Contents

Oxidize eBPF: eBPF programming with Rust

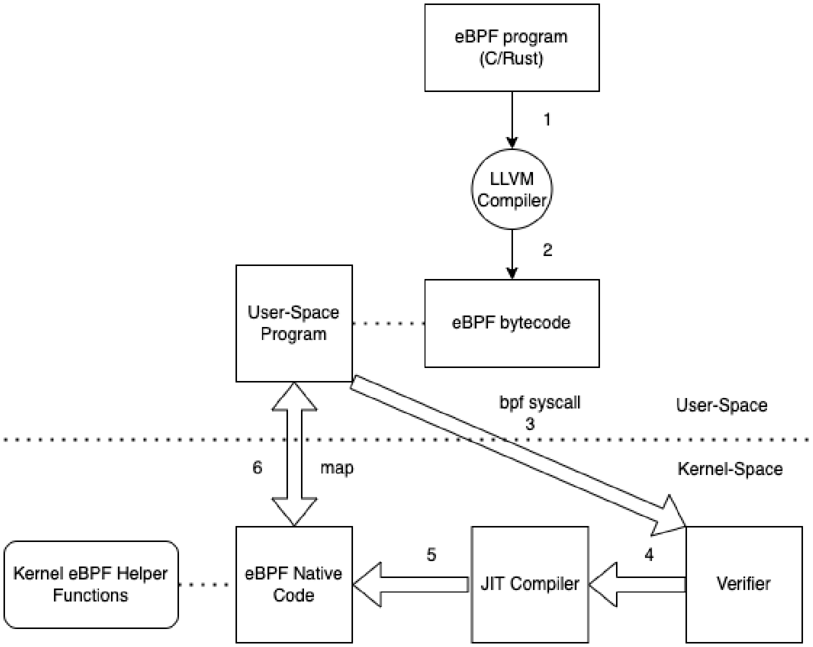
Introduction

The eBPF (extended Berkeley Packet Filter) infrastructure for Linux provides a **virtual machine (VM)** that can run programs safely inside the Linux kernel. These programs either modify the kernel behavior or safely exchange data between the kernel and the user space. The eBPF programs are event-driven and are run when the kernel or an application passes specific hook points. The hooks are predefined and include network events, system calls, function entry and exit, kernel tracepoints, and several others.

The eBPF infrastructure consists of the following parts.

Component	Location	Functionality
Compiler	Userspace	Convert eBPF program to bytecode
bpf syscall	Kernel	Load eBPF bytecode into the kernel
Verifier	Kernel	Ensure the program safety
JIT (Just In Time) compiler	Kernel	Convert bytecode to machine instructions

Component	Location	Functionality
VM (Virtual Machine)	Kernel	Run the eBPF program



The eBPF program is usually written using a subset of high-level programming languages such as C or Rust and compiled to bytecode utilizing the LLVM compiler toolchain. A user space program loads the bytecode object into the kernel via the **bpf** system call. As part of loading the program in the kernel space, the eBPF verifier inside the kernel checks the validity of the bytecode before accepting it into the kernel. The verifier ensures that the eBPF program is safe to run and will terminate. Once all of the checks pass, the eBPF program is loaded into the kernel at the intended location in a kernel code path (hook), and it then waits for an appropriate event. When the event is received, the eBPF program is loaded, and the VM executes its bytecode. The eBPF infrastructure also includes an optional JIT compiler that converts eBPF bytecode to machine code, thereby enhancing the performance of the eBPF program by executing it directly using native instructions.

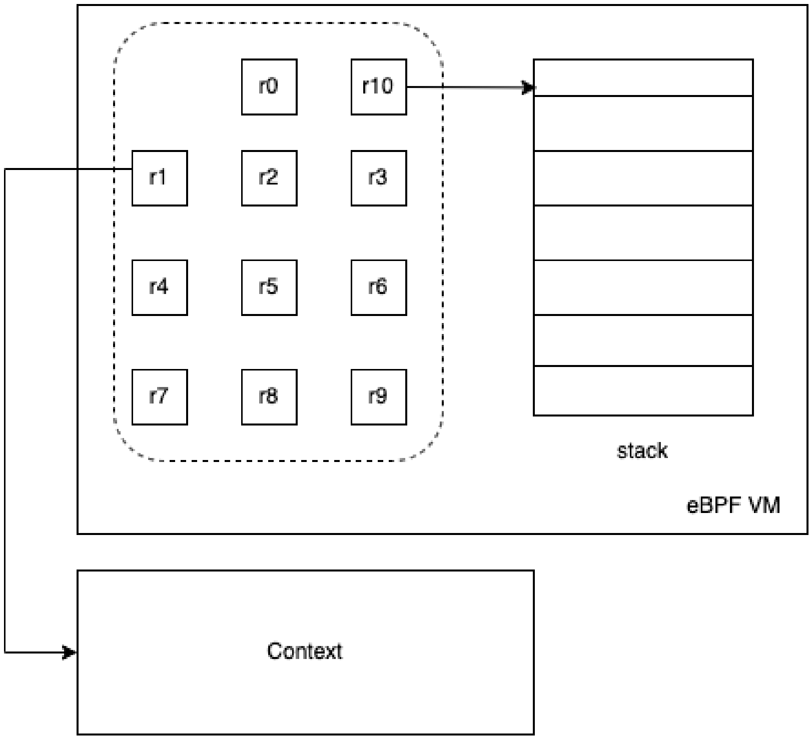
Data exchange between the kernel space and the user space is made possible using data structures called **maps**. The hook point to which an eBPF program can be attached and executed is determined by the **program type**. These concepts will be covered in detail in later chapters.

Part I

eBPF Virtual Machine

This chapter will go over the internals of the eBPF Virtual Machine (VM) with the help of an opensource **rbpf** project to illustrate how the VM executes an eBPF program. **rbpf** (https://github.com/qmonnet/rbpf) is the Rust implementation of the eBPF VM in the user space and provides a convenient way to try out the functionality of the eBPF VM in the user space. The workings of the VM will be illustrated by implementing a few sample programs using eBPF VM instructions. This chapter does not dwell on all of the details of the VM but provides enough context to understand how eBPF programs get executed in the Linux kernel.

Architecture



The eBPF VM is a 64-bit RISC (Reduced Instruction Set Computer) machine. The machine consists of 11 64-bit registers, a program counter (PC), and a 512-byte stack. Nine registers (r1 - r9) are general-purpose read-write registers, one register (r10) is a read-only stack-pointer, and the PC is implicit. The following table explains the functionality of each register.

Register	Functionality
r0	return value from function calls and exit value for eBPF programs

Register	Functionality
r1-r5	arguments for function calls. Upon program start, r1 contains the address of the input memory
r6-r9	callee saved registers that function calls will preserve
r10	read-only frame pointer to access stack

Each function call can have at most five arguments in registers r1-r5. This restriction applies to the eBPF program functions, ebpf-to-ebpf calls, and kernel helper function calls. Registers r1-r5 can only store numbers or pointers to the stack (to be used as arguments to functions), never direct pointers to arbitrary memory. All memory accesses must be done by first loading data to the eBPF stack before using it in the eBPF program. This restriction helps the eBPF verifier and simplifies the memory model to enable more straightforward correctness checking. When an eBPF program is started, the register r1 contains a pointer to the **context** memory, which serves as input data for the eBPF program to act upon.

Verifier

When an eBPF program is loaded into the Linux kernel using the **bpf** system call, before executing the program, it is first verified to make sure that it does not cause any ill side effects. This task is accomplished by running the eBPF program through a verifier. If the verification checks fail, the bpf system call returns -1 with an appropriate error set in **errno**.

The verification of an eBPF program is done using two steps.

- **Control Flow Analysis**: First, a DAG (Directed Acyclic Graph) of the eBPF instructions making up the program is generated, and this graph is traversed by the verifier using the DFS (Depth First Search) algorithm to ensure there are no loops, to prevent the eBPF programs from running forever. As part of walking the DAG, the verifier also ensures that there are no unused instructions (dead code) in the eBPF program and that the program has an expected end.

- **Data Flow Analysis**: The second check begins by starting at the first instruction and navigating all the possible code paths from it. The verifier monitors the registers and the stack while simulating all possible instruction executions and ensures that the program execution completes without causing any data-related issues, such as register or stack overflows.

Only when these checks pass is the eBPF program considered safe for execution. Please note the rbpf verifier does not perform any of these sophisticated checks.

Tail Calls

The eBPF VM limits the size of an eBPF program to restrict the problem size for the verification step. As of writing this book, the current limit is one million instructions. To overcome this limit on the eBPF program size, kernel functionality can be implemented by chaining multiple eBPF programs. An eBPF program can then call another eBPF program (eBPF-to-eBPF call) using a tail call (bpf_tail_call). As of writing this book, the tail call chain is limited to 32 calls.

Helper Functions

The eBPF VM architecture enables attaching of **helper functions** to the VM, which can then be directly called by the eBPF programs executed by the VM. This functionality will be demonstrated in the later part of the chapter.

Instruction Types

There are eight broad categories of instructions supported by the eBPF VM.

Type	Examples
LD	non-standard load operations
LDX	load into register operations
ST	store from immediate operations
STX	store from register operations
ALU32	32-bit arithmetic operations
ALU64	64-bit arithmetic operations
JMP32	32-bit jump operations
JMP64	64-bit jump operations

The following test programs will help illustrate how an eBPF program can be written using these instructions and executed on an eBPF VM.

Examples

Add two integer constants

```
extern crate rbpf;
use rbpf::helpers;
use rbpf::assembler::assemble;

#[test]
fn test_vm_add() {
  let prog = assemble("
      mov32 r0, 1
      mov32 r1, 2
      add r0, r1
      exit").unwrap();
  let vm = rbpf::EbpfVmNoData::new(Some(&prog))
             .unwrap();
  assert_eq!(vm.execute_program().unwrap(),
             0x3);
}
```

The **assemble** function takes an eBPF assembly program and converts it into eBPF bytecode. In the example above, we set the register **r0** to an immediate value of 1 and register **r1** to an immediate value of 2. The value of register r1 is then added to r0 to get a value of 3, which is verified by the assertion.

Add two integers stored in the context memory

Let us provide the two 32-bit integers as input to the VM using the context memory.

```
#[test]
```

```
fn test_vm_add_context() {
  let prog = assemble("
      ldxw r0, [r1]
      ldxw r1, [r1+4]
      add r0, r1
      be32 r0
      exit").unwrap();
  let mem = &mut [
      0x0, 0x22, 0x0, 0x0,
      0x0, 0x0, 0x33, 0x0
  ];
  let vm = rbpf::EbpfVmRaw::new(Some(&prog))
            .unwrap();
  assert_eq!(vm.execute_program(mem).unwrap(),
            0x00223300);
}
```

In this example, the eBPF program loads two 4-byte integers to be added from the context memory. The **ldxw** instruction reads 4 bytes from memory pointed to by **r1** and stores it in register **r0**. The program loads the next 4 bytes from the context memory into register **r1**. The add instruction adds the integer value stored in r1 to the integer value stored in r0. Finally, the result stored in r0 is converted to big-endian format to ease the comparison in the assert statement.

Calculate Fibonacci Number

```
#[allow(unused_variables)]
pub fn fib(r1: u64, r2: u64, r3: u64,
    r4: u64, r5: u64) -> u64 {

    let mut i = 1;
    let mut j = 1;
    let mut k = r1;

    while k - 2 > 0 {
        j = i + j;
        i = j - i;
        k -= 1;
    }
    j
}

#[test]
fn test_vm_fib() {

    let prog = assemble("
        mov r2, r1
        ldxw r1,[r1]
        call 0
        stxw [r2+4], r0
        exit").unwrap();

    let mem = &mut [
        0xa, 0x0, 0x0, 0x0,
        0x0, 0x0, 0x0, 0x0
    ];
    let mut vm = rbpf::EbpfVmRaw::new(Some(&prog))
                    .unwrap();
    vm.register_helper(0, fib).unwrap();
    assert_eq!(vm.execute_program(mem).unwrap(),
                0x37);
    assert_eq!(mem[4], 0x37);
}
```

This example illustrates how the Fibonacci number can be computed using the eBPF VM. **fib** is a Rust function that calculates the *nth* Fibonacci number. The function takes five input parameters that map to registers r1-r5. Of the five parameters, only the first parameter is used. The fib function is registered as a helper function with the eBPF VM with id 0. The eBPF program loads the value of n, which is located as a 32-bit integer in the first four bytes of the context memory. The function then computes the Fibonacci number by calling the registered helper function. It stores the result in register r0 and writes it to a 32-bit integer starting at offset 4 in the context memory using the **stxw** instruction.

The above programs assemble a BPF program into bytecode and then run the bytecode using the rbpf VM. It is also possible to JIT compile the program on an x86_64 platform and run it natively.

```
// Here prog is BPF program byte code
let mut vm = rbpf::EbpfVmRaw::new(prog)
                .unwrap();

// JIT-compile the program.
vm.jit_compile().unwrap();

// Then we execute it.
unsafe { vm.execute_program_jit()
            .unwrap(); }
```

Note that for rbpf, if errors occur during the program execution, the JIT-compiled version does not handle it as well as the interpreter, and the program may crash. For this reason, the functions are marked as unsafe.

JIT Compilation (Optional)

Suppose the Linux kernel hosting the eBPF program is compiled with the **CONFIG_BPF_JIT** option. In that case, the eBPF program bytecode is JIT (Just In Time) compiled into native assembly instructions after it has been verified and loaded. Otherwise, when the program is executed, it is run in the eBPF VM, which decodes and executes the eBPF bytecode instructions. The JIT compilation process offers execution performance near natively compiled in-kernel code.

Maps

An eBPF application consists of a user space program and the eBPF binary loaded by the user space program into the Linux kernel using the **bpf** system call. Maps are the message-passing mechanism between the user space program and the eBPF program executed by the eBPF VM in the kernel space. A map is a HashMap/HashTable data structure with a fixed-sized key and values. The type for the key and value of the map must be specified when a map is created. The key and the value are treated as binary blobs, and the user can store any data.

Overview

The aya-rs crate completely abstracts away the complexity of the eBPF map lifecycle, and the eBPF map functionality is very close to that of regular Rust data structures in the eBPF code.

User Space

An eBPF map can be declared in a user space program as follows.

```
// Userspace

use aya::maps::HashMap;
use std::net::Ipv4Addr;

...

let mut blocked_ips: HashMap<_, u32, u8> =
  HashMap::try_from(bpf.map_mut("BLOCKED_IPS")?)?;

let mut blocked_ip = Ipv4Addr::new(192, 168, 0, 1);
blocked_ips.insert(u32::from(blocked_ip), 1, 0)?;

blocked_ip = Ipv4Addr::new(192, 168, 0, 2);
blocked_ips.insert(u32::from(blocked_ip), 1, 0)?;
```

The above code creates a simple map (of type Hash) with the IPv4 address (32-bit) as the key and a flag (8-bit) as the value. This HashMap can be used as a Set data structure to communicate to the eBPF firewall program about which IP traffic needs to be blocked. In the above example, the IP addresses 192.168.0.1 and 192.168.0.2 were added to the map for the kernel eBPF firewall program to block them.

Kernel Space

The eBPF program in the kernel space can then access the hashmap.

```
// eBPF Program (Kernel)
use aya_bpf::{macros::{map}, maps::HashMap};
```

```
....

// Creates the map in the kernel
#[map(name = "BLOCKED_IPS")]
static mut BLOCKED_IPS: HashMap<u32, u8> =
    HashMap::<u32, u8>::with_max_entries(1024, 0);

....

if unsafe {BLOCKED_IPS.get(&src_addr).is_some()} {
   return Ok(xdp_action::XDP_DROP);
}
```

Note the string **BLOCKED_IPS** is used to tie a map between the userspace and the eBPF program.

Map Types

eBPF offers several different types of maps, and these types are defined in the **bpf_map_type** enum in the Linux kernel source code. The aya-rs crate has added support for a subset of them. The following sections provide information about the different map types implemented by the aya-rs crate.

Hash Map

This map resembles a HashTable/HashMap, and its use was covered in the overview section of this chapter.

Array Map

This map resembles an array. All array elements are pre-allocated and zero-initialized at init time. The size of the array is defined in the eBPF program using the bpf_map_def:: max_entries field.

```
// User space program

use aya::maps::Array;

...

let mut array: Array<_, u32> =
    Array::try_from(bpf.map_mut("ARRAY").unwrap())?;
...

// After the eBPF program has been loaded
array.set(1, 42, 0)?;
```

```
// eBPF Program (Kernel)

use aya_bpf::{macros::map, maps::Array};

...

#[map(name = "ARRAY")]
static mut array: Array<u32> =
    Array::with_max_entries(1024, 0);

...
```

```
let val = array.get(&1, 0)?;
assert_eq!(val, 42);
```

The above code defines a 32-bit unsigned integer array of size 1024 elements in the eBPF program. The program has an assertion to make sure the value added by the user space program is reflected in the kernel eBPF program.

Per-CPU Hash Map

This type of map is a specialized version of the regular Hash Map. When this type of map is allocated, each CPU sees its isolated version of the map, which makes it much more efficient for high-performant lookups and aggregations.

Per-CPU Array Map

This type of map is a specialized version of the regular Array Map. When this type of map is allocated, each CPU sees its isolated version of the map, which makes it much more efficient for high-performant lookups and aggregations.

Perf Event Array Map

This map stores **perf_events** data in a real-time buffer ring that communicates between the BPF program and the userspace programs. These maps are designed to forward the events that the kernel's tracing tools emit to user space programs for further processing. This is one of the most interesting maps and is the base of many observability tools.

For example, we would like to log the source and destination addresses of all the IPv4 packets. To do this, we first need to define a C-based struct to hold information about each event.

```
#[repr(C)]
pub struct PacketLog {
    pub src_addr: u32,
    pub dst_addr: u32,
}
```

```
// eBPF program
use aya_bpf::{macros::map, maps::PerfEventArray};

#[map(name = "EVENTS")]
static mut EVENTS: PerfEventArray<PacketLog> =
    PerfEventArray::<PacketLog>::with_max_entries(1024,
        0);
...

let source = u32::from_be(*ptr_at(&ctx,
  ETH_HDR_LEN + offset_of!(iphdr, saddr))?);

let dst = u32::from_be(*ptr_at(&ctx,
  ETH_HDR_LEN + offset_of!(iphdr, daddr))?);

let log_entry = PacketLog {
  src_addr: source,
  dst_addr: dst
};
EVENTS.output(&ctx, &log_entry, 0);
...
```

The user space program can then read these events from the perf event array, as shown below.

```
// Userspace

use aya::maps::perf::AsyncPerfEventArray;

...
let mut perf_array =
  AsyncPerfEventArray::try_from(
        bpf.map_mut("EVENTS")?)?;

 loop {
  let events = buf.read_events(&mut buffers).await.
      unwrap();
  for i in 0..events.read {
    let buf = &mut buffers[i];
    let ptr = buf.as_ptr() as *const PacketLog;
    let data = unsafe { ptr.read_unaligned() };
    let src_addr = net::Ipv4Addr::from(data.src_addr);
    let dst_addr = net::Ipv4Addr::from(data.dst_addr);
    println!("LOG: SRC {} DST: {}"
      src_addr, dst_addr);
    }
  }
```

Stack

A stack map represents a LIFO (Last In, First Out) data structure.

```
// Userspace

use aya::maps::Stack;
```

```
...

let mut stack: Stack<_, u32> =
    Stack::try_from(bpf.map_mut("STACK").unwrap())?;

...

// After the eBPF program has been loaded
let addr = stack.pop(0)?;
```

```
// eBPF program

use aya_bpf::{macros::map, maps::Array};

...

#[map(name = "STACK")]
static mut stack: Stack<u32> =
    Stack::with_max_entries(1024, 0);

...

let mut addr:u32;

...
unsafe { STACK.push(&addr, 0).unwrap() };
```

Queue

Queue map represents a FIFO (First In, First Out) data structure. The API for this map is the same as that of the Stack

Map, except the **pop** function pops the first element instead of the last element added to the map.

LPM Trie

LPM (Longest Prefix Match) Trie is a data structure that uses the LPM algorithm to look up elements in the map. The LPM algorithm selects the element in a tree that matches with the longest lookup key from any other match in the tree. This algorithm is used in routers and other devices that keep traffic forwarding tables to check IP addresses.

```
// Userspace
use aya::maps::lpm_trie::{LpmTrie, Key};
...

let mut routes: LpmTrie<_, u32, u8> =
     LpmTrie::try_from(bpf.map_mut("ROUTES")?)?;

let route_1 =
  Key::new(u32::from(Ipv4Addr::new(10,0,0,0)),8);

routes.insert(&route_1, 1, 0)?;

let route_2 =
  Key::new(u32::from(Ipv4Addr::new(10,11,0,0)),16);

routes.insert(&route_2, 2, 0)?;
```

In this example, a user space program uses the LPM trie to store information about routes for two different subnets.

```
// eBPF program
```

```
use aya_bpf::maps::lpm_trie::{LpmTrie, Key};

#[map(name = "ROUTES")]
static mut routes: LpmTrie<u32, u8> =
  LpmTrie::with_max_entries(1024, 0);

...

let route = unsafe { routes.get(&dst_ip) };
```

The eBPF uses the routing information stored by the user space program to determine the route for the IP packet.

Socket Map/Hash

Sockmap and SockHash are maps used to store kernel-opened sockets. Sockmap is backed by an array and enforces keys to be four bytes. This works well for many use cases. However, this has become limiting in more prominent use cases where a Sockhash is more appropriate since it enables looking up the socket using the 5-tuple (protocol, source IP, source port, destination IP, destination port) lookup key.

Program Array Map

The eBPF verifier limits the number of eBPF instructions that can make up an eBPF program. To overcome this limit, a kernel functionality implemented using eBPF can be split into multiple eBPF programs (a maximum of 32 eBPF programs can be chained together). The program array map (BPF_MAP_TYPE_PROG_ARRAY) enables the control to jump from one eBPF program to another using the **bpf_tail_call** helper function. The program array map will be explained in detail using an example in the book’s second part.

Program Types

An eBPF program type determines

- The subsystem in the Linux kernel that the eBPF program is attached to.
- The in-kernel helper functions that the program can call.

- The object type of the argument (context) passed to the program.

The eBPF program types vary as per the Linux kernel version. To get the complete list of the eBPF program types supported by your Linux distribution, use the command `bpftool feature` (please refer to the tools chapter). Following are some of the most interesting eBPF program types.

Type	Functionality
BPF_PROG_TYPE_SOCKET_FILTER	A network packet filter
BPF_PROG_TYPE_KPROBE	Determine whether a kprobe should fire or not

Type	Functionality
BPF_PROG_TYPE_XDP	A network packet filter run from the device-driver receive path
BPF_PROG_TYPE_PERF_EVENT	Determine whether a perf event handler should fire or not
BPF_PROG_TYPE_CGROUP_SKB	Traffic control for cgroup processes
BPF_PROG_TYPE_CGROUP_SOCK	Cgroup process socket control
BPF_PROG_TYPE_CGROUP_DEVICE	Determine if a device operation should be permitted or not
BPF_PROG_TYPE_CGROUP_SOCK_ADDR	Useful to implement socket-based NAT (network address translation)
BPF_PROG_TYPE_SOCK_OPS	A program for setting socket parameters
BPF_PROG_TYPE_SK_SKB	A network packet filter for forwarding packets between sockets
BPF_PROG_TYPE_SK_MSG	Egress packet filter

The second part of the book with go through multiple eBPF Rust programs that will cover some of these program types.