. A network packet filter is a network security mechanism for controlling what flows from and to a network by inspecting packets as they pass though 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 [4].

Alexei Starovoitov introduced eBPF in 2014 [12, 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) [7]. In the aforementioned commit, eBPF was exposed to user-space and soon after, eBPF stopped being limited to the networking stack. Over time eBPF programs evolved to look like kernel modules [4, 10].

To avoid having an unnecessarily large kernel, many components such as drivers can be enabled and disabled when the kernel is configured. When a module is changed or has to be made available, one must recompile the kernel which depending on the system can take up to hours. After recompilation, the system must be restarted to load the new module. eBPF has some of the capabilities of modules, but without requiring recompilation of the kernel and a restart of the system [4, 10].

eBPF is built with a static verifier, that ensures that a module cannot cause a kernel crash. 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 probe types, which will be described later in this paper. The eBPF architecture also contain maps, which are bidirectional data structures allowing eBPF programs to asynchronously share data with user-space [4].

2 ARCHITECTURE

```c
#include <linux/bpf.h>
#include "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 bpf_prog(struct
    syscalls_enter_kill_args *ctx) {

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

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

  return 0;
}

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

Listing 1: Kill eBPF example written in C. This example is of a tracepoint eBPF program (see section 3.2.1).

Listing 1 is an example of a eBPF program that attaches to the *kill* system call. The example above 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 program registers for this event using the *SEC* macro defined in the `bpf_helpers.h` file. 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 [8]. In the context of eBPF, the attribute is used to put code at specific places in the ELF binary

The section names follow a convention; 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 probe type is *tracepoint* and the hook is `syscalls/sys_enter_kill`. Furthermore, we use `bpf_trace_printk` to print out trace information to the common trace pipe (`/sys/kernel/debug/tracing/trace_pipe`). We will describe probe types in more details in section 3.

All eBPF programs takes a context as parameter. The context contains information about the information that the kernel is currently processing [4] including registers or function parameters. The context depends on the type of eBPF program as well as the location of the probe. In the listing above, the context is a `syscalls_enter_kill_args` struct, which follows the format published by the kernel[1]. The first 8 bytes is unused and should be ignored. We will describe the context parameter in more details in section 3.

In the bottom of listing 1 the license of the eBPF program is declared. Since the kernel is licensed under GPL, the kernel only loads programs that are also licensed under GPL.

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

2.1 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, call kernel functions exposed via the kernel headers like typical user-space programs. eBPF programs can also 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 any GNU C Library (glibc) functions as long as the function adheres to the requirements of the verifier, which are described in section 5.

2.2 Compilation

The rest of the eBPF paper assumes that you have a working eBPF environment to run the examples. If you have not compiled or run a 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 Python and C [4, 6]. In this paper, we will focus on C and compilation using *clang*. 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 therefore regarded as the tool of reference in the eBPF community.

When compiling using clang one may specify the target architecture. For example, compiling listing 1 can be done by calling clang with the *-target bpf* option. Ultimately, the command to compile the program is *clang -O2 -target bpf -c example1.c -o example1.o*. We choose to compile with optimisation level 2 as this is a reasonably level for most eBPF programs. Compiling with the *-target bpf* flag enables the compiler to remove unnecessary ELF sections and disable unsupported optimisations. Furthermore, it allows the compiler to stop compiling when unsupported inline assembly code is found. However, since C compilers try to stay architecture independent, eBPF programs can often be compiled successfully without the *-target bpf* flag.

When loading eBPF code, the Just-In-Time step translates the generic 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 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.

2.3 Loading

Loading eBPF programs is done by invoking the `load_bpf_file` kernel function with the name of the file with the compiled bytecode. In version 5.8 of the kernel, this function is defined in `samples/bpf/bpf_load.h` and implemented in `samples/bpf/bpf_load.c`.

```
#include "bpf_load.h"
#include "trace_helpers.h"

int main(int argc, char **argv) {
  if (load_bpf_file("src/kill.o") != 0) {
    printf("Load failed\n");
    return -1;
  }

  read_trace_pipe();

  return 0;
}
```

Listing 2: An example of a eBPF loader program.

Listing 2 shows an example of a loader program, that will load the program seen in listing 1 and output all of the messages received from it. In this case, the loader will print a message every time a process is being forcefully killed with the *-9* signal. The `read_trace_pipe` function continuously reads the trace pipe, which is located at `/sys/kernel/debug/tracing/trace_pipe`.

eBPF programs are automatically unloaded, when the user-space program that loaded the eBPF program terminates. Therefore, the eBPF program can in principle be seen as sub-process of the user-space program that invoked `load_bpf_file`.

3 PROGRAM TYPES

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

3.1 Networking

Networking eBPF programs are used to read, modify or drop network packets. The capabilities and access varies among eBPF programs.

3.1.1 Socket Filter Programs. The eBPF Socket Filter type was the first type to be added to the kernel [4]. This type enables a eBPF program to attach to sockets and read packets going through the socket, but does not allow altering or deletion of packets.

3.1.2 XDP Programs. The eBPF XDP type enables eBPF programs to inspect incoming network packets early in the call stack [4]. This allows the the eBPF program to drop the packet, before the kernel has used a significant amount of time on it. 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 network interface card (NIC) that originally received it.

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

3.2 Tracing

Tracing eBPF programs are used to debug or trace performance of either the kernel or applications. These eBPF programs can also be used to alter the context of kernel functions, either as the first or last instruction in a kernel function.

3.2.1 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.

3.2.2 Raw Tracepoint Programs. The eBPF Raw Tracepoint type works like the *Tracepoint* type, but can access 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 kernels current task, but it comes with a small performance overhead.

In many cases it is worth using the traditional *Tracepoint* type over the *Raw Tracepoint* type, as *Tracepoint* is more simple and more efficient.

3.2.3 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.

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 2) 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`.

3.2.4 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.

4 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 a 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 a IP address counter to keep track of popularity of remote hosts. The user-space program can at any point in time look into this program and inspect the 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. Access to the eBPF map is limited to the user-space program loading the eBPF program, as well as the eBPF program itself. Data is shared across eBPF program invocations.

4.1 Definition

```
struct bpf_map_def SEC("maps") my_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 100,
    .map_flags = BPF_F_NO_PREALLOC
};
```

Listing 3: An example of a eBPF map definition.

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

4.2 Usage

```
/* 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.

The user-space program can interact with the map 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 found in the `map_fd` global array. The array contains all the file descriptors of eBPF maps in the order they are defined in the program.

```
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.

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.

5 EBPF VERIFIER

As described in section 1, all eBPF programs are verified before being loaded into the kernel [4, 6, 5]. There exist a set of rules to ensure the safety and stability of the kernel. Common rules includes disallowing floating point instructions, a limitation of 4096 instructions per program, a stack limit of 512 bytes, no signed division and the absence of loops [6, 4, 5]. Loops can be used if it can be unrolled doing optimisation or is guaranteed to terminate. 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].

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 [11, 4].

5.1 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 are also blinded such that code in constants cannot be executed.

5.2 Risky Operations

One thing that differs significantly between typical user-space C programs and eBPF programs, are 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 does not have the necessary safeguards will not be accepted by the verifier. For example, to follow a pointer the program must use the `bpf_probe_read` function. This function will copy the desired memory space and verify that the pointer is valid before continuing the program execution.

6 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.

```
1  struct my_struct {
2    unsigned int foo;
3  };
4
5  SEC("kprobe/...")
6  int bpf_prog(struct pt_regs *ctx) {
7
8    struct my_struct *my_struct =
9      (void *) ctx->rdi;
10
11   char fmt[] = "Foo contains %u\n";
12
13   bpf_trace_printk(
14     fmt,
15     sizeof(fmt),
16     my_struct->foo,
17     sizeof(my_struct->foo)
18   );
19
20   return 0;
21 }
22
23 char _license[] SEC("license") = "GPL";
```

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

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.

Bear in mind that the location of the first function parameter and the structure of *pt_regs* may differ between systems. From version 5.5 of the kernel, one can use the macros defined in `tools/lib/bpf/bpf_tracing.h` 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.

```
1  bpf_load_program() err=13
2  0: (79) r1 = *(u64 *)(r1 +112)
3  1: (18) r2 = 0xa752520736e6961
4  3: (7b) *(u64 *)(r10 -16) = r2
5  4: (18) r2 = 0x746e6f63206f6f46
6  6: (7b) *(u64 *)(r10 -24) = r2
7  7: (b7) r2 = 0
8  8: (73) *(u8 *)(r10 -8) = r2
9  last_idx 8 first_idx 0
10 regs=4 stack=0 before 7: (b7) r2 = 0
11 9: (61) r3 = *(u32 *)(r1 +0)
12 R1 invalid mem access 'inv'
13 processed 8 insns (limit 1000000)
       max_states_per_insn 0 total_states 0
       peak_states 0 mark_read 0
```

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

Compiling listing 6 succeeds, but when loading the program, you will see output like listing 7. The first line shows that the eBPF loader failed with error code 13, and the next lines (until line 12) shows what the eBPF verifier checked before failing verification. The hexadecimal numbers in the parenthesises denotes the eBPF opcode.

At line 12 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 simple, the eBPF verifier fails verification, since the deferencing of the foo member of *my_struct* is risky. As described in section 5.2, this is due to the risk of segmentation faults as the pointer may not be valid. Section 5.2 also describes the solution to this problem, which is to safely copy the memory space before accessing it.

```
1  SEC("kprobe/...")
2  int bpf_prog(struct pt_regs *ctx) {
3
4    unsigned int tries = 0;
5
6    while(1){
7      if(bpf_get_prandom_u32() > 0) break;
8      tries++;
9    }
10
11   return 0;
12 }
13
14 char _license[] SEC("license") = "GPL";
```

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

The eBPF program seen in listing 8 will continuously sample random numbers. The program will terminate 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 $\approx 0.00000002\%$.

```
bpf_load_program() err=22
0: (85) call bpf_get_prandom_u32#7
1: (67) r0 <<= 32
2: (77) r0 >>= 32
3: (15) if r0 == 0x0 goto pc-4
 R0_w=inv(id=0,umax_value=4294967295,
    var_off=(0x0; 0xffffffff)) R10=fp0
4: (b7) r0 = 0
5: (95) exit

from 3 to 0: R0_w=inv0 R10=fp0
0: (85) call bpf_get_prandom_u32#7
1: (67) r0 <<= 32
2: (77) r0 >>= 32
3: (15) if r0 == 0x0 goto pc-4
 R0_w=inv(id=0,umax_value=4294967295,
    var_off=(0x0; 0xffffffff)) R10=fp0
4: (b7) r0 = 0
5: (95) exit

from 3 to 0: R0_w=inv0 R10=fp0
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
```

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

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.

```
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 a eBPF program with floating point arithmetic.

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.

```
src/bpf_program.c:7:5: error: A call to
    built-in function '__floatunsidf' is
    not supported.
int bpf_prog() {
    ^
src/bpf_program.c:7:5: error: A call to
    built-in function '__adddf3' is not
    supported.
src/bpf_program.c:7:5: error: A call to
    built-in function '__truncdfsf2' is not
     supported.
src/bpf_program.c:7:5: error: A call to
    built-in function '__extendsfdf2' is
    not supported.
4 errors generated.
make: *** [Makefile:35: build] Error 1
```

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. It is worth noting that this safeguard only exist when setting the target to bpf as described in section 2.

```
bpf_load_program() err=22
unknown opcode 00
processed 0 insns (limit 1000000)
    max_states_per_insn 0 total_states 0
    peak_states 0 mark_read 0
```

Listing 12: The result of compiling and loading listing 10.

If the target was not set to bpf, the compilation will succeed. Compiling listing 10 succeeds, but when loading the program, you will see output like listing 12. This states that the program contained an operation that was not part of the eBPF instruction set.

7 PRACTICAL USE CASES

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

7.1 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.

7.2 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 it's 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.

7.3 Cilium

Cilium is an open source system providing connectivity between applications in a secure and transparent manner [9]. 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 CONCLUSION

Extended Berkeley Packet Filter or eBPF is a low-level language that can be compiled to from multiple high-level languages such as Python and C. It is a redesign of the Berkeley Packet Filter system from 1992 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] 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.

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

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

[9] 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).

[10] 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.

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

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

A EXAMPLE SETUP

When compiling eBPF programs, a set of helper functions and libraries are needed. These are currently not distributed with the kernel headers. 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 January 2020 and tested on an Ubuntu 20.04 LTS with kernel 5.4 LTS.

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 the kernel.
- *gcc, clang* – GCC is used to compile the kernel, while clang is used to compile eBPF code.
- *libelf-dev, gcc-multilib* – These libraries contain necessary headers for compilation. Libelf are required by the eBPF loader and gcc-multilib makes `asm/types.h` available, which is required by the eBPF program.

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 libelf-dev gcc-multilib
```

As mentioned earlier, the kernel contains functions needed to load and run eBPF programs. Therefore, we need to download a copy of the kernel. We will fetch only a single commit containing the release of kernel version 5.4, which is a Long-Term Support (LTS) release.

```
$ git clone --depth 1 --single-branch
    --branch v5.4 \
    https://github.com/torvalds/linux.git
    kernel-src
```

Go to the BPF folder and compile the BPF library.

```
$ cd kernel-src/tools/lib/bpf && make &&
  make install prefix=../../../../ &&
  cd ../../../../
```

Now that we have all the dependencies installed as well as the kernel source code, we can continue to the eBPF-related files. First create the `src` folder.

```
$ mkdir src
```

Grab the loader from listing 2. We will save it as `src/loader.c`. We also need to grab one of the examples. Let us grab the kill-example from listing 1 and save as `src/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 -O2 -target bpf -c src/kill.c
  -Ikernel-src/tools/testing/selftests/bpf
  -Ikernel-src/tools/lib/bpf -o src/kill.o
```

Compile the loader.

```
$ clang -O2 -o src/ebpf -lelf
  -Ikernel-src/samples/bpf
  -Ikernel-src/tools/lib
  -Ikernel-src/tools/lib/bpf
```

```
  -Ikernel-src/tools/perf
  -Ikernel-src/tools/include
  -Ikernel-src/tools/testing/selftests/bpf
  -Llib64 -lbpf
  kernel-src/samples/bpf/bpf_load.c
  -DHAVE_ATTR_TEST=0 src/loader.c
```

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

```
$ sudo
  LD_LIBRARY_PATH=lib64/:$LD_LIBRARY_PATH
  ./src/ebpf
```

If you compiled the kill example, open a new terminal and force-kill an arbitrary process to trigger the program.

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,
};
```

Listing 13: The declaration of all eBPF program types. From bpf.h of kernel v5.10.6.

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,
  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,
};
```

Listing 14: The declaration of all eBPF map types. From bpf.h of kernel v5.10.6.