ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

BERKELEY PACKET FILTER: theory, practice and perspectives

Relatore:
Chiar.mo Prof.
RENZO DAVOLI

Presentata da:
MICHELE DI STEFANO

Sessione II
Anno Accademico 2019/2020

All operating systems sucks,

but Linux just sucks less.

Linus Torvalds

Introduction

Initially packet filtering mechanism in many Unix versions was implemented in the userspace, meaning that each packet was copied from the kernel-space to the user-space before being filtered. This approach resulted to be inefficient compared with the performance shown by the introduction of the BSD classic Berkeley Packet Filter (cBPF). cBPF consists of an assembly language for a virtual machine which resides inside the kernel. The assembly language can be used to write filters that are loaded in to the kernel from the user-space allowing the filtering to happen in the kernel-space without copying each packet as before. cBPF use is not limited to just the networking domain but it is also applied as a mechanism to filter system calls through seccomp mode. Seccomp allows the programmer to arbitrary select which system calls to permit, representing an useful mechanism to implement sandboxing.

In order to increase the number of use cases and to update the architecture accordingly to new advancements in modern processors, the virtual machine has been rewritten and new features have been added. The result is extended Berkeley Packet Filter (eBPF) which consists of a richer assembly, more program types, maps to store key/value pairs and more components. Currently eBPF (or just BPF) is under continuous development and its capacities are evolving, although the main uses are networking and tracing. It is worth to note that seccomp support has not been introduced despite the fact that recently two patches have been proposed.

The goal of this thesis is to provide an overview on cBPF and eBPF use cases, their components, how to use them introducing basic examples available on a dedicated GitHub repository and finally to describe an attempt to introduce seccomp to eBPF and why it could be useful.

Contents

List of Figures

Chapter 1

Classic BPF

Berkeley Packet Filter was introduced in the BSD operative system as a mean to filter packets as early as possible, avoiding the need to copy packets from the kernel-space to the user-space, before filtering them through user-space network monitoring tools. This approach greatly improved packet filtering performance compared with the existing ones. BPF allows to simply attach a filter to any socket from an user-space program [1].
The Linux version of the Berkeley Packet Filter (introduced in Linux 2.1.75) [2], referred as Linux Socket Filtering (LSF), although differs from the BSD version (for example there is no need to interact with devices), intends the same mechanism, in fact LSF filters use the same filter code of the BSD version. Therefore is adviced to consult the BSD bpf man page when writing filters [35].
During the years BPF has been rewritten adding different components and functionalities, then the original BPF will be referred as classic BPF (cBPF) to distinguish it from the new implementation, the extended BPF (eBPF).

Figure 1.1 shows how cBPF components work and interact with the system. Normally every time a packet arrives, the link-level device driver sends it to the protocol stack. With cBPF instead the packet is sent to user-defined process' filter which decides the number of bytes of the packet to accept that

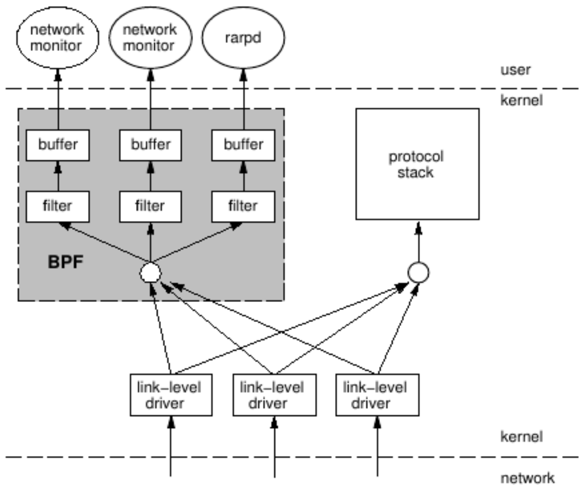

Figure 1.1: cBPF overview [1].

are consequently copied in to the buffer relative to that particular filter. Then if the packet destination is that specific host the packet is processed through the network protocol stack [1].

1.1 Filter model

The filters efficiency is a major point to consider because most applications which capture packets discard more packets than the ones they accept. cBPF uses directed acyclic control flow graph (CFG) filter model that can be implemented with a register machine code [1]. Figure 1.2 illustrates a filter which accepts IP, ARP, RARP packets that have *src* or *dst* field equals to *foo*.

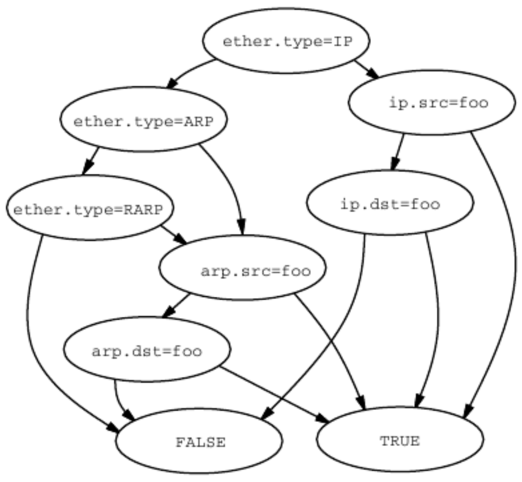

Figure 1.2: CFG example [1].

The filter is a boolean function which can be evaluated to true or false depending on the traversed path. For example, suppose an Ethernet frame which contains an IP packet arrives but the packet is not sent from *foo*, then the filter checks the next predicate - if the packet's destination is *foo* - if also it is not true then the filter is evaluated to false.

1.2 Architecture

An accumulator machine model is used to implement the BPF pseudo-machine. The machine consists of [1, 35]:

- A: a 32 bit accumulator
- X: a 32 bit index register
- M[ ] 16x32 bit misc registers (scratch memory store)
- an implicit program counter

The instructions that can be performed are: load, alu, branch, return and miscellaneous instructions. The format of each instruction is shown in Figure 1.3, where *opcode* encodes a particular instruction, *jt* and *jf* are conditional

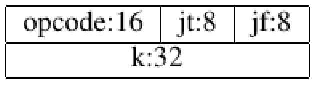

Figure 1.3: Instruction format [1].

jumps offsets and k contains a value whose semantic depends on the instruction type. The semantic of each instruction is described in the BSD bpf man page [33]. The return instructions indicate the number of bytes of the packet to be accepted, then 0 bytes means discard the packet [1].

Suppose the link level header format is Ethernet for example, the instructions reported in the Listing 1.1

```
(000) ldh      [12]
(001) jeq      #0x800     jt 2   jf 3
(002) ret      #262144
(003) ret      #0
```

Listing 1.1: cBPF IPv4

load in to the accumulator half word starting 12 bytes from the beginning of the ethernet header (see section C.6 of Appendix C for a reference to the Ethernet frame header), then compare the stored value with `0x800` (IPv4) and if the result of the comparison is true jump to the instruction `2` which returns a number of bytes great enough to include the entire packet, otherwise jump to the instruction `3` which returns `0` bytes of the packet (e.g. discard the packet). These instructions, using human readable macros in C language, can be rewritten as in the Listing 1.2

```
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x800, 0, 1),
BPF_STMT(BPF_RET+BPF_K, 0x40000),
BPF_STMT(BPF_RET+BPF_K, 0),
```

Listing 1.2: IPv4 with macro

that expands in

```
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 1, 0x00000800 },
{ 0x6, 0, 0, 0x00040000 },
{ 0x6, 0, 0, 0x00000000 },
```

Listing 1.3: IPv4 expanded

If the filter shown in the Listing 1.2 or in the Listing 1.3 is attached to a packet socket it has the effect to accept only IPv4 packets. More examples and details about writing cBPF filters, and how `tcpdump` uses cBPF are available in the relative appendix (see section A.1 of Appendix A).

The CFG of the previously shown filter in Figure 1.2 illustrates how each node is a predicate that needs to be evaluated. The resulting value is stored in the accumulator and determines which branch to choose next. Using BPF instructions the filter can be rewritten as reported in Figure 1.4.

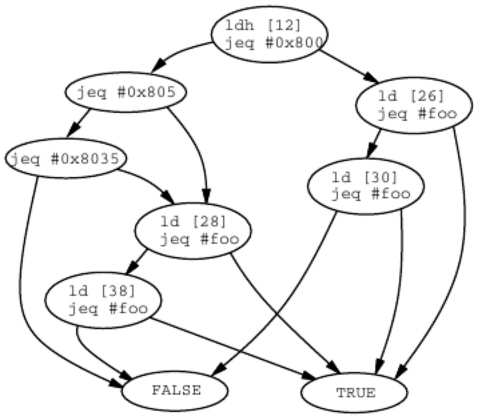

Figure 1.4: CFG filter with cBPF instructions [1].

As in the description of the previous higher-level example (Figure 1.2), suppose that an Ethernet frame which contains an IP packet arrives. The 2 bytes of the ethernet type field are loaded in the accumulator and the value is compared with the constant for IPv4 packets. As the first predicate is evaluated as true, 4 bytes (word) starting 26 bytes from the beginning of the ethernet header are loaded in the accumulator. This operation corresponds to load the source address field of the IPv4 header in the accumulator: 14 bytes of Ethernet frame header plus 12 bytes to reach the beginning of the IPv4 header source field (see section C.2 of Appendix C). If the address doesn't match with *foo* then the content of the destination address field of the IPv4 header is loaded in the accumulator and compared with *foo*. This last comparison determines the boolean value of the filter.

1.3 Internals

Injecting user-space code into the kernel can be dangerous because it raises the possibility of write/read kernel memory or crash/hang the kernel, then a verifier needs to check the user's filter code. The verifier for cBPF programs could be found in `net/core/filter.c` file. It was implemented by the `sk_chk_filter()` function which checked that the filter contained only

valid instructions, a return statement, no backward jumps, and no jumps or reference out of range. A maxium of 4096 instructions was allowed [34].
The verifier consists in a `for` loop which iterates through the filter's instructions and verifies if each of them is legal (Listing 1.4).

```
/* check the filter code now */
for (pc = 0; pc < flen; pc++) {
    ...
}
```

Listing 1.4: verifier_for_loop

As an example, in the Listing 1.5 is shown the test performed by the verifier on conditional jumps. It consists of verifying if adding the current jump offsets (*ftest->jt* and *ftest->jf*) to the program counter (*pc)* produces a value which is equal or greater than the filter length (*flen*). If the condition is true the function returns `-EINVAL`, otherwise it proceeds with the checks.

```
...
case BPF_JMP|BPF_JGT|BPF_X:
case BPF_JMP|BPF_JSET|BPF_K:
case BPF_JMP|BPF_JSET|BPF_X:
/* for conditionals both must be safe */
if (pc + ftest->jt + 1 >= flen || pc + ftest->jf + 1 >= flen)
   return -EINVAL
break;
```

Listing 1.5: sk_chk_filter

The interpreter was implemented by the `sk_run_filter()` function which consisted in a big switch case that performed decoding of the filter instructions and application of them to the socket buffer.
An extract of the code is reported in the Listing 1.6.

```
/*
 * Process array of filter instructions.
 */
for (pc = 0; pc < flen; pc++) {
        const struct sock_filter *fentry = &filter[pc];
        u32 f_k = fentry->k;

        switch (fentry->code) {
        case BPF_ALU|BPF_ADD|BPF_X:
                A += X;
                continue;
        case BPF_ALU|BPF_ADD|BPF_K:
                A += f_k;
                continue;
        ...
```

Listing 1.6: sk_run_filter

The program counter starts from the first instruction and executes the proper operation based on the current entry code.
Currently cBPF instructions are internally converted in eBPF instructions which are closer to the underlying architecture, therefore if a JIT compiler for that architecture is supported it is possible to improve the filter performance [35].

1.4 Application

1.4.1 tcpdump

The main use of cBPF is through `tcpdump` which is a network monitoring tool. `tcpdump` permits to use an higher level syntax to write packet filtering rules. An example of a command to filter all ICMP echo-reply packets on the loopback interface and its output (when on another terminal `ping -c1 localhost` is executed) is illustrated in the Listing 1.7

```
$ sudo tcpdump -nni lo icmp[icmptype] == 0
tcpdump: verbose output suppressed, use -v or -vv for
      full protocol decode
listening on lo, link-type EN10MB (Ethernet),
      capture size 262144 bytes
10:25:07.054584 IP 127.0.0.1 > 127.0.0.1: ICMP echo reply,
      id 4488, seq 1, length 64
```

Listing 1.7: tcpdump echo-reply

As shown in the Listing 1.7, the ouptut reports only the information about the echo-reply and not about the echo request. These high-level rules are transparently compiled to cBPF instruction through `libpcap` (see section A.1.2.2 of Appendix A for an example on how to show cBPF instructions from `tcpdump`).

1.4.2 seccomp

cBPF is not used only for packet filtering. The other most known use is SECure COMPuting (seccomp BPF).
Seccomp is a mean to limit the kernel surface exposed to an application by reducing the set of system calls that a user-space process can use [8]. Seccomp was first introduced by Andrea Arcangeli in the 2005 [4] as a mechanism to implement the security required for his *cpushare* project, which had the goal to let users sell theirs CPU's idle cycles to others, therefore the consequent need to control arbitrary code execution [3].
A process running in seccomp mode was allowed to perform just 4 syscalls: `read(2)`, `write(2)`, `_exit(2)`, `sigreturn(2)`. As reported by Google developer Markus Gutschke, that was a main downside to use seccomp for building a Chrome sandbox [5]. The discussion about the suggested patches

to extend seccomp lasted more than 2 years [6]. At the end, in the 2011, Will Drewry proposed a patch which extended seccomp reusing cBPF, then allowing to write personalized cBPF programs that instead of analyze packets would evaluate every system call (based on system call number and arguments) before to be executed [7]. The former and restrictive seccomp mode is named *strict mode*, while the latter is referred as *filter mode*.
It is adviced that developers using seccomp should keep attention to maintain filters because library updates could break code. For example, since glibc 2.26, the wrapper `open(2)` doesn't invoke directly kernel's `open` system call but it's a special case of `openat(2)` [9, 12]. It should be noted that currently seccomp can use only cBPF language and not eBPF [66].

Chapter 2

Extended BPF

As already mentioned in the section 1.3, cBPF instructions are converted in eBPF instructions. Extended BPF has been also referred as internal BPF because the user-space was not exposed directly to eBPF and the translation from cBPF to eBPF was (and currently is) transparent to the user [35, 11]. At present, it is also known just as BPF. eBPF was introduced (first in the kernel 3.15) because the cBPF virtual machine architecture was far from the modern processors architecture which started to use 64 bit registers and more instructions [56], and also with the intend to add new features beyond packets/system calls filtering mechanism.
Major changes were made to the older virtual machine architecture bringing several new functionalities with the effect of making the kernel programmable while maintaining the user/kernel space separation. As an example of the modifications, the instruction set and the registers were expanded.
It's worthwhile to note that eBPF does not consist only just of a new instruction set but also provides different infrastructures which extend use cases. The number of events to which attach programs increased, therefore the application of eBPF is not limited to the kernel networking subsystem and allows kernel/user space communication through data structures (maps), then introducing the concept of state.

2.1 Architecture

This section describes the new virtual machine architecture.

2.1.1 Registers and stack

A 512 byte fixed size stack is available. The number and the width of registers increased respectively, from 2 to 11 (R0-R10) and from 32 bit to 64 bit. As a consequence eBPF registers map one to one to x86 hardware registers as illustrated in the Listing 2.1.

```
R0 − rax
R1 − rdi
R2 − rsi
R3 − rdx
R4 − rcx
R5 − r8
R6 − rbx
R7 − r13
R8 − r14
R9 − r15
R10 − rbp
```

Listing 2.1: registers mapping [35]

`R0` stores the return value of an helper (in kernel) function or the return value of the eBPF program. `R1-R5` contain the arguments for the helper function called by the eBPF program. `R6-R9` are the callee saved register and they are preserved by the called helper function. The content of these registers is moved to the stack (spill) or the content of the stack is moved to them (fill). The aforementioned registers are also used for eBPF to eBPF calls. `R10` is the read only frame pointer for the stack [35, 13].

2.1.2 Instruction set

The instruction set is a mix of real CPUs instructions (mostly x86) with two operands specified, like on real architectures. Also a bpf call instruction was added to call in kernel functions. Each eBFP program loaded as normal user can have at maximum 4096 instructions, instead eBPF programs loaded as root can have up to 1 million instructions [21]. The calling convention has been defined analyzing x86/arm64/risc calling conventions and in a way to avoid extra copy in calls.
As a consequence of these changes not only the registers but also the eBPF instructions map one to one into x86 real ones [2].
Internally, a cBPF instruction is converted in the eBPF instruction format shown in the Listing 2.2

```
op:8, dst_reg:4, src_reg:4, off:16, imm:32
```

Listing 2.2: eBPF instruction format

Where *op* determines the operation to be executed and it's further subdivided as reported in the Listing 2.3

```
code:4 source:1 class:3
```

Listing 2.3: op code

class specifies the instruction class (`BPF_LD, BPF_ALU`, etc.), *code* determines an operation code within that class, for instance if the class is `BPF_ALU` *code* can be `BPF_ADD, BPF_MUL`, etc. and *source* specifies if the source operand is an immediate value or a register [35].
dst_reg and *src_reg* indicate which registers to use for the operation. *off* is used for operations that require a relative offset. *imm* contains an immediate value.
To make the conversion from cBPF instructions to eBPF instructions easier the cBPF encoding is reused. These architectural choices have led to the

implementation of a Just-In-Time compiler from eBPF to native machine code which in turn improved the performance [56]. JIT compiler is available for the most common architectures. If the JIT compiler is not available for an architecture it is still possible to execute eBPF programs using the interpreter as a fallback (if it is enabled).

2.2 Program type

If with cBPF is possible to write only two kind of programs: socket filters and system call filters, eBPF introduced several program types which are still evolving. The main categories are networking and tracing. Listing 2.4 reports an extract of the content of **enum** bpf_prog_type, defined in `uapi/linux/bpf.h`, which enumerates all the currently available program types (more than 20 at time of writing).

```
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,
        ...
};
```

Listing 2.4: bpf_prog_type

When working with eBPF program it's important to comprehend the following information about program types: which use case the program satisfies; how to attach the eBPF program for that particular program type; what context (data) is passed as an input to the program; which event triggers the attached program [59]. To better illustrate how the program type affects these information, 3 program types (highlighted in bold in the Listing 2.4) are now described in more details.

2.2.1 BPF_PROG_TYPE_SOCKET_FILTER

As the original classic BPF use case, socket filter type applies to network traffic filtering, then allows discarding or trimming of packets based on the return value. eBPF programs of this type are attached to a socket using `setsockopt(2)` with `SO_ATTACH_BPF` (see Listing B.26). The context is represented by a data structure (`__sk_buff`) that is a mirror of the in kernel data structure (`sk_buff`) used to represents packets metadata (see the description of Listing B.6). Every time a packet is received on that socket the program is executed.

2.2.2 BPF_PROG_TYPE_XDP

XDP stands for eXpress DataPath. The goal of XDP is to improve packet processing performance by providing a hook closer to the hardware (at the driver level), accessing a packet before the operative system creates metadata stored in `sk_buff`. Return values of a XDP program can indicate different actions: drop the packet (`XDP_DROP`), pass the packet to the networking stack (`XDP_PASS`), send the packet out the same interface it was received (`XDP_TX`), redirect the packet to userspace or to another interface (`XDP_REDIRECT`) [14]. The XDP program can be attached to a network interface through a netlink socket message. The context of XDP programs is a lightweight data structure (**struct** xdp_md). XDP program runs every time a packet arrives to the NIC.
It's worth mentioning that XDP can operates in 3 modes: *driver* (native) mode that is, the eBPF program is executed at the driver level (the device needs XDP support); *generic* mode which is used as fallback mode for net-

work devices that don't support native XDP. It's slower compared to the driver mode because works at an higher level in the stack; *hardware offload* mode which allows to run the XDP program directly on the NIC bringing higher speed than native mode. Just one mode at time can be used [13].

2.2.3 BPF_PROG_TYPE_TRACEPOINT

This program type allows to instrument kernel code. To attach an eBPF program first a perf event is opened with `perf_event_open(2)`, then through `ioctl(2)` the returned file descriptor is used to enable the associated individual event or event group and to attach the eBPF program to the tracepoint event. The context definition depends on the specific tracepoint. The eBPF program is executed every time the tracepoint is hit (see section B.1.4 and the example reported in the Listing B.1.10 for more details).

2.3 Helper functions

As already mentioned in section 2.1.2 from an eBPF program it is possible to call helper functions. Program type is what determines which subset of in kernel functions can be called. Helper functions are called from within eBPF programs in order to interact with the system, to operate on the data passed as context or to interact with maps. Calling these functions don't introduce an overhead [15]. Then in order to be able to write eBPF programs is required to know which helpers can be called from a specific program type. A script to generate the man page which documents helper functions has been added to the kernel source code [17].
The prototypes are declared in the kernel source code within `tools/testing/selftests/bpf/bpf_helpers.h`. To retrieve the list of the helpers supported by a program type is possible to execute from within the kernel source code tree the command reported in the Listing 2.5

```
$ grep -R 'func_proto(enum bpf_func_id func_id' kernel/  \
        net/ drivers/
```

Listing 2.5: grep func_proto

For instance, the function definition (see `net/core/filter.c`) shown in the Listing 2.6 illustrates that socket filter programs can use base helper functions and four more.

```
static const struct bpf_func_proto *
    sk_filter_func_proto(enum bpf_func_id func_id ,
    const struct bpf_prog *prog)
{
        switch (func_id) {
        case BPF_FUNC_skb_load_bytes:
                return &bpf_skb_load_bytes_proto;
        case BPF_FUNC_skb_load_bytes_relative:
                return &bpf_skb_load_bytes_relative_proto;
        case BPF_FUNC_get_socket_cookie:
                return &bpf_get_socket_cookie_proto;
        case BPF_FUNC_get_socket_uid:
                return &bpf_get_socket_uid_proto;
        default:
                return bpf_base_func_proto(func_id);
        }
}
```

Listing 2.6: sk_filter_func_proto

The section 2.8.1 explains how the verifier uses this information to check if an helper call is valid.

2.4 Maps

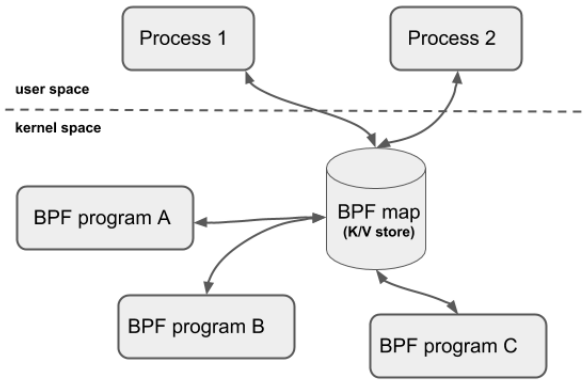

Figure 2.1: Maps [13].

Another main component of the eBPF infrastructure is represented by maps, which are data structures that store key-value pairs (with an arbitrary structure chosen by the user) and allow the communication between user space program and eBPF program. Maps permit to keep state between different program executions. Moreover, different eBPF programs can access the same map [51, 18].
Using the proper helpers it's possible to create/update or delete a map and to lookup for an element. Several types of maps (more than 20 at the time of writing) are available and enumerated by `enum bpf_map_type`. An extract is reported in the Listing 2.7

```
enum bpf_map_type {
        BPF_MAP_TYPE_HASH,
        BPF_MAP_TYPE_ARRAY,
        BPF_MAP_TYPE_PROG_ARRAY,
        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        ...
};
```

Listing 2.7: enum bpf_map_type

The `bpf(2)` man page describes the main types in more details. For instance, array maps don't support value deletion and the key size can be only 4 bytes. In this case the key is the index of the array. Hash maps keys instead can have a different size and values deletion is supported. It's worth to mention `BPF_MAP_TYPE_PROG_ARRAY` which stores file descriptors of other eBPF programs that can be called from the current one (see section 2.5). Maps are defined in the eBPF program source code as a separated section. See the section B.1.9 of Appendix B for an example of array map usage.

2.5 Tail calls

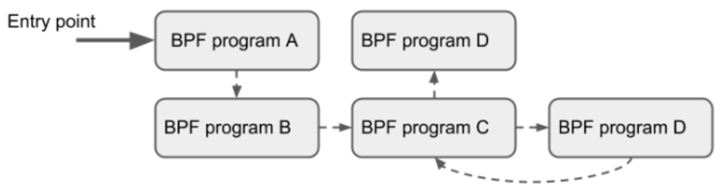

Figure 2.2: Tail calls [13].

As mentioned in the Maps section an eBPF program can call another eBPF program using a proper map type. To implement this functionality another component is required. In fact `bpf_tail_call()` helper function has been introduced to make it possible. As input this helper takes the context, a reference to the program array map and the index where jump to. This mechanism is called tail call because the current stack frame is reused to execute another eBPF program avoiding to add a new one. To prevent loops the limit of tail calls number is 32 [16]. Main use cases are simplify complicated programs, dynamically modify an eBPF program behaviour and dispatch into other programs.

2.6 eBPF to eBPF calls

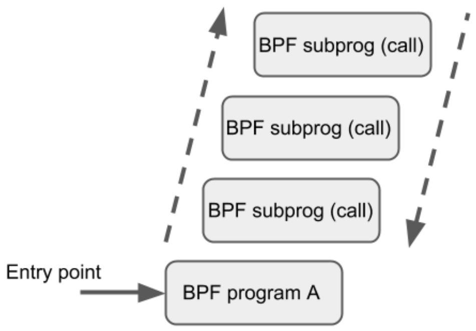

Figure 2.3: eBPF to eBPF Calls [13].

Initially there was no concept of function in eBPF, then no support was provided. As a consequence reusable code was declared with the `always_inline` attribute so each function with that attribute was copied many times with the effect of increasing the object code size. The support for eBPF to eBPF calls has made possible to rewrite functions without the `always_inline` attribute.

2.7 Object pinning

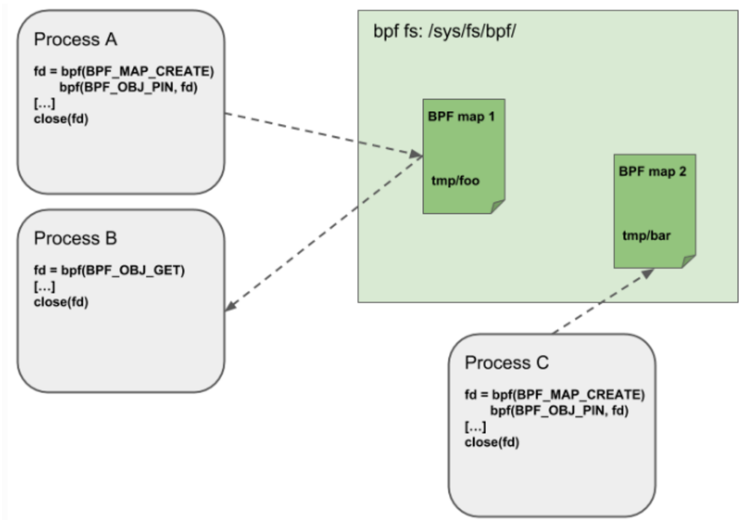

Figure 2.4: Object Pinning [13].

eBPF objects are accessed through file descriptors and have a reference counter which until is strictly greater than 0 guarantees that the kernel keeps the object alive. For example if a map is created by an user space process the kernel increases maps reference counter to 1 and sends the associated file descriptor to the user space process but, if the process which created the map terminates, as a consequence the file descriptor is closed, the reference counter is decreased to 0 and the memory is freed.

Instead the eBPF program reference counter is increased when the program is attached to a hook. When a user space process creates a program and maps, and the program is attached to a hook, the process can exit because program's reference counter will be greater than 0 and, since the maps are used by the program their reference count is also set to 1 and the kernel will keep program and maps alive [20].

Initially the design didn't provide a mechanism for eBPF programs and maps to persist after the termination of the process that created them. Kernel 4.4 introduced this feature through a kernel virtual file system (mounted at `sys/fs/bpf`). Files within this file system can have arbitrary names. They represent eBPF persistent objects and then can be accessed between different program invokations. Indeed a pinned object will have an increase in its reference counter, as a consequence the object will be kept alive also if the BPF program is not attached or the BPF map is not used by any program. New commands for the `bpf(2)` system call have been introduced: `BPF_OBJ_PIN` to pin eBPF objects and `BPF_OBJ_GET` to retrieve their file descriptor [19]. An use case can be found in networking, for instance an eBPF program handles packet processing and stores information in a map which is pinned and periodically the admin checks the map's content getting the file descriptor associated with the map [20].

2.8 Internals

2.8.1 Verifier

Also with eBPF it is critical to guarantee that an eBPF program terminates and doesn't cause damage to the kernel (e.g. access arbitrary kernel memory). The code that implements the verifier can be found within `/kernel/bpf/verifier.c`. The function `bpf_check()` performs a static analysis of the eBPF bytecode.
The verification process is performed in two steps: in the first step the program's control flow graph is build and a depth-first-search is executed to check if it is a Directed Acyclic Graph (DAG). In this phase the programs are rejected if they: are too big; contain forbidden loops (program contains back-edge and is loaded by an unprivileged user or program contains back-edge performed using a call); call functions which are not eBPF helpers or other eBPF programs; present unreachable instructions; have out of range jumps. During the second step the verifier simulates the execution of all the possible branches of the program starting from the 1st instruction and checks the state of the stack and the registers after each instruction.
Moreover, if the user that loads the eBPF program doesn't have `CAP_SYS_ADMIN` the verifier is run with secure mode and pointer arithmetic is forbidden. If the verifier finds that there is an attempt to read uninitialized register the program is rejected. The verifier also check if the context access is valid and if the helper functions called by the eBPF program are a subset of in-kernel functions associated with that specific program type [56].

The verifier code is complex and consists of about 10.000 lines of code. Just to have an insight on how it performs some checks an example on how helper calls verification is executed is now described. The information about the helpers that a particular program type can call are listed in the program's function prototype that is defined in the form of a *type*`_func_proto` function

(see Listing 2.6 for socket filter program type example). A reference to this function is stored in a field (`get_func_proto`) of a data structure used by the verifier (see Listing 2.8) to check if the eBPF program calls an allowed helper.

```
const struct bpf_verifier_ops sk_filter_verifier_ops = {
        .get_func_proto         = sk_filter_func_proto,
        ...
};
```

Listing 2.8: sk_filter_verifier_ops

If the verifier in the `do_check()` function, called inside `bpf_check()`, finds a `BPF_CALL` opcode it first checks if it is handling a call to another BPF function or a call to an helper function (see Listing 2.9).

```
...
if (insn->src_reg == BPF_PSEUDO_CALL)
        err = check_func_call(env, insn, &env->insn_idx);
else
        err = check_helper_call(env, insn->imm, env->insn_idx);
...
```

Listing 2.9: call check

In the latter case the function `check_helper_call()` implements the verification on the allowed helpers as reported in the Listing 2.10

```
...
if (env->ops->get_func_proto)
   fn = env->ops->get_func_proto(func_id, env->prog);
if (!fn) {
        verbose(env, "unknown func %s#%d\n",
            func_id_name(func_id), func_id);
```

```
          return -EINVAL;
}
...
```

Listing 2.10: check_helper_call

If the value of `func_id` maps to an helper function present within `sk_filter_func_proto` definition, the verifier subsequently checks if the type of the arguments passed to the helper and stored in the relative registers satisfy the constraints required by that helper (see Listing 2.11), otherwise returns invalid argument error.

```
/* check args */
err = check_func_arg(env, BPF_REG_1, fn->arg1_type, &meta);
if (err)
   return err;
err = check_func_arg(env, BPF_REG_2, fn->arg2_type, &meta);
...
```

Listing 2.11: check helper arguments

The contraints that the arguments need to satisfy are stored within **struct** bpf_func_proto. Suppose that a socket filter eBPF program calls `bpf_skb_load_bytes()` helper function, the type required by its arguments is shown in the Listing 2.12

```
static const struct bpf_func_proto bpf_skb_load_bytes_proto = {
          .func          = bpf_skb_load_bytes,
          .gpl_only      = false,
          .ret_type      = RET_INTEGER,
          .arg1_type     = ARG_PTR_TO_CTX,
          .arg2_type     = ARG_ANYTHING,
          ...
};
```

Listing 2.12: bpf_skb_load_bytes_proto

Furthermore, it is worth to note that the goal of the verifier is to check eBPF programs safety to protect the kernel and not to check if they perform their intended function efficiently. Therefore is a programmer's responsibility to not write programs which, for example, can slow down the machine [14].

2.9 How to use an eBPF program

All the components which are required to write an eBPF program have been introduced. The diagram shown in Figure 2.5 illustrates an overview on how these components interact and on how eBPF programs are compiled, loaded into the kernel and executed when a specific event occurs. As reported in the figure eBPF programs are written in restricted C (a subset of C language) because, although is possible to use other high level languages, currently is the standard way to write them and also in the Appendix B, C language and LLVM/Clang compiler are used.

The steps to run an eBPF program are summarized in the following [47]:

1. the user-space code loads the eBPF program into the kernel specifying the program type, which determines accessible kernel's functions
2. the kernel checks if the program is safe through the verifier
3. the kernel, if possible and if it is desirable, JIT-compiles the eBPF bytecode to native machine code, otherwise the program is interpreted
4. the injected code is attached to an event and is executed every time that event occurs
5. the loaded code through the helpers can write data to maps and ring-buffers and the user-space code can read/write from them

All the operations related to eBPF are performed through the `bpf(2)` system call which is described in the section B.1.2 of the Appendix B.

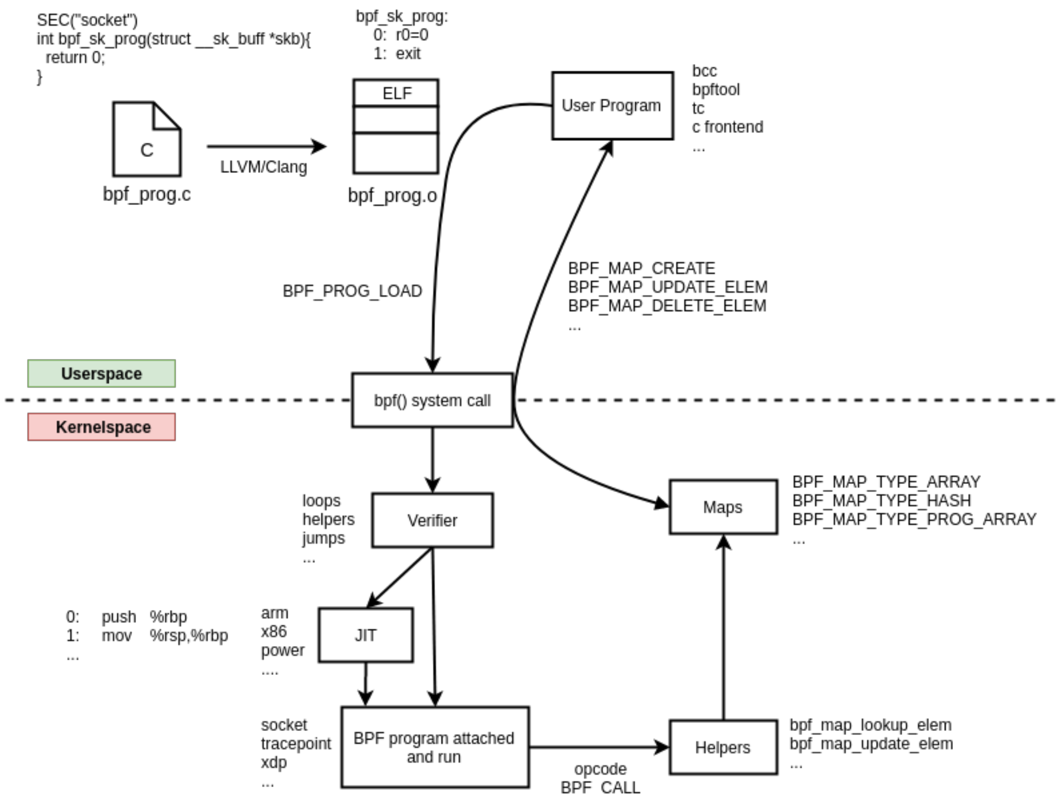

Figure 2.5: eBPF usage overview. Adapted from [22]

2.10 Application

The major applications of eBPF can be summarized in two main groups: networking and tracing.

2.10.1 Networking

In the context of networking eBPF can be used for routing software (XDP has an helper which performs routing table lookups), DoS mitigation (packet filtering) and load balancing [14].

For instance, XDP use is largely increasing. Two examples created by Facebook are represented by Katran, an open source network load balancer [25] and Droplet, a DDoS protection based on eBPF [26].

Another example are the products of Netronome company which has implemented a JIT compiler that compiles eBPF to code directly executable by a SmartNIC. This approach provides different advantages: the use of XDP offload mode moves processing from host to the NIC hardware (e.g. CPU can be used for other tasks); the hardware offload brings an increase in performance; the ability to dynamically load and unload programs prevents the need to restart the system after programs bug correction and adjustments. This is critical for big data centers [24].

Currently there is the will to replace the `iptables` backend with an eBPF based one. This project is `bpfilter` and the purpose is to translate existing iptables rules to eBPF programs while preserving the semantic of the rules. The translation can be complex, therefore is performed by a bpfilter user mode helper which runs in the user space and is invoked by bpfilter kernel module. A proof of concept is already available [27]. This mechanism has been compared to the other existing mechanisms and the results are promising (see Figure 2.6).

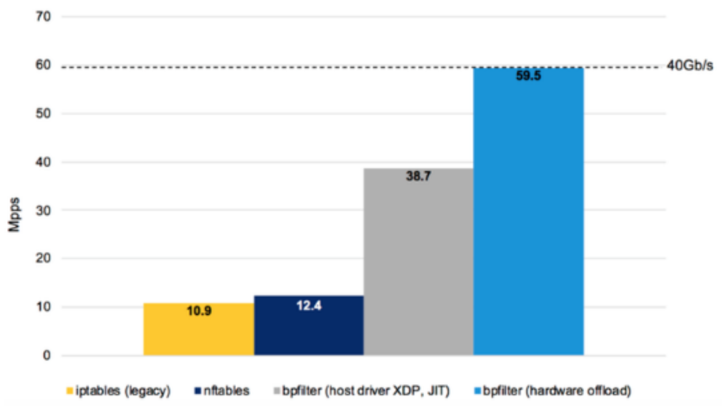

Figure 2.6: bpfilter performance [24].

2.10.2 Tracing

Two frameworks which take advantages of the several pogram types related to the system tracing are BPF Compiler Collection (BCC) and bpftrace.

- BCC includes more than 70 tools for tracing and monitoring and provides a way to write eBPF programs outside the kernel source tree. BCC supports both Python and Lua as languages for the user space code. As an example the tool `opensnoop.py` prints a line for each call to `open(2)`. It can be useful to acquire information about the files used by an application.

```
sudo python ./opensnoop.py
PID    COMM          FD ERR PATH
1602   gsd-color     15   0 /etc/localtime
1602   gsd-color     15   0 /etc/localtime
1384   vminfo         4   0 /var/run/utmp
...
```

Listing 2.13: opensnoop.py

- while BCC is mainly used to write complex tracing tools, bpftrace is a frontend for eBPF tracing features that employs some BCC libraries and is used to write personalized short scripts on the fly. For example, Listing 2.14 shows a command which traces all the files as soon as they are opened. The format of the tracepoint (`tracepoint:syscalls:sys_enter_openat` in this case) is explained in the section B.1.4 of Appendix B where it is described how to define the context for an eBPF program which is attached to a tracepoint.

```
# bpftrace -e 'tracepoint:syscalls:sys_enter_openat \
    { printf("%s %s\n", comm, str(args->filename)); }'
Attaching 1 probe...
snmp-pass /proc/cpuinfo
```

```
snmp-pass /proc/stat
snmpd /proc/net/dev
snmpd /proc/net/if_inet6
^C
```

Listing 2.14: bpftrace

These frameworks internally work using `libbpf` library (see section B.1.8) to manipulate eBPF object files and attaching eBPF programs to kprobes, uprobes, tracepoints and perf events.

Chapter 3

eBPF and seccomp

As mentioned in the section 1.4.2 currently seccomp filters can be written only in cBPF. At the time of writing two patches to extend eBPF usage to seccomp have been proposed: one which adds seccomp type program but doesn't provide maps [28] and one which adds the support for maps and is focused on libseccomp [29].

3.1 Patch

This section shows the main changes reported in the first patch. The purpose is to illustrate how some parts of seccomp are implemented and the main components to modify when adding a new program type to eBPF. This should help to have a better idea on the eBPF architecture and how it is developed. In both patches one of the reason which drove the authors to propose them is to improve seccomp filter performance, although this point has been attacked in both cases.

3.1.1 Discussion

Despite the patch is not merged in the kernel, an attempt to enable eBPF for seccomp was made by Sargun Dhillon in the 2018. The patch doesn't give the possibility to use all the features introduced by eBPF, for instance maps

usage is not allowed. The reason behind this choice, explains the author, is to avoid new complexities around `PR_SET_NO_NEW_PRIVS` (the mechanism which guarantees that processes can't escalate privileges through `execve`).
Kees answered stating that there should be really solid reasons to use eBPF for seccomp. The main reasons reported by Sargun are: user space eBPF tools are better than cBPF user space tools; use eBPF to write seccomp policies for Docker (C instead of json) and a better performance reported using eBPF array instead of normal branches to lookup rules. Regarding this last point it is worth to note that currently seccomp puts the shorter rules (syscall only) at the top while the longer (syscall + arguments) towards the end.
Anyway Kees mantained his opinion that seccomp doesn't need a richer language and proposed to improve libseccomp instead, also because rules in seccomp filters can be written as a balanced tree and ordered by frequency to reach a better performance [28].

3.1.2 Code

A new kernel configuration option is added within `arch/Kconfig` (Listing 3.1). The new option can be enabled only if seccomp filter and bpf syscall options are already selected.

```
config SECCOMP_FILTER_EXTENDED
  bool "Extended BPF seccomp filters"
  depends on SECCOMP_FILTER && BPF_SYSCALL
  help
   Enables seccomp filters to be written in eBPF, as opposed
   to just cBPF filters.
```

Listing 3.1: seccomp kernel configuration

If the new configuration option is selected then within
`include/linux/bpf_types.h` is added the macro (Listing 3.2) which defines

struct bpf_verifier_ops (needed by the verifier) and **struct** bpf_prog_ops data structures for seccomp (`seccomp_verifier_ops` and `seccomp_prog_ops`).

```
#ifdef CONFIG_SECCOMP_FILTER_EXTENDED
BPF_PROG_TYPE(BPF_PROG_TYPE_SECCOMP, seccomp)
#endif
```

Listing 3.2: seccomp BPF_PROG_TYPE

Further, the seccomp program type is added within `include/uapi/linux/bpf.h` to the eBPF program types list (Listing 3.3).

```
enum bpf_prog_type {
        ...
        BPF_PROG_TYPE_SECCOMP,
 };
```

Listing 3.3: seccomp bpf_prog_type

Within `include/uapi/linux/seccomp.h` is added the extended seccomp mode filter constant (Listing 3.4), then invoking `seccomp(2)` or `prctl(2)` with this mode will enable the use of eBPF seccomp filters.

```
* Valid values for seccomp.mode and
* prctl(PR_SET_SECCOMP, <mode>) */
...
#define SECCOMP_MODE_STRICT          1 /*hard-coded filter*/
#define SECCOMP_MODE_FILTER          2 /*user-supplied filter*/
#define SECCOMP_MODE_FILTER_EXTENDED 3 /*eBPF filter from fd*/
```

Listing 3.4: mode filter extended constant

To let unprivileged users to use seccomp filters written in eBPF (`no_new_privs` needs to be set) the seccomp prog type is added within `kernel/bpf/syscall.c` to the list of program types which don't require high privileges (Listing 3.5).

```
if (type != BPF_PROG_TYPE_SOCKET_FILTER &&
                type != BPF_PROG_TYPE_CGROUP_SKB &&
                type != BPF_PROG_TYPE_SECCOMP &&
                !capable(CAP_SYS_ADMIN))
                        return -EPERM;
```

Listing 3.5: unprivileged seccomp

If the new seccomp filter configuration is enabled the function to prepare the seccomp filter (`seccomp_prepare_extended_filter()`) is defined as illustrated in the if branch, otherwise the body just consists in a return statements that returns an invalid argument error (Listing 3.6).
The function first gets the file descriptor from the user space with a call to the kernel API's function `get_user()` and then checks, through `bpf_prog_get_type()`, if the eBPF program corresponding to that file descriptor (*fd*) has the same program type of the second argument (`BPF_PROG_TYPE_SECCOMP`). If this is the case the *prog* field of the filter is initialized to *fp* and the reference counter of the filter is increased by one to indicate that it is used.

```
#ifdef CONFIG_SECCOMP_FILTER_EXTENDED
/*
 * seccomp_prepare_extended_filter
 * prepares a user-supplied eBPF fd
 * @user_filter: pointer to the user data containing an fd.
 *
 * Returns filter on success or an ERR_PTR on failure.
 */
static struct seccomp_filter *
seccomp_prepare_extended_filter(const char __user *user_fd)
{
        ...
        /* Fetch the fd from userspace */
        if (get_user(fd, (int __user *)user_fd))
        ...
        /* Allocate a new seccomp_filter */
        sfilter = kzalloc(sizeof(*sfilter), GFP_KERNEL | \
              __GFP_NOWARN);
        ...
        fp = bpf_prog_get_type(fd, BPF_PROG_TYPE_SECCOMP);
        ...
        sfilter->prog = fp;
        refcount_set(&sfilter->usage, 1);

        return sfilter;
}
#else
static struct seccomp_filter *
seccomp_prepare_extended_filter(const char __user *filter_fd)
{
        return ERR_PTR(-EINVAL);
}
#endif
```

Listing 3.6: prepare_extended_filter

A new parameter (*filter_type*) is added to the internally called function `seccomp_set_mode_filter()` (Listing 3.7). This argument is needed to distinguish between cBPF and eBPF filters usage so the proper `seccomp_prepare_{extended_filter, user_filter}` function can be called.

```
static long seccomp_set_mode_filter(unsigned int flags,
                              const char __user *filter,
                              unsigned long filter_type)
{
 /*We use SECCOMP_MODE_FILTER for both eBPF and cBPF filters*/
```

```
    const unsigned long filter_mode = SECCOMP_MODE_FILTER;
    struct seccomp_filter *prepared = NULL;
    long ret = -EINVAL;
    ...
    /* Prepare the new filter before holding any locks. */
    if (filter_type == SECCOMP_SET_MODE_FILTER_EXTENDED)
            prepared = seccomp_prepare_extended_filter(filter);
    else if (filter_type == SECCOMP_SET_MODE_FILTER)
            prepared = seccomp_prepare_user_filter(filter);
    else
            return -EINVAL;
    ...
}
```

Listing 3.7: seccomp_set_mode_filter

As reported in the section A.2 of the Appendix A seccomp can be enabled either with `prctl(2)` or `seccomp(2)`. The common entry point for the aforementioned system calls is represented by `do_seccomp()` function. In fact this function appears within both `seccomp(2)` (Listing 3.8) and `prctl(2)` implementations.

```
SYSCALL_DEFINE3(seccomp, unsigned int, op, unsigned int,
                flags, void __user *, uargs)
{
        return do_seccomp(op, flags, uargs);
}
```

Listing 3.8: seccomp implementation

In the case of `prctl(2)` implementation (see Listing 3.9) the `do_seccomp()` function is called internally by `prctl_set_seccomp()`, as shown in the Listing 3.11

```
SYSCALL_DEFINE5(prctl, int, option, unsigned long, arg2,
unsigned long, arg3, unsigned long, arg4, unsigned long, arg5)
{
  switch(option){
  ...
  case PR_SET_SECCOMP:
        error = prctl_set_seccomp(arg2, (char __user *)arg3);
        break;
  ...
}
```

Listing 3.9: prctl implementation

The patch adds a new case for *op* argument of `do_seccomp()` (Listing 3.10). This value determines which function to call to prepare the filter.

```
static long do_seccomp(unsigned int op, unsigned int flags,
                       void __user *uargs)
{
 switch (op) {
 ...
 case SECCOMP_SET_MODE_FILTER:
        return seccomp_set_mode_filter(flags, uargs, op);
 case SECCOMP_SET_MODE_FILTER_EXTENDED:
        return seccomp_set_mode_filter(flags, uargs, op);
 ...
 }
}
```

Listing 3.10: do_seccomp

A new case to select extended filter mode is added to `prctl_set_seccomp()` function which configures seccomp mode and then calls `do_seccomp()` (Listing 3.11).

```
long prctl_set_seccomp(unsigned long seccomp_mode,
                        char __user *filter)
{
    switch(op) {
    ...
        case SECCOMP_MODE_FILTER_EXTENDED:
                op = SECCOMP_SET_MODE_FILTER_EXTENDED;
                uargs = filter;
                break;
                ...
     }
    return do_seccomp(op, 0, uargs);
}
```

Listing 3.11: prctl_set_seccomp

A function which checks if the eBPF program is accessing a valid address within the seccomp data structure is needed (Listing 3.12). This function is used by the verifier to check eBPF program memory accesses to the context.

```
static bool seccomp_is_valid_access(int off, int size,
                enum bpf_access_type type,
                struct bpf_insn_access_aux *info)
{
 if (type != BPF_READ)
  return false;

 if (off < 0 || off + size > sizeof(struct seccomp_data))
  return false;

 switch (off) {
  case bpf_ctx_range_till(struct seccomp_data, args[0], \
        args[5]):
   return (size == sizeof(__u64));
  case bpf_ctx_range(struct seccomp_data, nr):
   return (size == FIELD_SIZEOF(struct seccomp_data, nr));
  case bpf_ctx_range(struct seccomp_data, arch):
   return (size == FIELD_SIZEOF(struct seccomp_data, arch));
  ...
 }
 return false;
}
```

Listing 3.12: seccomp_is_valid_access

The seccomp program type function prototype which lists the allowed helpers is defined as reported in Figure 3.13. Therefore with this patch a seccomp program can just get the GID, UID and use `bpf_trace_printk()` to print messages for debugging purpose.

```
static const struct bpf_func_proto *
seccomp_func_proto(enum bpf_func_id func_id)
{
 switch (func_id) {
  case BPF_FUNC_get_current_uid_gid:
        return &bpf_get_current_uid_gid_proto;
  case BPF_FUNC_trace_printk:
        if (capable(CAP_SYS_ADMIN))
         return bpf_get_trace_printk_proto();
  default:
        return NULL;
 }
}
```

Listing 3.13: seccomp_func_proto

The data structure used by the verifier is initialized as shown in Listing 3.14, similar as previously reported in the socket type program example (Listing

2.8) with the difference that in this case the values are relative to the seccomp eBPF program type.

```
const struct bpf_verifier_ops seccomp_verifier_ops = {
        .get_func_proto         = seccomp_func_proto,
        .is_valid_access        = seccomp_is_valid_access,
};
```

Listing 3.14: seccomp_verifier_ops

3.1.3 eBPF seccomp example

To have an idea on how an eBPF seccomp filter would be written after applying this patch, Listing 3.15 reports one of the examples added by Sargun. The programs return the operation not permitted error if the process tries to close a file descriptor with the value of 999.

```
/* Returns EPERM when trying to close fd 999 */
SEC("seccomp")
int bpf_prog1(struct seccomp_data *ctx)
{
        if (ctx->nr == __NR_close && ctx->args[0] == 999)
                return SECCOMP_RET_ERRNO | EPERM;

        return SECCOMP_RET_ALLOW;
}

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

Listing 3.15: eBPF seccomp example

3.2 Another application of seccomp eBPF

In addition to the reasons exposed by Sargun to support the use of eBPF for seccomp, this section introduces another possible application. Generally there are two ways to give a process a different view of its execution environment.

The first requires the concept of namespaces, which is a feature of the kernel that allows each process to be associated with a particular namespace. Therefore each process can have a different view on the resources, as if it has its own isolated instance of the global resource. Linux kernel provides 7 types of namespaces: `Cgroup`, `IPC`, `Network`, `Mount`, `PID`, `User`, `UTS` [61]. A process must belong to some namespace and if it is not specified Linux adds the process to the default namespaces. This mechanism is used to implement a lightweight virtualization tool (containers).

The code which supports the namespace functionality resides inside the kernel and it involves between 7% and 15% of the core Kernel code [64]. As a consequence the kernel attack surface increases because a flaw can open the way to privilege escalation exploits. For example the CVE 2013-1858 affects the implementation of the user namespace in the Linux kernel before 3.8.3 leveraging the mismatch between the scope of two flags (`CLONE_NEWUSER`, `CLONE_FS`) used in the `clone(2)` system call [62]. The Figure 3.1 provides an high level overview on how the namespace feature is an integral part of the kernel and how this affects the kernel attack surface (highlighted in orange).

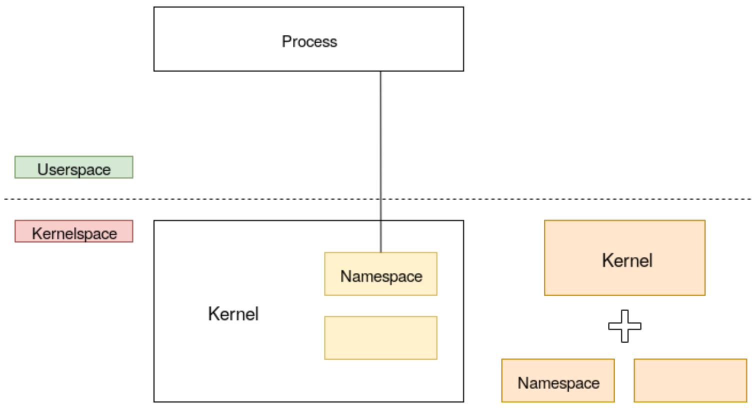

Figure 3.1: Namespace

The second approach consists of introducing an intermediate layer located in the userspace with the task of providing the process with a different view of the resources. The kernel attack surface in this case remains the same (Figure 3.2), anyway this approach is less efficient because adding a layer introduces an overhead.

A third approach could be to let the intermediate layer to offload all the

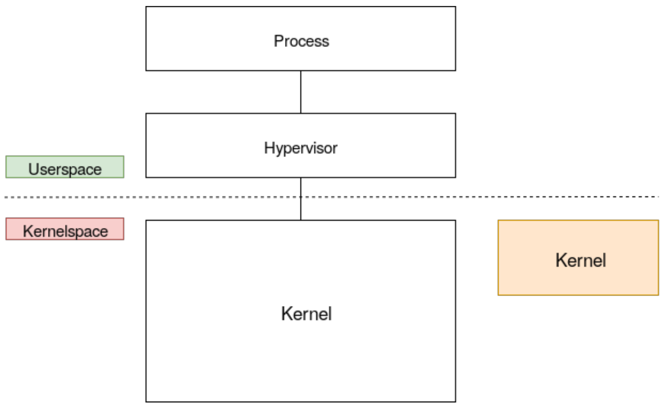

Figure 3.2: Intermediate layer

possible computations to the kernel (Figure 3.3). This solution may be implemented taking advantage of BPF. Furthermore, it is both efficient, due to the offload of some operations to the kernel, and also safe because: although some code is loaded in to the kernel, a BPF program is first checked by the verifier which guarantees that it doesn't harm the kernel. As a result there is a minor increase of the kernel attack surface.

UMVU, which is an implementation of VUOS is an example of the aforementioned approach. The idea behind VUOS is that it is possible to give processes their specific running environment (view). UMVU uses partial virtual machines (PVM). PVM is a system call virtual machine which operates as a filter, meaning that a system call can be forwarded to the hosting kernel, if it needs access to non virtualized resources, or processed by the hypervisor. Basically UMVU intercepts system calls and modifies their behaviour accordingly to the calling process view [63, 64].

The current tracer implementation handles system calls manipulation us-

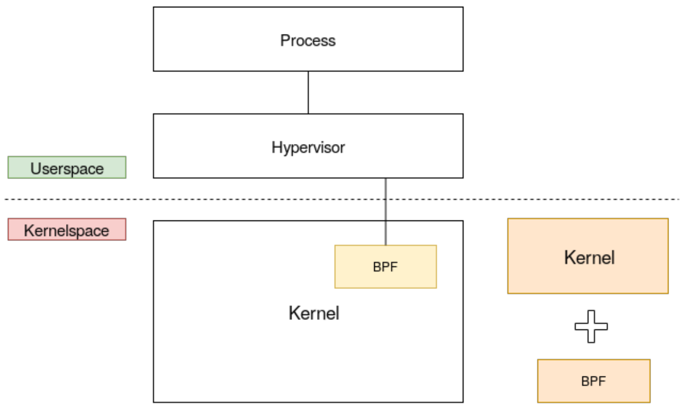

Figure 3.3: intermediate layer with bpf

ing `ptrace(2)` which is mainly used for system calls tracing and debugging purposes. This brings some performance issues because for every system call the tracee is stopped and two calls to `ptrace(2)` are required (one at the enter to and one at exit from a system call). The reason is that there is no way to tell the Kernel which system calls to trace [64].

The forwarding decision is made using seccomp. However, seccomp filters are limited to cBPF which doesn't provide maps and other features exposed by eBPF. For instance, maps could be used to store the set of system calls (e.g. open) file descriptors which are real/virtual. Consequently, the hypervisor could communicate with the offloaded module to specify the set of real file descriptors, then a system call with a real file descriptor would be directly elaborated by the kernel without redirecting it to the hypervisor.

3.3 Would eBPF seccomp introduce new weaknesses?

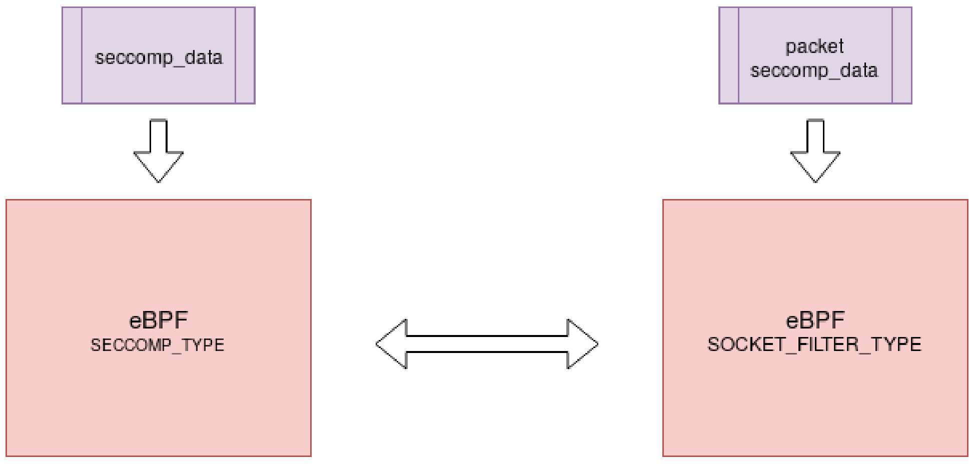

Figure 3.4: Hypothetical scenario

If it is assumed an hypothetical scenario where: eBPF support for seccomp is already implemented giving to this program type the same constraints given to the socket filter program type; there is a malicious seccomp eBPF program which violates system security when it handles a particular configuration of **struct** seccomp_data.
Then if the malicious seccomp eBPF program can be rewritten as a packet filter program, sending a packet which contains the same sequence of bytes of the `seccomp_data` to the filter, should bring the system in the same compromised state.

As a first attempt to consider this hypothesis, a test was conducted to verify if an UDP socket with a eBPF program attached to it can receive a `struct seccomp_data` as a payload and act on it. To realize the test an ethernet frame crafter was written in C language and was used to send an UDP datagram to another physical machine where the eBPF filter was running. The first field (`nr`) of the `seccomp_data` was initialized to the value of an arbitrary system call (e.g. `__NR_exit`) and the fiter shown in the Listing 3.16 was used on the second machine. The filter used an offset of `8` bytes to directly access the datagram payload to load the first `4` bytes (`nr` field of `seccomp_data`). As a result the filter successfully received only the packets transporting a `seccomp_data` with the value of `__NR_exit`. That would mean that eBPF seccomp would not introduce new weaknesses which are not already present because the compromised state would be reached through the existing packet filtering mechanism.

```
SEC("socket")
int bpf_prog(struct __sk_buff *skb)
{
        int nr = load_word(skb, 8);
        if (nr == 0x3c000000) //__NR_exit
                return -1; // return the entire packet
        else
                return 0; // discard the packet
}
char _license[] SEC("license") = "GPL";
```

Listing 3.16: nr filter

Anyway few considerations are needed: seccomp is a critical piece of code, consequently the maintainers are not inclined to add new features. Furthermore, Alexei Starovoitov the BPF mantainer, talking about how the verifier is under continuous development, argued that he *"would hate to see* *arguments against adding more verifier features just because eBPF is used by seccomp/landlock/other_security_thing"* [65]. Therefore, it may be difficult to see seccomp support implemented in a brief time.

Conclusions

eBPF is a promising technology that is undergoing an intensive development. The increasing number of kernel hooks, maps and program types combined with BPF calls have introduced safe kernel programmability which allows programmers to create user-space programs that can communicate with kernel-space eBPF programs. These features can be used to aggregate kernel data for tracing purposes but also to store maps and programs within pinned objects that can be retrieved at a later time. It is worth to mention that most of program types require high privileges and that probably this scenario will not change [66].

As discussed, currently the main applications are represented by tracing and networking domains, in particular it is worth to remember the ability to offload XDP eBPF programs to NICs hardware which is of paramount importance to avoid the high CPU overload due to the new available network speeds.

Although the recent advancements of eBPF and different expressions of interest in eBPF support for seccomp filters, at the moment it doesn't seem that it will be added in a brief time for different reasons: seccomp is critical code; BPF mantainer reconsidered unprivileged BPF and he wants to freely add new features to the verifier.

Future works could go in different directions: deeper explore other program types, applications and tools (e.g. bpftool) with the aim to produce a more accessible documentation and basic examples; further explore the hypotetheical scenario described in the last section studying in more details how the introduction of eBPF support for seccomp would affect kernel code; write/apply a patch which adds eBPF seccomp support, modify UMVU implementation accordingly and compare the performances.

Appendix A

Classic BPF usage

This appendix describes the modus operandi to write and to use cBPF filters. The first section illustrates how to apply filters for packet filtering, instead the second section explains how to apply filters to system calls .
Each section describes the components (data structures, macros, headers, constants) and the steps to follow to implement a filter. Furthermore, practical examples are explained in detail. All the examples are available at the following repository: `https://github.com/midist0xf/cbpfexamples`.

A.1 Packet Filtering

The main steps to follow to write a network packets filter are the following:

- create a socket for network communication
- write a filter managing the offsets based on the socket type
- attach the filter to the socket

A.1.1 Socket creation

It is important to know which socket type is used because, at each network stack level, the filter is applied to the payload relative to that level. For

example, a socket with the AF_PACKET domain and with the SOCK_RAW type can access also the data link information, while TCP or UDP sockets can access just to the transport level information. Then the more the packet goes upward through the network stack, the less information are visible to the filter [30].
The aforementioned considerations affect filters writing because, to access by offset the payload's fields of interests, first the base address needs to be determined. To highlight this difference the examples will use two socket types: packet socket and UDP socket.
The basic network programming tools required to realize the examples are briefly reported.

Listing A.1 shows the `socket(2)` system call prototype:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Listing A.1: socket system call

domain specifies the protocol family used to communicate. *type* specifies the semantic of the communication and *protocol* specifies the type of the protocol to use with the socket.
The code fragments in the Listing A.2 and in the Listing A.3 show the calls to create both sockets. Checks on the returned values are omitted for brevity.

```
...
sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

```
...
```

Listing A.2: packet socket

Packet socket (AF_PACKET) are used to send and receive raw packets at the data link level. With the SOCK_RAW socket type, packets include data link level header. ETH_P_ALL indicates that all protocols encapsulated in ethernet frames are received. It is worth to note that packets are passed to any raw socket before they are passed to the protocols implemented within the kernel [31].

```
...
sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
...
```

Listing A.3: udp socket

AF_INET indicates the familiy of the IPv4 protocols, while SOCK_DGRAM specifies that the semantic is UDP. UDP sockets allow to access the packet at an higher level of the network stack, then starting from the information contained in the UDP datagram.

A.1.2 Writing a filter

Once the socket type has been chosen is possible to write the proper filter. To write a filter it is required to know how the network stack operates, the formats of packets headers and the semantic of the instructions. Please refer to the Appendix C for packet headers structure details. The semantic of the instructions is available at `https://www.freebsd.org/cgi/man.cgi?bpf(4)`.

A.1.2.1 Data structures

The user space program through **#include**<linux/filter.h> can access to the data structure shown in the Listing A.4:

```
struct sock_filter {    /* Filter block */
        __u16   code;   /* Actual filter code */
        __u8    jt;     /* Jump true */
        __u8    jf;     /* Jump false */
        __u32   k;      /* Generic multiuse field */
};
```

Listing A.4: struct sock_filter

struct sock_filter is the data structure which represents a single instruction for the virtual machine. `code` indicates the operation code, `jt` and `jf` are the jump offsets while `k` is a generic value used differently by the diverse instructions [33, 35].

```
struct sock_fprog {     /* Required for SO_ATTACH_FILTER. */
        unsigned short  len;    /* Number of filter blocks */
        struct sock_filter __user *filter;
};
```

Listing A.5: struct sock_fprog

struct sock_fprog (Listing A.5) is the data structure for which a pointer to it is passed to the kernel as a parameter of the `setsockopt(2)` system call. `len` indicates the number of the filter instructions and `filter` is a pointer to the program which represents the filter [35].

Suppose now to write a program which filters IP packets that encapsulate UDP datagrams with 1030 as source port.

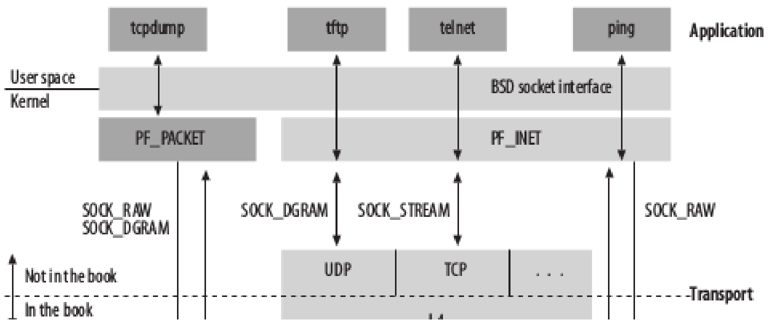

Figure A.1: Tcpdump socket [32].

A.1.2.2 Packet socket example

To generate the filter code is possible to use `tcpdump`. This network tool, allows to use expressions written in an high level language, which are translated in programs written in cBPF [1]. Furthermore `tcpdump`, through the library `libpcap`, creates a packet socket (see Figure A.1) to receive packets, as a consequence all the offsets will be already correct.

Suppose `src port 1030` is the filter expression. `tcpdump` allows to generate three different filter representations:

- `tcpdump udp and src port 1030 -d`

```
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2 jf 6
(002) ldb      [20]
(003) jeq      #0x11            jt 4 jf 15
(004) ldh      [54]
(005) jeq      #0x406           jt 14 jf 15
(006) jeq      #0x800           jt 7 jf 15
(007) ldb      [23]
(008) jeq      #0x11            jt 9 jf 15
(009) ldh      [20]
(010) jset     #0x1fff          jt 15 jf 11
(011) ldxb     4*([14]&0xf)
```

```
(012) ldh      [x + 14]
(013) jeq      #0x406           jt 14 jf 15
(014) ret      #262144
(015) ret      #0
```

- tcpdump udp and src port 1030 -dd

```
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 4, 0x000086dd },
{ 0x30, 0, 0, 0x00000014 },
{ 0x15, 0, 11, 0x00000011 },
{ 0x28, 0, 0, 0x00000036 },
{ 0x15, 8, 9, 0x00000406 },
{ 0x15, 0, 8, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 6, 0x00000011 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 4, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x0000000e },
{ 0x15, 0, 1, 0x00000406 },
{ 0x6, 0, 0, 0x00040000 },
{ 0x6, 0, 0, 0x00000000 },
```

- tcpdump udp and src port 1030 -ddd

```
16
40 0 0 12
21 0 4 34525
48 0 0 20
21 0 11 17
40 0 0 54
21 8 9 1030
21 0 8 2048
```

```
48 0 0 23
21 0 6 17
40 0 0 20
69 4 0 8191
177 0 0 14
72 0 0 14
21 0 1 1030
6 0 0 262144
6 0 0 0
```

In the first output the filter is represented by human-readable instructions. In the second output the filter is represented as a usable C fragment (in the case of a packet socket). In the third output first the number of the instructions is reported and then instructions are illustrated as decimal numbers.
Any filter can be written using macros (Listing A.6), which are defined, along with the other instructions codes, within **#include**<linux/filter.h>.

```
/*
 * Macros for filter block array initializers.
 */
#ifndef BPF_STMT
#define BPF_STMT(code, k) \
   { (unsigned short)(code), 0, 0, k }
#endif
#ifndef BPF_JUMP
#define BPF_JUMP(code, k, jt, jf) \
   { (unsigned short)(code), jt, jf, k }
#endif
```

Listing A.6: macro BPF

Both macros expand to initialize a single instruction of the filter. As an example it is explained how to write the filter using macros, illustrating for each step the meaning of the instructions and how they refer to the packet.

The following instructions considered all together implement the filter.

```
/* check if the ethernet type field is ip6 */
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x86dd, 0, 4),
```

Listing A.7: type IPv6

The instructions shown in the Listing A.7 verify that the type field in the ethernet frame header is IPv6.
The first instruction semantic is `A <- P[k:2]`, where `k=12`, meaning that it loads in the accumulator the first 2 bytes starting from the 12th byte, so the content of the type field.
The second instruction semantic is `pc += (A == k) ? 0 : 4`, where `k=0x86dd`, meaning that if the type field is `0x86dd` (IPv6) it adds to the program counter an offset of 0, otherwise it adds an offset of 4 and in this case the next instruction to be executed will be the one which checks if the type field is IPv4.

```
/* check if the next header field is UDP */
BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 20),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x11, 0, 11),
```

Listing A.8: next header UDP

The two instructions in the Listing A.8 verify if the next header field within the IPv6 packet header is UDP.
The first instruction semantic is `A <- P[k:1]`, where `k=20`, meaning that it loads in the accumulator the first byte starting from the 20th byte. The ethernet header is 14 bytes long to which are added 6 bytes to reach the start of the next header field of the IPv6 packet header.
The second instruction semantic is `pc += (A == k) ? 0 : 11`, where `k=0x11`, meaning that if the next header field is `0x11` (UDP) it adds to the program

counter an offset of 0, otherwise it adds an offset of 11 and in this case the next instruction to be executed will be the one which discards the packet.

```
/* check if the UPD header src port is 1030 */
BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 54),
BPF_JUMP(BPF_JUMP+BPF_JEQ+BPF_K, 0x406, 8, 9),
```

Listing A.9: src port 1030

The two instructions in the Listing A.9 verify that the src port field in the UDP header is equal to 1030.
The first instruction semantic is `A <- P[k:1]`, where `k=54`, meaning that it loads in the accumulator the first byte starting from the 54th byte. The ethernet header is 14 bytes long to which are added 40 bytes of the IPv6 header to reach the start of the src port field in the UDP header.
The second instruction semantic is `pc += (A == k) ? 8 : 9`, where `k=0x406`, meaning that if the src port field is `0x406` (1030) it adds to the program counter an offset of 8 and the packet is accepted, otherwise it adds an offset of 9 and the packet is discarded.

```
/* check if the ethernet type field is ip4 */
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x800, 0, 8),
```

Listing A.10: type IPv4

The semantic of the instruction in the Listing A.10 is `pc += (A == k) ? 0 : 8`, where `k=0x800`, meaning that if the type field of the ethernet header is `0x800` (IPv4) it adds to the program counter an offset of 0, otherwise it adds an offset of 8 and in this case the next instruction discards the packet.

```
/* check if the ip4 header protocol field is UDP */
BPF_STMT(BPF_LD+BPF_B+BPF_ABS, 23),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x11, 0, 6),
```

Listing A.11: protocol UDP

The two instructions in the Listing A.11 verify if the protocol field of the IPv4 packet header is UDP.
The first instruction semantic is `A <- P[k:1]`, where `k=23`, meaning that it loads in the accumulator the first byte starting from the 23rd byte. The ethernet header is 14 bytes long to which are added 9 bytes to reach the start of the protocol field in the IPv4 header.
The second instruction semantic is `pc += (A == k) ? 0 : 8`, where `k=0x11`, meaning that it adds to the program counter an offset of 0 if the protocol field is `0x11` (UDP), otherwise it adds an offset of 8 and in this case the next instruction to be executed discards the packet.

```
/* check if ip4 header fragment offset is 0 */
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 20),
BPF_JUMP(BPF_JMP+BPF_JSET+BPF_K, 0x1fff, 4, 0),
```

Listing A.12: fragment offset 0

The two instructions in the Listing A.12 verify if the fragment offset is 0 (0 means that it is the first fragment).
The first instruction semantic is `A <- P[k:2]`, where `k=20`, meaning that it loads in the accumulator the first 2 bytes starting from the 20th byte. The ethernet header is 14 bytes long to which are added 8 bytes to reach the start of the flags field, loaded along with the fragment offset field.
The second instruction semantic is `pc += (A & k) ? 4 : 0`, where `k=0x1fff`, meaning that it adds to the program counter an offset of 4 if the fragment offset is not 0 and in this case the next instruction to be executed discards the packet, otherwise it adds an offset of 0.

```
/* load ip4 header length in the index register */
```

```
BPF_STMT(BPF_LDX+BPF_B+BPF_MSH, 14),
```

Listing A.13: load IHL

The instruction in the Listing A.13 implements a frequent operation: it loads in the index register the length of the IPv4 header.
The instruction semantic is `X <- 4*(P[k:1]&0xf)`, where `k=14`, meaning that it takes the first byte starting from the 14th byte (version and Internet Header Length fields), then using a bit mask it considers just the last 4 bits (IHL) and it multiplies this value by 4 to have the header length in bytes, which finally is loaded in the index register.

```
/* check if UDP src port is 1030 */
BPF_STMT(BPF_LD+BPF_H+BPF_IND, 14),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x406, 0, 1),
```

Listing A.14: src port 1030

The two instruction in the Listing A.14 verify if the src port field in the UDP header is 1030.
The first instruction semantic is `A <- P[X+k:2]`, where `k=14`, meaning that it adds 14 (header ethernet length) to the content of the index register `X` (IPv4 header length) and starting from the resulting offset it loads the first 2 bytes in the accumulator, that is the content of the src port field of the UDP header.
The second instruction semantic is `pc += (A == k) ? 0 : 1`, where `k=0x406`, meaning that it adds to the program counter an offset of 0 if the src port is 1030 and then the next instruction to be executed accepts the packet, otherwise it adds an offset of 1 and the next instruction discards the packet.

```
/* return the entire packet */
BPF_STMT(BPF_RET+BPF_K, 0x40000),
```

Listing A.15: return

The instruction in the Listing A.15 returns the entire packet.
The instruction semantic is `accept k bytes`, where `k=0x40000`, a value which guarantees to return all the bytes of the packet.

```
/* discard the packet */
BPF_STMT(BPF_RET+BPF_K, 0),
```

Listing A.16: discard

The instruction in the Listing A.16 discards the packet.
The instruction semantic is `accept k bytes`, where `k=0`, meaning that no byte of the packet will be returned.

A.1.2.3 UDP socket example

The filter for an UDP socket instead, as already mentioned (Section A.1.1) is different. In this case it is not possible to directly use the output of `tcpdump` but it needs to be properly adapted as shown in the Listing A.17.

```
/* check if UDP src port is 1030 */
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 0),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x406, 0, 1),
/* return the entire packet */
BPF_STMT(BPF_RET+BPF_K, 0x40000),
/* discard the packet */
BPF_STMT(BPF_RET+BPF_K, 0),
```

Listing A.17: UDP socket filter

The reason is that in this scenario link level information are not visible, then the base address is related to the UDP header and not to the Ethernet header.

A.1.2.4 Macros and constants

In the aforementioned example numeric values were used both to indicate protocol types (0x800 for IPv4, etc.) and to indicate the different field offsets. To make the filter more readable and to facilitate the offsets computation it is possible to use already defined constants and macros.

#include<stddef.h> makes available the macro in the Listing A.18

```
#define offsetof(TYPE, MEMBER)((size_t)&((TYPE *)0)->MEMBER)
```

Listing A.18: macro offsetof

which returns the offset in bytes of `MEMBER` starting from the base address of `TYPE`.

#include<linux/if_ether.h> makes available constants and data structures of the ethernet header, see Listing A.19:

```
#define ETH_HLEN   14      /* Total octets in header.  */
#define ETH_P_IP   0x0800     /* Internet Protocol packet   */
#define ETH_P_IPV6    0x86DD      /* IPv6 over bluebook       */
...
```

Listing A.19: ethernet constants

#include<linux/in.h> makes available constants which are useful when handling the protocol field in the IP packet header. Few examples are reported in the Listing A.20:

```
  IPPROTO_IP = 0,       /* Dummy protocol for TCP       */
#define IPPROTO_IP          IPPROTO_IP
  IPPROTO_UDP = 17,       /* User Datagram Protocol       */
#define IPPROTO_UDP         IPPROTO_UDP
```

```
...
```

Listing A.20: protocol constants

In order to use UDP, TCP, IP, etc. protocols data structures, proper C headers are needed, for example **#include**<linux/ip.h> makes available **struct** iphdr, while **#include**<linux/udp.h> defines **struct** udphdr, and so forth. These data structures allow programmer to use the macro `offsetof` to get fields offsets easily.

As an example, in the Listing A.21 is shown how to rewrite some instructions of the previously explained filter (Listings A.7, A.8)

```
/* check if the ethernet type field is ip6 */
BPF_STMT(BPF_LD+BPF_H+BPF_ABS, offsetof(struct ethhdr, h_proto)),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETH_P_IPV6, 0, 9),
/* check if the next header field is UDP */
BPF_STMT(BPF_LD+BPF_B+BPF_ABS, ETH_HLEN +     \
      offsetof(struct ipv6hdr, nexthdr) ),    \
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, IPPROTO_UDP, 0, 11),
...
```

Listing A.21: instructions with macros

A.1.3 Attach the filter

The last step is to attach the filter to the socket through the `setsockopt(2)` system call. This step is introduced illustrating the code structure in its entirety. A packet socket is used in the example. Some details are omitted, however they are available at the repository.

```
/* initialize the filter program */
struct sock_filter bpfcode[] = {
    /* check if the ethernet type field is ip6 */
    BPF_STMT(BPF_LD+BPF_H+BPF_ABS, offsetof(struct ethhdr, \
         h_proto)),
    ...
};

struct sock_fprog bpf;
bpf.len = sizeof(bpfcode)/sizeof(struct sock_filter);
bpf.filter = bpfcode;

/* open a packet socket */
sockfd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
...

/* attach the filter to the socket */
ret = setsockopt(sockfd, SOL_SOCKET, SO_ATTACH_FILTER, \
          &bpf, sizeof(bpf));
...

close(sockfd);
```

Listing A.22: code structure

It is recalled that when the array which contains the filter code is initialized, in the case of a packet socket, it is possible to directly use the output of `tcpdump` with the option `-dd`.
Lastly, programs which use a packet socket with needs to be assigned the `CAP_NET_RAW` capability or to be executed as `root`.

A.2 Seccomp

System calls filtering with `seccomp` provide two modes:

- strict
- filter, which is fine-grained

In the first mode there is no need to write a filter, only predefined system calls are allowed.
In the second mode the programmer writes the filter using cBPF syntax implementing it based on the system calls he needs to allow or block.
Both modes can be enabled either through `prctl(2)` (since Linux 2.6.23) or through `seccomp(2)` (since Linux 3.17). To invoke seccomp it is required to use `syscall(2)` because currently the glibc library doesn't have a wrapper for seccomp [36]. All the constant regarding seccomp are available through `#include <linux/seccomp.h>`.

A.2.1 Strict mode

#include <sys/prctl.h> defines parameters for the `prctl(2)` system call, which permits to enable strict mode if invoked as shown in the Listing A.23:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
...
```

Listing A.23: prctl strict

Instead using `seccomp(2)` to enable the strict mode (Listing A.24) requires also the following headers: **#include** <sys/syscall.h>, to have access to system calls numbers (including `__NR_seccomp`); **#include** <unistd.h> to invoke `syscall(2)`.

```
syscall(__NR_seccomp, SECCOMP_SET_MODE_STRICT, 0, NULL);
```

```
...
```

Listing A.24: seccomp strict

A.2.1.1 Allowed system calls

The allowed system calls in strict mode are only 4: `read(2)`, `write(2)`, `sigreturn(2)` and `_exit(2)` (but not `exit_group(2)`). It is noteworthy that usually the applications don't directly invoke system calls, instead they calls wrapper functions exposed by the glibc library. Wrappers can internally invoke system calls with different names. Furthermore, system calls may vary depending on the underlying architecture, lastly internally called system calls can change based on glibc version [37].
An example to illustrate a first application of seccomp is now introduced, also to highlight some of the aforementioned points regarding the wrappers.

```
int main(int argc, char **argv)
{
    /* activate seccomp strict mode */
    r = prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
    ...

    _exit(0);
}
```

Listing A.25: strict _exit

When the program reported in the Listing A.25 is compiled and executed, the process is terminated by the signal `SIGKILL`.
This happens because the function `_exit(2)`, since glibc version 2.3, invokes `exit_group(2)` system call [38], which is not allowed in strict mode. The code can be modified substituting the call to `_exit(0)` with the call in the Listing A.26

```
/* syscall(2) allows to call directly
* the kernel system call  _exit */
syscall(__NR_exit, 0);
```

Listing A.26: syscall _exit

In this case the process terminates normally because an allowed system call is invoked.

A.2.2 Filter mode

The strict mode is very rigid. To have more flexibility it is possible to enable the second seccomp mode and take advantage of the data structure described in the section A.1.2.1 to write personalized filters.
Generally it is advisable to follow a whitelisting approach [37], then it is required to know all the system calls needed by the application. The main steps to follow to write a system call filter are the following:

- write the filter verifying the architecture and creating a system calls whitelist
- possibly set the bit which avoids that new processes use filters with greater privileges
- enable filter mode
- install the filter

A.2.2.1 bit no_new_privs

To enable filter mode the thread needs `CAP_SYS_ADMIN` capability assigned to it or the `no_new_privs` bit set. If the bit is not already set the following call is required:

```
r = prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
```

```
...
```

Listing A.27: no_new_privs

The call in the Listing A.27 reduces the attack surface exploitable by a non privileged user by ignoring set-user-id, set-group-id and capabilities [39], then preventing that a child process applies filters with higher privileges than the process which installed them [8]. The bit is inherited by children processes created through `clone(2)`, `fork(2)` and it is preserved after the execution of `execv(2)` system call [40].

A.2.2.2 Data structures

In addition to the already mentioned **struct** sock_filter and **struct** sock_fprog, a new data structure accessible through `#include <linux/seccomp.h>` is described. While in the packet filtering application the filters operate on packets information, in this case they operate on system call information, described within **struct** seccomp_data (Listing A.28)

```
struct seccomp_data {
    int nr;                  /* System call number */
    __u32 arch;              /* AUDIT_ARCH_* value */
    __u64 instruction_pointer;    /* CPU instruction pointer */
    __u64 args[6];     /* Up to 6 system call arguments */
};
```

Listing A.28: struct seccomp_data

where `nr` is the number of the invoked system call, `arch` indicates the architecture. This last value needs to be checked because system call numbers may vary between the architectures and also because some architectures allow processes to use different calling conventions. `instruction_pointer` represents the machine instruction that executed the system call. `args[6]` is the field to access system call arguments, for a maximum of 6 arguments

[37].

A.2.2.3 Return values

The 16 most significant bits, of the 32 bits returned by the filter, specify to the kernel the action to perform. If more filters are installed, for each invoked system call, they are executed all starting from the last added filter and, the action which will be performed by the kernel, is the one which has been encountered first and that has the highest precedence.
The actions used in the examples are `SECCOMP_RET_KILL`, which terminates the thread, `SECCOMP_RET_ALLOW` which allows the execution of the system call and `SECCOMP_RET_TRAP`, that sends a `SIGSYS` signal to the thread and sets some fields within `siginfo_t`, including `si_syscall`, useful for debugging purposes [37].

A.2.2.4 Strict mode

Listing A.29 shows a filter which acts as the strict mode but in addition it allows the `exit_group(2)` system call.

```
/* validate the architecture */
BPF_STMT(BPF_LD+BPF_W+BPF_ABS,      \
      (offsetof(struct seccomp_data, arch))),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, AUDIT_ARCH_X86_64, 1, 0),
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
/* load syscall number in the accumulator */
BPF_STMT(BPF_LD+BPF_W+BPF_ABS,      \
      (offsetof (struct seccomp_data, nr))),
/* check if the syscall number is allowed */
/* exit_group */
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_exit_group, 0, 1),
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
/* _exit */
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_exit, 0, 1),
15 BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
16 /* write */
17 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_write, 0, 1),
18 BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
19 /* read */
20 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_read, 0, 1),
21 BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
22 /* sigreturn */
23 BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_rt_sigreturn, 0, 1),
24 BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
25 /* kill the process */
26 BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL)
```

Listing A.29: strict più exit_group

To validate the architecture the `arch` field is loaded in the accumulator and it is compared with the proper value chosen between the constants (available within `#include <linux/audit.h>`) which distinguish different system call tables (lines 1-5). Then the `nr` field is loaded in the accumulator (lines 6-8) and its value is compared with the numbers of the system calls reported in the whitelist (lines 9-25). The list consists of the system calls allowed by default in the strict mode plus the `exit_group` system call. The filter, in contrast to the behaviour of the example in the Listing A.25, now allows also the call `_exit(0)`, so in this case the process would not terminate.

A.2.2.5 dup

To illustrate how to filter a system call based on its actual parameters suppose to allow the `dup(STDOUT_FILENO)` system call. At the filter reported in the Listing A.29, after the line 24, the following lines should be added.

```
/* dup(STDOUT_FILENO) */
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_dup, 0, 3),
BPF_STMT(BPF_LD+BPF_W+BPF_ABS,      \
        (offsetof (struct seccomp_data, args[0]))),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, STDOUT_FILENO, 0, 1),
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
```

Listing A.30: dup

The first and only argument of the system call is loaded in the accumulator and it is compared with the constant `k=STDOUT_FILENO`, if the expression is true the call is allowed, otherwise a jump to the instruction which terminates the process is performed.

A.2.2.6 Macros

Some sequences of frequently used instructions can be rewritten in a more compact way using the macros reported in the Listing A.31:

```
#define arch_num (offsetof(struct seccomp_data, arch))
#define syscall_num (offsetof(struct seccomp_data, nr))

#define VERIFY_ARCHITECTURE(arch_audit_num) \
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, arch_num), \
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, \
                arch_audit_num, 1, 0), \
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL)

#define LOAD_SYSCALL_NUMBER \
        BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_num)

#define ALLOW_SYSCALL(syscall_name) \
        BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, \
                __NR_##syscall_name, 0, 1), \
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW)

#define KILL_THREAD \
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL)

#define TRAP_THREAD \
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRAP)
```

Listing A.31: macro seccomp

The complete filter, which allows the execution of both the default strict mode system calls, plus `exit_group(2)` and `dup(2)` with the parameter `STDOUT_FILENO`, can be rewritten as shown in the Listing A.32

```
VERIFY_ARCHITECTURE(AUDIT_ARCH_X86_64),
LOAD_SYSCALL_NUMBER,
ALLOW_SYSCALL(exit_group),
ALLOW_SYSCALL(exit),
ALLOW_SYSCALL(write),
ALLOW_SYSCALL(read),
ALLOW_SYSCALL(rt_sigreturn),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_dup, 0, 3),
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, \
     (offsetof (struct seccomp_data, args[0]))),
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, STDOUT_FILENO, 0, 1),
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
KILL_THREAD
```

Listing A.32: seccomp filter with macros

A.2.3 Installing the filter

The last step is the filter installation. As already seen for mode strict activation, either `prctl(2)` (see Listing A.33) or `seccomp(2)` (see Listing A.34) can be used.

```
syscall(__NR_seccomp, SECCOMP_SET_MODE_FILTER, 0, &bpf);
...
```

Listing A.33: seccomp mode filter

An example which shows the complete code structure including all the steps previously described is reported in the Listing A.34.

```
/* initialize the filter program */
struct sock_filter bpfcode[] = {
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, \
          (offsetof(struct seccomp_data, arch))),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, AUDIT_ARCH_X86_64, 1, 0),
    BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
    ...
};

struct sock_fprog  bpf;
bpf.len = ( sizeof bpfcode / sizeof bpfcode[0] );
bpf.filter = bpfcode;

/* set no_new_privs bit (if the thread doesn't
 * have CAP_SYS_ADMIN assigned) */
r = prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
...

/* install the filter */
r = prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &bpf);
...
```

Listing A.34: seccomp code structure

A.2.3.1 Debug

A command which can be useful while writing a filter is `strace(1)`, that can be used to track both the system calls called by a process and the received signals [41]. Suppose to add new instructions, which allow the `fork(2)` system call, to a preexisting filter. A code fragment of the original filter is shown in the Listing A.35.

```
struct sock_filter  bpfcode[] = {
        ...
        BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL)
};
...
int main(int argc, char **argv)
{
          r = prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
          ...
          r = prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &bpf);
          ...
          fork();
          return 0;
}
~
```

Listing A.35: fork example

A first attempt could consist of adding the instructions reported in the Listing A.36 to the filter

```
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_fork, 0, 1),
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
```

Listing A.36: allow fork

However if the program is compiled and executed it can be noted that the process is terminated. To verify which is the system call that causes the termination, the line in the Listing A.37 can be temporarily substituted with the line in the Listing A.38

```
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL)
```

Listing A.37: kill

```
BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_TRAP)
```

Listing A.38: trap

As a consequence now is possible to examine the information contained within `siginfo_t`. In fact executing the program as in the Listing A.39

```
$ strace ./prog
```

Listing A.39: strace prog

produces the output in the Listing A.40

```
--- SIGSYS {si_signo=SIGSYS, si_code=SYS_SECCOMP,  \
      si_call_addr=0x7f1d65215b1c, si_syscall=__NR_clone, \
      si_arch=AUDIT_ARCH_X86_64} ---
+++ killed by SIGSYS (core dumped) +++
Bad system call (core dumped)
```

Listing A.40: strace output

which highlights, through the `si_syscall` field, how in reality the actual system call is `__NR_clone`, then also in this example, as already seen in the Listing A.25, the called wrapper internally invokes a system call with a different name (for this system call since glibc 2.3.3 version [42]). Consequently `__NR_fork` needs to be substituted with `__NR_clone` in the filter instruc-

tions.
In conclusion, `strace` is useful to create the whitelist because it helps to identify all the system calls needed by the application to work properly.

A.2.4 Libseccomp

To simplify filter writing the `libseccomp` library has been introduced. This library provides an high level API to abstract from the underlying cBPF language [43]. The main steps to use the library can be summarised in the following [44]:

- initialize the seccomp filter state defining the default action to perform
- add the rules to the filter
- load the filter in to the kernel
- release the seccomp filter state

To explore the use of the main functions provided by the library, an example which implements the filter already described in the Listing A.29, is now explained along with the API exposed by `libseccomp`.

A.2.4.1 seccomp_init

The first step consists in the filter initialization through the `seccomp_init()` function. The prototype is shown in the Listing A.41

```
scmp_filter_ctx seccomp_init(uint32_t def_action);
```

Listing A.41: seccomp_init

The function `seccomp_init()` needs to be called before of all the others functions. As an argument it takes the default action to perform, meaning the action to execute if the system call doesn't match any rule. Available default actions are the following:

- SCMP_ACT_KILL terminates the thread with the `SIGSYS` signal
- SCMP_ACT_KILL_PROCESS terminates the entire process with the `SIGSYS` signal
- SCMP_ACT_TRAP the thread receive a `SIGSYS` signal which can be captured
- SCMP_ACT_ALLOW has no effect

The first code fragment to implement the filter is reported in the Listing A.42.

```
/* initialize filter state and set
 * kill as deafult action */
scmp_filter_ctx ctx;
ctx = seccomp_init(SCMP_ACT_KILL);
```

Listing A.42: seccomp_init

`scmp_filter_ctx` is a data structure which contains the filter context, returned on success from `seccomp_init()`.

A.2.4.2 seccomp_rule_add

Now the rules can be added to the filter through `seccomp_rule_add()` (Listing A.43).

```
int seccomp_rule_add(scmp_filter_ctx ctx, uint32_t action,
                     int syscall, unsigned int arg_cnt, ...);
```

Listing A.43: seccomp_rule_add

ctx is the context returned during the filter initialization phase. *action* can take one of the values already seen for `seccomp_init()`. *syscall* indicates the number of the system call to which the rule is referring. It is advisable to pass this argument using the macro `SCMP_SYS(syscall_name)` to be sure that all the correct operations related to the different architectures will be performed. *arg_cnt* specifies the number of the rules related to the system call arguments.

To write these last rules different macros are available which allow to compare an actual parameter with a particular value (Listing A.44).

```
struct scmp_arg_cmp SCMP_CMP(unsigned int arg,
                             enum scmp_compare op, ...);
```

Listing A.44: SCMP_CMP

`SCMP_CMP()` allows to choose an arbitrary argument through *arg* and to use one of the comparison operator. The `SCMP_A{0-5}` macros instead, refer to a specific argument of the system call.

The second code fragment of the filter, which implements the whitelist, is illustrated in the Listing A.45. In this scenario the general use of the system calls is allowed, then there are no rules about the arguments and for this reason *arg_cnt* is `0`.

```
/* add rules to build the whitelist */
/* exit_group */
r = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, \
      SCMP_SYS(exit_group), 0)
/* exit */
r = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, \
      SCMP_SYS(exit), 0);
/* write */
```

```
r = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, \
      SCMP_SYS(write), 0);
/* read */
r = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, \
      SCMP_SYS(read), 0);
/* sigreturn */
r = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, \
      SCMP_SYS(rt_sigreturn), 0);
```

Listing A.45: whitelist libseccomp

To permit the process to duplicate its standard output, as in the previously shown example (Listing A.30), a new rule can be added as in the Listing A.46)

```
/* dup(STDOUT_FILENO) */
r = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(dup), 1,
                        SCMP_A0(SCMP_CMP_EQ, STDOUT_FILENO));
```

Listing A.46: dup SCMP_A0

or using the more generic macro `SCMP_CMP()`, as in the Listing A.47

```
/* dup(STDOUT_FILENO) */
r = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(dup), 1,
                        SCMP_CMP(0, SCMP_CMP_EQ, STDOUT_FILENO));
```

Listing A.47: dup SCMP_CMP

In both cases *arg_cnt* is set to `1` to indicate that there is a rule relative to the system call arguments and also the comparison operator is the same (`SCMP_CMP_EQ`) which specifies that there is a match only if the chosen argument is equal to a particular value (`STDOUT_FILENO`). With the generic macro the rule relative to the system call arguments has three parameters, of which the first specifies which argument to evaluate. In this example the only possible value is `0` because `dup(2)` takes just one argument.

A.2.4.3 seccomp_rule_add_array

In addition to `seccomp_rule_add()` it is possible to invoke `seccomp_rule_add_array()` that allows to pass the system call arguments rules as an array. For example to permit the call shown in the Listing A.48

```
open("hello.txt", O_WRONLY|O_CREAT, S_IWUSR|S_IRUSR );
```

Listing A.48: open con flag

the proper rule can be added through the call reported in the Listing A.49 which has `arg_cmp` as the last parameter, that is a pointer to an array of `struct scmp_arg_cmp` containing the rules for the single arguments.

```
 /* open */
 r = seccomp_rule_add_array(ctx, SCMP_ACT_ALLOW, \
        SCMP_SYS(openat), 2, arg_cmp);
```

Listing A.49: add array

In particular, using `strace`, may be verified how `open(2)` internally calls `openat(2)` with `AT_FDCWD` as argument to specify that the pathname needs to be interpreted relatively to the current directory. The initialized array with the proper rules is shown in the Listing A.50.

```
struct scmp_arg_cmp arg_cmp[] = {
        SCMP_A0(SCMP_CMP_EQ, AT_FDCWD),
        SCMP_A2(SCMP_CMP_EQ, O_WRONLY|O_CREAT),
        SCMP_A3(SCMP_CMP_EQ, S_IWUSR|S_IRUSR)
};
```

Listing A.50: openat arguments array

It is worth to note that the filter is not active until it is loaded in to the kernel.

A.2.4.4 seccomp_load

Loads the filter in to the kernel (Listing A.51).

```
int seccomp_load(scmp_filter_ctx ctx);
```

Listing A.51: seccomp_load

The third filter code fragment is reported in the Listing A.52

```
r = seccomp_load(ctx);
```

Listing A.52: load

A.2.4.5 seccomp_release

Releases the filter and frees the memory associated to `ctx` (Listing A.53)

```
void seccomp_release(scmp_filter_ctx ctx);
```

Listing A.53: seccomp_release

After loading the filter into the kernel the filter state can be destroyed releasing the associated resources. The filters already loaded into the kernel are not affected. Furthermore, after the function is invoked, the filter context can't be used anymore [45, 46]. Therefore the filter code includes the call shown in the Listing A.54,

```
seccomp_release(ctx);
```

Listing A.54: release

wherever an error, denoted by a return value, occurs in one of the library functions.

A.2.4.6 libseccomp strict example

The overall filter code structure comprehensive of all the steps previously explained is now illustrated (Listing A.55)

```
/* initialize filter state and set
 * kill as default action */
scmp_filter_ctx ctx;
ctx = seccomp_init(SCMP_ACT_KILL);

/* add rules to build the whitelist */
...
/* sigreturn */
r = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, \
      SCMP_SYS(rt_sigreturn), 0);
if (r<0){
   seccomp_release(ctx);
}

/* load the filter into the kernel */
r = seccomp_load(ctx);
if (r<0){
   seccomp_release(ctx);
}

/* release filter state */
seccomp_release(ctx);
...
_exit(EXIT_SUCCESS);
```

Listing A.55: seccomp full code

The library offers also other functions. The explanation of few of them follows.

A.2.4.7 seccomp_syscall_priority

It allows to assign a priority to the system calls in the filter (Listing A.56).

```
int seccomp_syscall_priority(scmp_filter_ctx ctx,
                             int syscall, uint8_t priority);
```

Listing A.56: seccomp_syscall_priority

System calls with higher priority are positioned at the beginning in the filter code. *priority* may vary within a range of `0-255`. The function takes a context as its first argument, so it needs to be called before the context is made unusable by `seccomp_release()`.

A.2.4.8 seccomp_syscall_export_pfc

It allows to export the filter in a human readable format (Listing A.57).

```
int seccomp_export_pfc(const scmp_filter_ctx ctx, int fd);
```

Listing A.57: lst:seccomp_export_pfc

To invoke this function some system calls need to be allowed (including `fcntl(2)`). To identify them, as already seen in the section, `strace` can be used.

An example is now introduced to show how the change of a system call priority has an impact on its position within the filter. In the Listing A.58 is reported the output of the call to `seccomp_export_pfc(ctx, STDOUT_FILENO)` on a filter which allows both `exit_group` and `fcntl`.

```
$ ./strictexlib
```

```
#
# pseudo filter code start
#
# filter for arch x86_64 (3221225534)
if ($arch == 3221225534)
  # filter for syscall "exit_group" (231) [priority: 65535]
  if ($syscall == 231)
    action ALLOW;
  # filter for syscall "fcntl" (72) [priority: 65535]
  if ($syscall == 72)
    action ALLOW;
...
```

Listing A.58: fcntl pre

It can be noted that `fcntl` has the same priority of `exit_group`.
Instead if the program invokes `seccomp_syscall_priority()` as shown in the Listing A.59

```
 r = seccomp_syscall_priority(ctx, SCMP_SYS(fcntl), 1);
```

Listing A.59: seccomp_syscall_priority

the system call priority is changed through the hint `1` passed as an argument, which will be used by the filters generator to modify the positioning of the system calls. The new output is shown in the Listing A.60

```
$./strictexlib
#
# pseudo filter code start
#
# filter for arch x86_64 (3221225534)
if ($arch == 3221225534)
  # filter for syscall "fcntl" (72) [priority: 131071]
  if ($syscall == 72)
    action ALLOW;
  # filter for syscall "exit_group" (231) [priority: 65535]
  if ($syscall == 231)
    action ALLOW;
...
```

Listing A.60: fcntl post

which clearly shows how `fcntl` has been moved up. The effect is to reduce the overhead for the system calls with an higher priority.

Appendix B

Extended BPF usage

To write an eBPF program basically two approaches can be adopted :

- using eBPF instruction set
- using an higher level language (e.g. C, go, rust)

With the first approach also to write a small program can become a cumbersome and time-consuming task, in fact it is like writing processor assembly. The second approach permits to compile the eBPF program written in the higher level language into an eBPF object file making the process easier. The latter approach is the most adopted and will be described.
Useful tools and libraries will be discussed along with examples. All the examples and more details are available at the following repository `https://github.com/midist0xf/ebpfexamples`.

Generally an eBPF program is made of different components [47]:

- the *backend*: the eBPF bytecode
- the *loader*: loads the eBPF bytecode into the kernel
- the *front-end*: reads data written by the backend in the data structures

- *data structures*: permit communication between backends and frontends

Section B.1 explains how to use llvm/clang to compile eBPF programs written in restricted C and how to load them into the kernel using C as frontend language and with the help of `libbpf` library.
Section B.1.8 describes `libbpf` that is a BPF library which goal is to provide a standard way to access eBPF object files.

B.1 Clang - LLVM

LLVM Clang compiler provides eBPF backend which compiles a subset of C language (often referred as "restricted C") to eBPF bytecode [56]. Restricted C does not provide global variables, variadic functions, floating-point numbers, and passing structures as function arguments [51]. Also (not unrolled) loops were forbidden but, since recently, bounded loops are allowed [57].
For an example on how to use `clang` and `llc` to compile an eBPF program written in restricted C see section B.10.

The program structure and the main components required to write an eBPF program will be described before to introduce few complete examples.

B.1.1 eBPF program files structure

Each program can be divided in two files: one which contains the frontend and the loader code (`*_user.c`) and one which contains the eBPF program and data structures code (`*_kern.c`). Several examples are available in the `samples/bpf` directory within the kernel source code but with the disadvantage that they need to be compiled from within the kernel source tree. Therefore, it is also described how to use libraries and tools to compile and loads eBPF programs indipendently from the kernel source tree.

B.1.2 bpf system call

`bpf(2)` system call is available since Linux 3.18 through **#include** <linux/bpf.h> and allows to execute operations related to eBPF [51].

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

Listing B.1: bpf

cmd value determines the operation performed by `bpf(2)`. *attr* is a pointer to an union of anonymous structures used by different `bpf` commands. `size` is the size of the union pointed by `attr`.

enum bpf_cmd within `linux/bpf.h` lists all the available commands usable as *cmd* value. Few of them are reported in the Listing B.2:

```
enum bpf_cmd {
        BPF_MAP_CREATE,
        BPF_MAP_LOOKUP_ELEM,
        ...
        BPF_PROG_LOAD,
        BPF_OBJ_PIN,
        BPF_OBJ_GET,
        ...
};
```

Listing B.2: bpf_cmd

They allow to operate on maps (`BPF_MAP_*`), to verify and load programs into the kernel (`BPF_PROG_LOAD`), to pin maps and programs in the filesystem (`BPF_OBJ_*`) and more.

Regarding the anonymous structures, for example, to operate on maps with `BPF_MAP_*_ELEM` commands the following attributes are needed:

```
union bpf_attr {
    struct { /* anonymous struct used by
           * BPF_MAP_*_ELEM commands */
           __u32            map_fd;
           __aligned_u64    key;
           union {
                   __aligned_u64 value;
                   __aligned_u64 next_key;
           };
           __u64            flags;
    };
}
```

Listing B.3: BPF_MAP_*_ELEM attributes

Suppose a lookup operation on a map is required. Once the attributes are initialized (which depends on the operation that is performed) is possible to lookup for an element in the map executing the call in the Listing B.4

```
syscall(__NR_bpf, BPF_MAP_LOOKUP_ELEM, &attr, sizeof(attr));
```

Listing B.4: syscall bpf

In this case, if the element is found, its value is stored in `value`, otherwise `-1` is returned. To perform a different operation the same steps are applied using the proper anonymous structure.

B.1.3 Macro

Writing an eBPF program requires to tell LLVM in which separated ELF sections put the eBPF program object file, maps and license. `bpf_helpers.h` (accessible in the kernel source code) makes available a macro named `SEC` which has the effect to initialize the section attribute.

```
/* helper macro to place programs, maps, license in
 * different sections in elf_bpf file. Section names
 * are interpreted by elf_bpf loader
 */
#define SEC(NAME) __attribute__((section(NAME), used))
```

Listing B.5: SEC macro

The bpf loader interprets the section name to understand to which system event to attach the ebpf program. Moreover, the kernel can allow the access to some helper-functions only to GPL compatible license, possibly rejecting programs with a different one [52], therefore a section named `license` will be present in eBPF programs.

B.1.4 eBPF program context

Each eBPF program type determines the input (context) passed to the program [51]. In the case an eBPF program is attached to a socket

```
SEC("socket")
int bpf_socket_prog(struct __sk_buff *skb)
{
    /* discard the packet */
    return 0;
}
char _license[] SEC("license") = "GPL";
```

Listing B.6: context socket

the context consists of a pointer to **struct** __sk_buff (see Listing B.6), that is an user accessible mirror of in-kernel **struct** sk_buff, which is the structure where the packet is stored in the kernel [32]. The access to the context it's converted into the access to the kernel data structure by `sk_skb_convert_ctx_access()` (see `net/core/filter.c`).

An eBPF program can be attached to several hooks in to the kernel. Suppose that a program needs to be attached to a tracepoint. It's possible to list all the tracepoints that can be used executing the command in the Listing B.7. The output is in the form of *category:name*.

```
$ sudo perf list
   ...
   syscalls:sys_enter_open                     [Tracepoint event]
   syscalls:sys_enter_open_by_handle_at        [Tracepoint event]
   syscalls:sys_enter_openat                   [Tracepoint event]
   syscalls:sys_enter_pause                    [Tracepoint event]
   ...
```

Listing B.7: perf list tracepoint

In this case the context of the program consists in a data structure which can be build using the fields found within `/sys/kernel/debug/tracing/events/`*category*`/`*name*`/format`. For instance, in the `samples/bpf/xdp_monitor_kern.c` kernel example, `SEC("tracepoint/xdp/xdp_redirect_*")` are the sections where eBPF programs will reside. Consequently, the relative tracepoint context struct is defined, as reported in the Listing B.8, based on the fields contained within `/sys/kernel/debug/tracing/events/xdp/xdp_redirect_*/format` files.

```
struct xdp_redirect_ctx
        u64 __pad;     // First 8 bytes are not accessible
                       // by bpf code
        int prog_id;   // offset:8;  size:4; signed:1;
        ...
```

```
};
```

Listing B.8: struct xdp_redirect_ctx

This solution is required because an eBPF program in the kernel can't read these files [60].
It should be noted that eBPF programs can't access the first 8 bytes of tracepoint data because they are common to all tracepoints and are used to store the pointer to **struct** pt_regs which some of the bpf helpers will use, therefore an 8 bytes pad field is added at the beginning of the tracepoint context struct [58, 60].

The general structure of an eBPF program which will be attached to a tracepoint is reported in the Listing B.9

```
struct category_name_ctx {
        ...
};
SEC("tracepoint/category/name")
int bpf_tracepoint_prog(struct category_name_ctx *ctx)
{
        ...
}
char _license[] SEC("license") = "GPL";
```

Listing B.9: context tracepoint

where *category* and *name* need to be replaced accordingly to the chosen tracepoint. See section B.1.10 for a complete example.
An article which describes the main program types contexts is available online [59].

B.1.5 eBPF program section

Suppose that the file which contains the eBPF program in the Listing B.6 is called `sockex_kern.c`. To illustrate how the eBPF program object code is loaded in the relative ELF section, first it is required to compile `sockex_kern.c` file with the command shown in the Listing B.10

```
clang -S -I. -O2 -emit-llvm -c sockex_kern.c -o - | \
      llc -march=bpf -filetype=obj -o sockex_kern.o
```

Listing B.10: clang/llvm sockex

The command first emits LLVM intermediate assembly language which is subsequently compiled into the assembly language for a specific architecture (bpf in this case). It is possible to read sections of the generated ELF file using the command reported in the Listing B.11.

```
llvm-readelf-8 -sections sockex_kern.o
```

Listing B.11: llvm-readelf-8

which in turn produces the output shown in the Listing B.12

```
Section Headers:
  [Nr] Name                Type            Address           ...
  ...
  [ 3] socket              PROGBITS        0000000000000000  ...
  [ 4] license             PROGBITS        0000000000000000  ...
  ...
```

Listing B.12: llvm-readelf-8 output

that shows both the sections previously defined in the source code. Furthermore, to read the content of `socket` section, the command in the Listing B.13 can be used.

```
llvm-objdump -no-show-raw-insn -section=socket -S sockex_kern.o
```

Listing B.13: llvm-objdump socket

The output (Listing B.14) is the eBPF bytecode of the program.

```
Disassembly of section socket:
bpf_socket_prog:
        0:          r0 = 0
        1:          exit
```

Listing B.14: llvm-objdump socket output

In this example it consists of just two instructions. The first stores the value `0` into the register `R0`. The second terminates the program. In this case, since a socket program type has been used, a return value of `0` tells to the kernel to discard the packet.

B.1.6 eBPF program loader

An eBPF program in C is first compiled with LLVM into an ELF file (as seen in the Listing B.13), then a loader in the user-space parses the ELF file and loads it into the kernel [53]. Different loaders are available (tc, iproute2, bcc, libbpf, etc.). They differs in the ELF conventions. Currently the use of `libbpf` is suggested [55] and there is an effort to use `libbpf` as the loader standard implementation [54], then it will be used in the examples.

B.1.7 eBPF program verifier

Suppose that the eBPF program shown in the Listing B.6 - without the `return 0;` statement - has been loaded into the kernel using `libbpf` through the user-space frontend program (see section B.1.9 for an example). The eBPF calling convention requires that the `R0` register of the virtual machine contains the return value of an helper function, the exit value for the eBPF program [35] or the return value from an eBPF function for BPF to BPF calls. As a consequence, when the user space code is executed, the verifier shows the error message reported in the Listing B.15 because `R0` is not initialized (e.g. no `return 0;` statement).

```
libbpf: -- BEGIN DUMP LOG ---
libbpf:
0: (95) exit
R0 !read_ok

libbpf: -- END LOG --
```

Listing B.15: verifierreadok

The error message makes clear that the output is generated by `libbpf`. In particular it's the `load_program()` function which is responsible for printing the error message. This function is called by `bpf_program__load()`, which is invoked through a sequence of function calls that starts with `bpf_prog_load()`, the function that loads the eBPF program in to the kernel (more details on `libbpf` in section B.1.8).
The BPF kernel document [35] illustrates other examples of verifier error messages which can be extremely helpful to debug eBPF programs.

B.1.8 libbpf

Last example regarding the verifier introduced the `libbpf` library. It is available in the kernel source tree [48], as a stand-alone version [49] or as a Debian sid package [50].
`libbpf` provides different groups of types and functions. The library facilitates the operations on eBPF object files exposing `bpf(2)` wrappers, "objects" and functions to work with them [49]. Objects and relative functions

are in `libbpf.h`, instead system call wrappers can be found in `bpf.h` [49]. The description of few helpful functions used in the examples follows.

B.1.8.1 bpf_prog_load

Load an eBPF program object file into the kernel

```
int bpf_prog_load(const char *file, enum bpf_prog_type type,
                  struct bpf_object **pobj, int *prog_fd)
```

Where *file* is a pointer to a string that is the name of the eBPF object file, for example `"prog_kern.o"`. *type* is the eBPF program type. *pobj* is a pointer to a pointer to the data structure that represents the ELF object. *prog_fd* will contain the file descriptor associated to the program.

For example, to load an eBPF program object file, a call as reported in the Listing B.16, is performed

```
fd = bpf_prog_load("prog_kern.o", BPF_PROG_TYPE_SOCKET_FILTER,
                   &obj, &prog_fd)
```

Listing B.16: bpf_prog_load socket

Internally `bpf_object__load_xattr()` is invoked, which in turns calls consecutively

```
bpf_object__create_maps();
bpf_object__relocate();
bpf_object__load_progs();
```

Listing B.17: bpf_prog_load calls

therefore this function also handles maps creation before the programs are loaded.

B.1.8.2 bpf_object__find_map_fd_by_name

This function allows to retrieve a map file descriptor by the map's name.

```
int bpf_object__find_map_fd_by_name(const struct bpf_object *obj,
                                    const char *name);
```

Listing B.18: bpf_object__find_map_fd_by_name

For instance if a map is defined in `prog_kern.c` as follows,

```
struct bpf_map_def SEC("maps") my_map = {
        .type = BPF_MAP_TYPE_ARRAY,
        ...
};
```

Listing B.19: my_map

after that the eBPF program named `prog_kern.o` has been loaded into the kernel with `bpf_prog_load()`, it's possible to get the map's file descriptor performing a call, as reported in the Listing B.20.

```
map_fd = bpf_object__find_map_fd_by_name(obj, "my_map");
```

Listing B.20: find_map_by_name

The call combines two functions, in fact internally it invokes `bpf_map__fd(bpf_object__find_map_by_name())`. Therefore first, iterating through all the map sections a reference to the map (or `NULL`) with that map name is retrieved, then its file descriptor is returned.

`map_fd` can be used to interact with the map as shown in the Listing B.21

```
bpf_map_update_elem(map_fd, &key, &value, BPF_ANY);
```

Listing B.21: bpf_map_update_elem

B.1.8.3 bpf_object__find_program_by_title

This function returns a pointer to a program object, given the section name where the program resides within the ELF file.

```
struct bpf_program *bpf_object__find_program_by_title(
        const struct bpf_object *obj, const char *title)
```

Listing B.22: bpf_object__find_program_by_title

Where *obj* is a pointer to the ELF file object and *title* is the name given to the section where the eBPF program resides. For instance, if a program is defined in the `*_kern.c` file as follows

```
SEC("tracepoint/syscalls/sys_enter_clone");
int bpf_tracepoint_prog(struct syscalls_sys_enter_clone_args \
                              *ctx)
{
        ...
}
char _license[] SEC("license") = "GPL";
```

after loading the program into the kernel, it is possible to get a pointer to the program object executing the call shown in the Listing B.23

```
prog = bpf_object__find_program_by_title(obj,   \
                        "tracepoint/syscalls/sys_enter_clone");
```

Listing B.23: bpf_object__find_program_by_title example

B.1.8.4 bpf_program__attach_tracepoint

This function attaches an eBPF program to a tracepoint. In the example reported in the Listing B.23 a pointer to the program object was retrieved. That program can be attached to a particular tracepoint performing the call in the Listing B.24

```
bpf_program__attach_tracepoint(prog, "syscalls", \
                        "sys_enter_clone");
```

Listing B.24: bpf_program_attach_tracepoint example

In this case, every time a call to `clone(2)` is performed, the eBPF program is executed.
The function calls `perf_event_open_tracepoint()` which in turn invokes `perf_event_open(2)` to create the file descriptor relative to the event to be measured. Successively the eBPF program is attached to the tracepoint event and is enabled calling `ioctl(2)` with, respectively,
`PERF_EVENT_IOC_SET_BPF` and `PERF_EVENT_IOC_ENABLE` flags.

B.1.9 Socket filter example

To show the usage of the different components previously introduced, this example illustrates an eBPF program which reads from a map, updated from the userspace code, the number of the port that should be allowed and then compares the port number with the source port of the received UDP packet header. If port numbers are equal the program accepts the packet, otherwise the program discards it. More details are available at the repository.

The `udpmap_kern.c` file contains the eBPF program and the map definition, as shown in the Listing B.25

```
struct bpf_map_def SEC("maps") my_map = {
	.type = BPF_MAP_TYPE_ARRAY,
	.key_size = sizeof(uint32_t),
	.value_size = sizeof(uint32_t),
	.max_entries = 1,
};

SEC("socket")
int bpf_socket_prog(struct __sk_buff *skb)
{
	/* index of the first and only element in the
	 * array map */
	int index = 0;
	/* src port written from userspace in the map */
	uint32_t *srcport_usr = bpf_map_lookup_elem(&my_map,\
			&index);
	if(srcport_usr){
		/* src port from udp packet header */
		int srcport_pkt = load_half(skb, offsetof( \
			struct udphdr, source));
		/* compare port numbers */
		if (srcport_pkt == (*srcport_usr))
			return -1;  // return the entire packet
	}
	return 0; // discard the packet
}
char _license[] SEC("license") = "GPL";
```

Listing B.25: udpmap_kern.c

Initially an array map is defined (lines 1-6). The array map has just one element, since the only value to be inserted is the port number. Every time an UDP packet arrives, through the helper function `bpf_map_lookup_elem()` the value inserted in to the map is retrieved accessing it by an index value of `0` (lines 15-16). It is noteworthy that before to use the `srcport_usr` variable, its content needs to be checked (line 17), otherwise the verifier emits an error message (e.g. `R0 invalid mem access 'map_value_or_null'`, again refer to the kernel documentation [35] for errors explanation). The call to `load_half()` LLVM builtin function loads the UDP header source port field (line 19). At the end port numbers are compared and a proper decision is taken (lines 22-25).

The `udpmap_user.c` file contains the userspace program, as shown in the Listing B.26

```
int main(int ac, char **argv)
{
 /* load the ebpf program object code into the kernel */
 if (bpf_prog_load("udpmap_kern.o",  \
          BPF_PROG_TYPE_SOCKET_FILTER, &obj, &prog_fd))
         return 1;

 /* get the map fd by name */
 map_fd = bpf_object__find_map_fd_by_name(obj, "my_map");
 ...

 /* create socket file descriptor for UDP protocol */
 int fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
 ...

 /* attach the filter */
 assert(setsockopt(fd, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd,
18                          sizeof(prog_fd)) == 0);
19
20  /* bind the local address to the socket */
21  int n;
22  n = bind(fd, (struct sockaddr *)&servaddr, sizeof(servaddr));
23  ...
24
25  /* updates the map with the port number to allow */
26  for (i = 1031; i < 1035; i++) {
27          ret = bpf_map_update_elem(map_fd, &key, &i, BPF_ANY);
28          ...
29  }
30  return 0;
31 }
```

Listing B.26: udpmap_user.c

The userspace codes loads the ebpf program into the kernel (lines 4-5). Then retrieves the file descriptor associated with the map (line 9) and, after attaching the program to the UDP socket (lines 17-18), through a for loop updates the map with the port number which should be allowed next (lines 26-28).

B.1.10 Tracepoint example

The `traceopenat_kern.c` file contains the eBPF program, as shown in the Listing B.27

```
struct syscalls_enter_openat_ctx {
        __u64 pad;
        int __syscall_nr;
        const char * filename;
        int flags;
        unsigned short modep;
```

```
 7  };
 8  SEC("tracepoint/syscalls/sys_enter_openat")
 9  int bpf_tp_prog(struct syscalls_enter_openat_ctx *ctx)
10  {
11          char fmt[] = "Hello\n";
12          int flags = ctx->flags;
13
14          if ((flags & O_RDONLY) == O_RDONLY)
15                  bpf_trace_printk(fmt, sizeof(fmt));
16          return 0;
17  }
18  char _license[] SEC("license") = "GPL";
```

Listing B.27: traceopenat_kern.c

First the tracepoint context struct and the specific section name are defined (lines 1-8), as already described in the eBPF program's context section B.1.4. Every time the `openat(2)` system call is invoked the eBPF program is executed and the *flags* value is checked to verify if `O_RDONLY` flag is set, in that case an helper function which prints a message is called (lines 14-15).
`bpf_trace_printk()` should be used only for debugging purposes because is slow [15] . The output of `bpf_trace_printk()` can be read executing the command illustrated in the Listing B.28.

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

Listing B.28: cat trace_pipe

The `traceopenat_user.c` file contains the userspace program, as shown in the Listing B.29

```
int main(int argc, char **argv)
{
```

```
3        ...
4        /* load the ebpf program object code into the kernel */
5        if (bpf_prog_load("traceopenat_kern.o", \
6                BPF_PROG_TYPE_TRACEPOINT, &obj, &prog_fd))
7                return 1;
8
9        /* get a reference to the eBPF program object */
10       prog = bpf_object__find_program_by_title(obj, \
11               "tracepoint/syscalls/sys_enter_openat");
12
13       /* attach the program to the tracepoint */
14       bpf_program__attach_tracepoint(prog, "syscalls", \
15               "sys_enter_openat");
16
17       /* trigger the eBPF program */
18       fd = open("file.txt", O_RDONLY);
19
20       return 0;
21   }
```

Listing B.29: traceopenat_user.c

First the eBPF program is loaded into the kernel (lines 5-7). Then a reference to the eBPF program is retrieved (lines 10-11) and the program is attached to the proper tracepoint (lines 14-15). At the end to trigger the eBPF program execution a call to `open(2)`, which internally invokes `openat(2)` (as already seen in the seccomp section), is performed.

Appendix C

Packet headers

This Appendix contains the headers structure of the main network protocols to better understand how BPF filters refer to specific fields.

C.1 IPv6

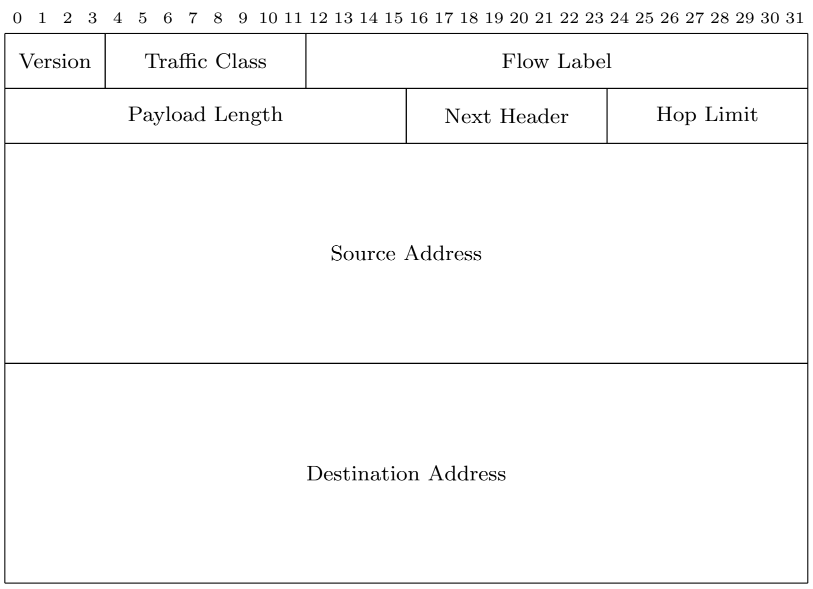

Figure C.1: Header IPv6 [67].

C.2 IPv4

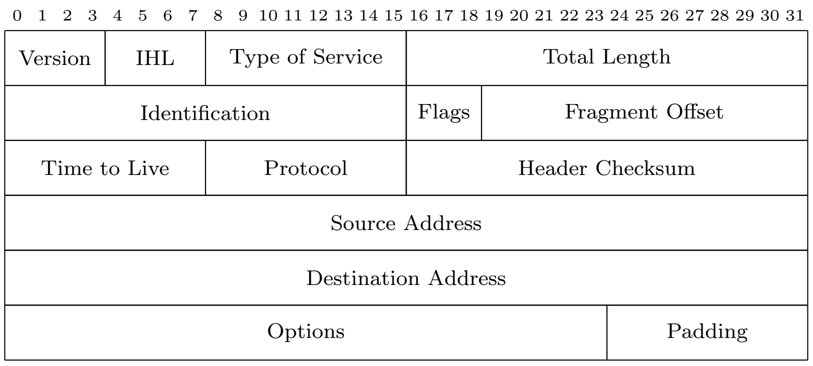

Figure C.2: Header IPv4 [68].

C.3 TCP

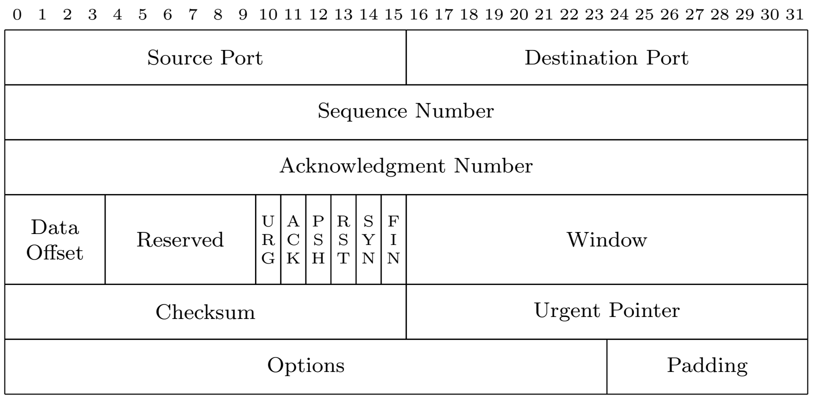

Figure C.3: Header TCP [69].

C.4 UDP

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Source Port	Destination Port
Length	Checksum

Figure C.4: Header UDP [70].

C.5 ICMP - Echo/Echo-Reply

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Source Port	Destination Port
Length	Checksum

Figure C.5: Header ICMP [71].

C.6 Ethernet 802.3

Destination MAC Address	Source MAC Address	Type
6 Byte	6 Byte	2 Byte

Figure C.6: Header Ethernet [72].

Bibliography

[1] S. McCanne, V. Jacobson. *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. Lawrence Berkeley Laboratory, 1992.

[2] `https://events.static.linuxfound.org/sites/events/files/slides/bpf_collabsummit_2015feb20.pdf`

[3] `https://lwn.net/Articles/120647/`

[4] `https://lwn.net/Articles/120192/`

[5] `https://lwn.net/Articles/332974/`

[6] `https://lwn.net/Articles/450291/`

[7] `https://lwn.net/Articles/475019/`

[8] `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`

[9] `https://lwn.net/Articles/738694/`

[10] `https://lwn.net/Articles/796328/`

[11] `https://lwn.net/Articles/599755/`

[12] `http://man7.org/linux/man-pages/man2/open.2.html`

[13] `http://docs.cilium.io/en/latest/bpf/`

[14] Høiland-Jørgensen et al. *The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel.* CoNEXT, 2018.

[15] http://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[16] https://lwn.net/Articles/645169/

[17] https://lwn.net/Articles/751527/

[18] https://qmonnet.github.io/whirl-offload/2017/02/11/implementing-openstate-with-ebpf/#stateful-processing

[19] https://lwn.net/Articles/664688/

[20] https://facebookmicrosites.github.io/bpf/blog/2018/08/31/object-lifetime.html

[21] https://lwn.net/Articles/784571/

[22] http://media.frnog.org/FRnOG_28/FRnOG_28-3.pdf

[23] https://github.com/iovisor/bcc

[24] https://www.netronome.com/blog/bpf-ebpf-xdp-and-bpfilter-what-are-these-thing

[25] https://engineering.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

[26] https://cilium.io/blog/2018/11/20/fb-bpf-firewall/

[27] https://lwn.net/Articles/747551/

[28] https://lists.linuxfoundation.org/pipermail/containers/2018-February/038477.html

[29] https://groups.google.com/forum/#!msg/libseccomp/pX6QkVF0F74/ZUJlwI5qAwAJ

[30] https://www.linuxjournal.com/article/5617.

[31] https://linux.die.net/man/7/raw.

[32] C. Benvenuti. *Understanding Linux Network Internals.* O'Reilly, 2006: 268-269.

[33] https://www.freebsd.org/cgi/man.cgi?bpf(4).

[34] https://elixir.bootlin.com/linux/v2.6.32.71/source/net/core/filter.c

[35] https://www.kernel.org/doc/Documentation/networking/filter.txt.

[36] https://lwn.net/Articles/655028/

[37] http://man7.org/linux/man-pages/man2/seccomp.2.html

[38] https://linux.die.net/man/2/_exit

[39] https://www.kernel.org/doc/Documentation/userspace-api/no_new_privs.rst

[40] http://man7.org/linux/man-pages/man2/prctl.2.html

[41] https://linux.die.net/man/1/strace

[42] http://man7.org/linux/man-pages/man2/fork.2.html

[43] https://github.com/seccomp

[44] https://lwn.net/Articles/494252/

[45] http://man7.org/linux/man-pages/man3/seccomp_release.3.html

[46] https://github.com/seccomp/libseccomp/blob/master/include/seccomp.h.in

[47] https://www.collabora.com/news-and-blog/blog/2019/04/26/an-ebpf-overview-part-3-walking-up-the-software-stack/

[48] https://lore.kernel.org/patchwork/patch/587924/

[49] https://github.com/libbpf/libbpf

[50] https://packages.debian.org/sid/libbpf-dev

[51] http://man7.org/linux/man-pages/man2/bpf.2.html

[52] http://man7.org/linux/man-pages/man8/tc-bpf.8.html

[53] https://docs.cilium.io/en/v1.6/bpf/

[54] https://kinvolk.io/blog/2018/10/exploring-bpf-elf-loaders-at-the-bpf-hackfest/

[55] https://github.com/netoptimizer/prototype-kernel

[56] https://lwn.net/Articles/740157/

[57] https://git.kernel.org/pub/scm/linux/
kernel/git/davem/net-next.git/commit/?id=
2589726d12a1b12eaaa93c7f1ea64287e383c7a5

[58] https://github.com/torvalds/linux/commit/
98b5c2c65c2951772a8fc661f50d675e450e8bce

[59] https://blogs.oracle.com/linux/notes-on-bpf-1

[60] https://lwn.net/Articles/683504/

[61] http://man7.org/linux/man-pages/man7/namespaces.7.html

[62] https://lwn.net/Articles/543273/

[63] http://wiki.virtualsquare.org/#!index.md

[64] https://amslaurea.unibo.it/13184/1/tesi.pdf

[65] https://lists.linuxfoundation.org/pipermail/containers/2018-
February/038527.html

[66] https://lwn.net/Articles/796328/

[67] https://tools.ietf.org/html/rfc2460.

[68] https://tools.ietf.org/html/rfc791.

[69] https://tools.ietf.org/html/rfc793.

[70] https://tools.ietf.org/html/rfc768.

[71] https://tools.ietf.org/html/rfc792.

[72] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 802.3-2008*, 2008: 50-52.

Acknowledgements

I would like to express my deep gratitude to Professor Renzo Davoli, for his passion and advices. I would also like to thank ULISSE lab for the stimulating environment and attitude. My special thanks are extended to qeole for sharing his BPF knowledge, to Mirty which led to my second epiphany, to my colleagues and my flatmates who tolerated my anxiety. Finally, I wish to thank my parents and my sister, for their patience and encouragement throughout my study.