Masaryk University
Faculty of Informatics

XDP test suite for Linux kernel

Bachelor's Thesis

Štěpán Horáček

Brno, Spring 2020

Masaryk University
Faculty of Informatics

XDP test suite for Linux kernel

Bachelor's Thesis

Štěpán Horáček

Brno, Spring 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Štěpán Horáček

Advisor: Ing. Milan Brož, Ph.D.
Consultant: Toke Høiland-Jørgensen, Ph.D.

Abstract

The eXpress Data Path is still a quite young technology, and as such, its testing environment may prove insufficient. Existing test suites may require duplicated code, and yet be too specific to cover multiple use cases. This work introduces test suite, with a test harness providing functionality to solve this problem and several test cases testing the basic capabilities of the eXpress Data Path.

Keywords

eBPF, XDP, testing, network, Python, Linux

Contents

List of Figures

1 Introduction

When dealing with high-volume network traffic, a great deal of importance is posed upon the performance of the system. Many different approaches to increasing the efficiency exist, some of those being early packet filtering, load balancing and monitoring. All of the mentioned approaches can be realized using the *eXpress Data Path* (XDP) framework [1]. XDP, based on *extended Berkeley Packet Filter* (eBPF) [2], gives its users the ability to attach a special program, which decides about how to deal with incoming packets, to predefined locations in the receiving path of the Linux kernel network layer.

Given the fact that XDP implementations for different drivers often differ, it is necessary to test the implementations, to ensure their behaviour is the same. However, currently, the tests present in the Linux kernel attach the XDP programs manually, which results in an increased amount of duplicate code and decreased abstraction. This approach makes it difficult and time-consuming to test multiple XDP modes.

The test suite introduced in this bachelor's thesis creates an abstraction of the hooking mechanism, in order to enable test cases to test multiple XDP modes at once, together with test cases testing correct functionality of basic capabilities of XDP.

The text of this thesis is structured into four chapters. The first chapter describes the tested technology, which consists of XDP and inherently of relevant components of eBPF. The second chapter describes the technology appropriate for building the test suite, such as the *BPF Compiler Collection* [3], Python *unittest* library [4], *Scapy* library [5]. The third chapter describes the structure and implementation of the testing framework and the abstraction of individual XDP modes. The fourth chapter describes the individual test cases. This chapter is divided into sections according to the component being tested, such as responses of programs, context changes or helper functions. The final chapter concludes with a discussion about possible follow-ups and evaluation of the test suite.

The resulting test suite is available online from the GitHub repository[1] under the GNU General Public License.

1. `https://github.com/shoracek/xdp-test-suite`

2 Filtering and tracing technology

The focus in this chapter is on describing the primary target of the test suite – the eXpress Data Path framework. However, before proceeding to describe it in further detail, it is appropriate to introduce extended Berkeley Packet Filter as the underlying technology and its predecessor.

2.1 Extended Berkeley Packet Filter

The extended Berkeley Packet Filter (eBPF) [2] is an in-kernel virtual machine that gives the user the ability to attach an eBPF program from user space into a set of hook points, mostly located in the kernel space. High performance of the programs is achieved by the just-in-time compiler in the kernel [2]. Thanks to its generality, eBPF can be used in many forms, such as tracing, filtering and performance analysis.

An eBPF program is a sequence of instruction with defined type, which specifies helper functions available to the program, structure of the program's context and events to which it can be attached [6].

Before the loading of the eBPF program into the kernel, a verifier checks that the program is safe to run. If the verifier does not reject the program, it is then loaded and can be attached to an event. Every time the event, to which the program is attached, is activated, the program is run, provided with a context specific to the program type containing information about the event. The program then can use maps, a data structure described in subsection 2.1.3, and possibly, depending on its type, change contents of its context and call helper functions [6].

2.1.1 Verifier

To assure that the eBPF program being loaded into the kernel is safe to run, the inserted program is checked before the insertion by the verifier. The verifier checks that the program terminates, does not access unauthorized data and cannot crash. As described in the documentation [2], the verifier works in two passes.

The first pass constructs a directed acyclic graph upon the program instructions and rejects programs containing more instructions than

the BPF_MAXINSNS constant, unbounded loops, unreachable instructions or out of bounds jumps.

The second pass analyses all possible instruction paths in the program. While going through the paths, the verifier keeps track of the types of values stored in registers, by assigning them the types of instructions return values. If the types of instruction arguments and used registers do not match, the program is rejected.

If a program uses direct access to data, the verifier assures that the program checked all accessed data for out of bounds addresses before accessing them.

2.1.2 Helper functions

Since eBPF programs cannot call any function that is not a part of an eBPF program directly, it is provided with helper functions. Helper functions are a set of functions in the kernel, through which eBPF programs can interact with the system, context of the program and its maps.

2.1.3 Maps

In order to keep information between separate runs of the program and to communicate with the user-space program, eBPF programs have access to maps. Maps are data structures that can be created and accessed from user space through system call and accessed from the kernel-space program through helper functions.

2.2 eXpress Data Path

One of the possible uses of eBPF is the eXpress Data Path (XDP). This framework gives its users the ability to insert an eBPF program into the network device driver, before the kernel networking stack starts the processing of the packet. This location allows to make decisions about the incoming packet, even before the kernel creates a socket buffer for it, and therefore can decrease the time spend on filtered out packets [7].

An XDP program is a specialization of an eBPF program, with the type of BPF_PROG_TYPE_XDP. This type most importantly enables

access to XDP hook points, specified by selected mode, and direct packet access.

2.2.1 XDP actions

Decisions about the packet are made through the return value of the XDP program. This value is called *XDP action*. Currently, there are five possible XDP actions: `XDP_PASS`, `XDP_DROP`, `XDP_TX`, `XDP_REDIRECT` and `XDP_ABORTED` [7].

`XDP_PASS` Packets, to which the program responds with `XDP_PASS`, continue further into the network stack.

`XDP_DROP` Packets do not continue further into the stack, and instead, the memory they occupy gets reused for new packets immediately.

`XDP_TX` Packets are transmitted through the device, on which they were received, back into the network. This action, together with changing the destination address in the packet, can be used to resend the packet to another recipient.

`XDP_REDIRECT` Packets are transmitted into a target. This action, in contrast to others, should be returned through calling one of the redirect helper functions, in order to specify the target. The target can currently be either a network device, a CPU or an AF_XDP socket (using a map with the type of devmap, cpumap or xskmap respectively) [1, 7].

`XDP_ABORTED` This XDP action is used to indicate an error in the program and has the same effect on the packets and their subsequent processing as the `XDP_DROP` action. However, additionally, a tracepoint is triggered, which can be used for debugging of the program.

2.2.2 Helper functions

Helper functions exclusive to XDP are focused on packet manipulation, and currently consist of: `bpf_xdp_adjust_head`, `bpf_xdp_adjust_meta`,

`bpf_xdp_adjust_tail`. These functions offer the ability to move data inside the packet and metadata about the packet, in case change is made that requires size change.

Other than the helper functions exclusive to XDP, XDP programs also have access to other helper functions shared with other types of eBPF programs.

2.2.3 Operation modes

Depending on the driver support and users initiative, an XDP program can be run in several modes. Each mode specifies the location of the hook point and brings certain advantages and disadvantages described below [7].

Native mode This is the intended mode of XDP programs, as it allows to make use of the speed increase XDP is supposed to offer. The hook point for this mode is located in the network interface driver, which means that implementation of XDP in the driver is required for usage of this mode[1].

Offloaded mode A subgroup of native mode is an offloaded mode. This mode differs from the native mode by running XDP programs directly on the network interface card. This frees the processor of having to attend to an incoming packet that would have been dropped[2].

Generic mode Due to the fact that the hook point resides further in the networking stack, and therefore the attached program gets activated after the socket buffer was created, it does not offer the performance advantages of native XDP. However, it is found in every Linux kernel since version 4.12 [8] regardless of the hardware setup, which allows it to be used for testing XDP programs without the need for supporting drivers.

---

1. List of drivers currently supporting this mode can be found on `github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp`
2. As of the time of writing, the only driver supporting this is Netronome's `nfp` driver [7].

`BPF_PROG_TEST_RUN` Even though not being operation mode per se, the `BPF_PROG_TEST_RUN` offers another way to run an XDP program. This operation does not attach the program to a real hook point and instead simulates run of the program with arbitrary context. After the dry run finishes, it provides context transformed by the program and its chosen XDP action.

2.3 Probes

Probes are a mechanism offering a point for inserting a tracing program into a code. Introduction of two types of probes follows.

Tracepoints are statically defined probes, offering a stable API. They need to be defined by the programmer before the compilation of the probed code and require updating if the code changes [9]. Kprobes, on the other hand, can be attached to any kernel function or instruction. This comes at the price of not having a predefined interface [10].

Probes are used in the testing harness to analyse functions called by the XDP. The usage is described further in subsection 4.2.2.

3 Technologies relevant to the implementation

In this chapter, technologies relevant to the test suite implementation are described. Each section focuses on the parts of the technology, that are used by the test suite, with details about the usage itself described in chapter 4. Each section is, if not said otherwise, based on information from the source provided at the start of it.

3.1 unittest

The *unittest* library [4] is a unit testing framework and a part of the Python standard library. In this section, its most important concepts and methods are described.

Test case In *unittest*, test cases are implemented using the subclasses of `TestCase` and its test methods.

Test loader Before the tests can be run, they need to be found. This job is done by the test loader. The default test loader identifies subclasses of `TestCase` and their methods with a prefix `test` as tests. Tests can be found by either specifying a starting folder, module, `TestCase` subclass or using a string with module, class and method separated by dots.

Test runner After the tests are loaded, they can be run. Responsible for this is the test runner. The test runner, other than running the tests, also provides an output, by default in text format outputted into the standard error output stream.

Test fixture Each test case class can specify its test fixture methods. These methods are run before or after a test and are used to prepare for its execution or clean after it. There are two pairs of test fixture methods in *unittest*, one that runs for every test method (`setUp` and `tearDown`) and one that runs for every test class (`setUpClass` and `tearDownClass`).

Assert methods In order to verify the results and states of tested components, an assortment of assert methods are available, such as `assertEqual`, `assertIn`, or `assertTrue`[1]. By default, failure of an assert method stops the current method and test runner continues with the following test if there is any.

Skipping tests and expected failures Since not every test is expected to succeed, or even be able to be run under every condition, *unittest* offers decorators to indicate such tests, may they be only a single method or an entire class. These decorators are `skipIf`, which skips the decorated test if the given condition is fulfilled, its opposite `skipUnless`, `skip`, skipping every time, and `expectedFailure`, which leaves the method to be run as usual, but with the test passing regarded as a failure and a failing as a success.

Since decorators are resolved on import time, if they are given an argument, its value must be known before importing the module containing tests.

3.2 Scapy

Scapy [5] is a packet manipulation library for Python, providing functions to create and send packets and listen to the network traffic.

3.2.1 Sending and receiving packets

In order to send packets into the network, *Scapy* uses multiple functions, notable two are: `send` and `sendp`, where the former sends packets at the network layer and the latter sends packets at the link layer.

Capturing traffic on a network interface in *Scapy* is mediated by the `AsyncSniffer` class and its wrapper function `sniff`. It should be noted that both of these approaches capture not only incoming traffic but also outgoing. This is due to the fact that *Scapy*'s sniffing does on Linux by default use the packet socket. Packet socket is a type of socket that allows sending and receiving packets at the link layer and,

1. Full listing can be found at `docs.python.org/3/library/unittest.html#assert-methods`.

if not set otherwise, on reading gives both incoming and outgoing packets [11].

Other than functions for purely sending or capturing packets, *Scapy* also offers a function to combine these two. These functions are `sr`, `srp`, `sr1`, `srp1`, with the difference between these four functions being the layer at which they are sent and received and whether to keep only the first answer to a packet.

3.2.2 Packet classes

Even though packet sending functions also accept binary representation of packets, for readability and modifiability, *Scapy* provides classes representing headers of individual protocols. These header classes can be then concatenated into packets. While using these classes, it is not required to specify every parameter of each header, since before sending, *Scapy* fills certain missing parameters such as checksum, header length, or source address.

3.3 BPF Compiler Collection

BCC (BPF Compiler Collection) [3] is a toolkit for compiling, loading and attaching eBPF programs. It offers easy integration into other languages like C++, Lua, Python thanks to its front-ends.

In the following subsections, the Python front-end of *BCC* is described.

3.3.1 Compiling eBPF program

In order to attach an eBPF program, the program must be first compiled. This can be accomplished by creating a BPF object. The constructor requires a parameter determining an XDP program, specified as a string either of the definition or the location of the program, and also contains an optional parameter containing flags to be passed to the compiler, such as macro definitions. The returned object can then be used when referencing to the program.

```
int bpf_prog_test_run(
  int prog_fd, int repeat,
  void *data, __u32 size,
  void *data_out, __u32 *size_out,
  __u32 *retval, __u32 *duration
)
```

Figure 3.1: Signature of `bpf_prog_test_run` function

3.3.2 Attaching compiled program

After compiling an eBPF program, it can be then attached. *BCC* supports many different hook types, but for the test suite implementation, the three most important are the XDP, kprobe and tracepoints hook points introduced earlier.

XDP provides several tracepoints in order to monitor XDP programs attached to interfaces. There exist several methods to get their listing, such as using the `/sys/kernel/debug/tracing/event/XDP` directory of the sysfs mechanism.

3.3.3 `BPF_PROG_TEST_RUN`

BCC does not cover every function surrounding eBPF in Linux kernel in native Python and instead uses Python's built-in `ctypes` foreign function library. One of those functions is `bpf_prog_test_run`, offering access to the `BPF_PROG_TEST_RUN` command of the `bpf` system call. This command has the ability to run an eBPF program with arbitrary input, and as such, it is useful in debugging created programs.

3.4 Pyroute2

Pyroute2 [12] is a Python network configuration library. By using this library, it is possible to inspect and change the configuration of network interfaces on the local machine.

The *pyroute2* library provides two approaches to the network interface manipulation. High-level database approach and a lower-level approach mediated by the IPRoute class.

In this section, the text is focused on the IPRoute class.

3.4.1 IPRoute class

IPRoute class uses RTNL[2] API [13] in order to communicate with the kernel. Thanks to its close mapping of the RTNL API, this class offers an API similar to that of the often-used iproute2 utility, providing easier usage by those familiar with it.

In order to configure or inspect a single interface, two methods are available: the `link` method for the second layer and the `addr` method for the third layer. The first parameter of those methods specifies the command and the keyword parameters depending on the type of command.

The `link` method allows manipulation of the interface on a second layer, using the following commands: `"add"` for creating new interfaces, `"delete"` for removing existing interfaces, `"set"` for setting attributes of existing interfaces, `"dump"` to get a list of all interfaces, `"get"` to get a single interface.

The `addr` method allows to add and remove IP addresses, using either the command `"add"` or `"delete"` and keyword parameters for the index, address and mask.

Network namespace support is provided by the NetNS module. This module implements a `NetNS` class implementing the same API as the IPRoute class, differs from it the by implicitly using the network namespace different from the default namespace.

2. Stands for Routing Netlink, a socket for communication between kernel and user-space application. Enables configuring and gathering information about interfaces.

4 Test harness implementation

In this chapter, the implementation of the test harness is described, starting with the goals of the harness, followed by an overview of the harness's structure and its parts and finished by the description of the parts.

The goal of the harness is to provide an abstract environment for tests to run XDP programs in. This is done to enable tests using this harness to assess implementations of XDP in drivers or to test the XDP programs themselves. It does so by providing tools for black-box testing, with everything between the point for sniffing and sending packets acting as the black box. However, what the harness is not trying to provide are the tools for testing the performance of XDP or programs using it, or their formal verification.

The standard structure of the harness consists of a client and servers. The client acts as a test runner, whereas the servers are communication partners for the client, providing the client with incoming packets and gathering the outgoing packets. Servers are used instead of merely sniffing the packets so that the testing can also cover XDP in physical network interface cards. Servers are connected to the client by a pair of interfaces. Each of the pairs consists of one testing interface and one communication interface. The testing interface should be used only for packets used by tests, the communication interface, on the other hand, is used for synchronisation of the client and the server and does not have to be used exclusively for the synchronisation, but may be used for other traffic. This enables the communication interface to be connected to a network or shared for multiple servers. A server can be either a physical server, running on a machine different than the one the client is running on, or a virtual server, running inside a network namespace. In both cases, the implementation of the server is the same. The purpose of a virtual server is to provide additional server in case there are not enough physical interfaces or to use for testing virtual interfaces. One of the servers is designated as the main server. This server is then the one that sends packets to the client by default.

An example of a structure of the harness is shown in diagram 4.1. This diagram depicts an example configuration of the test harness,

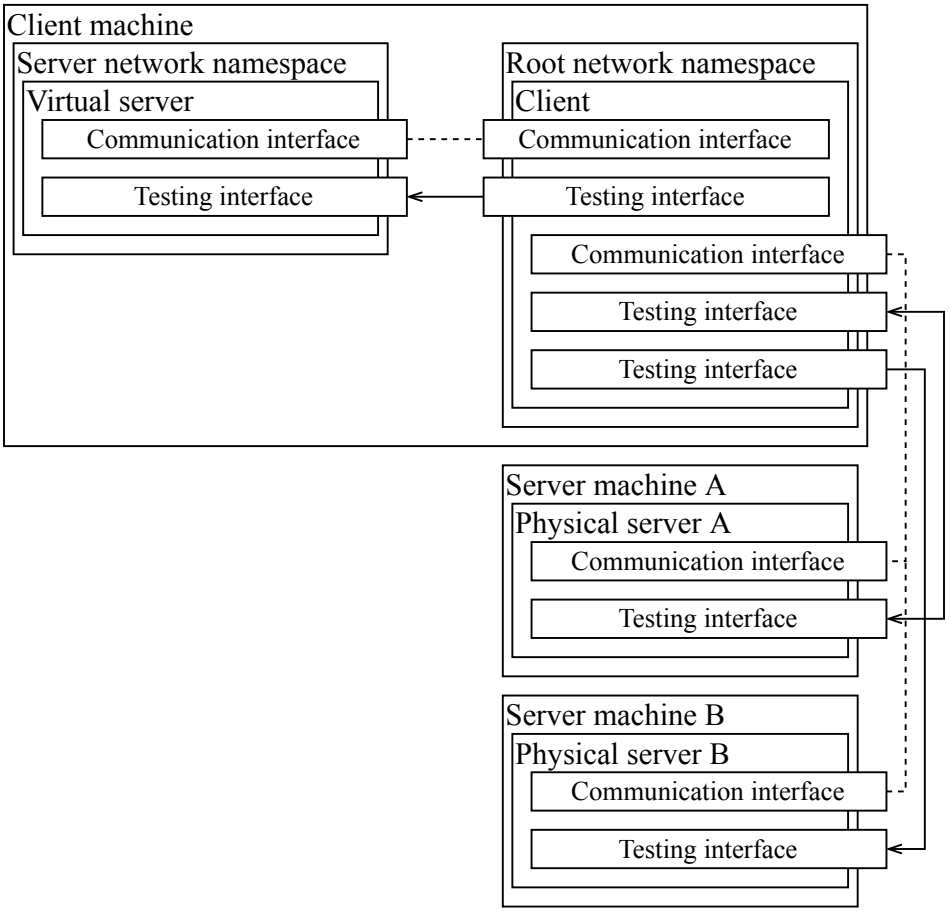

Figure 4.1: Diagram of the harness

running in the root network namespace, consisting of one virtual server, two physical servers and a client. The client is connected by a pair of interfaces to each of the servers. The connection between testing interfaces, depicted by a solid line, are separate for each server, the connection between communication interfaces, depicted by a dashed line, may be shared between multiple servers. Possible flows of packets between testing interfaces are denoted by arrows. In this case, it can be seen that the physical server A, acting as the main server, does both send packets to the client and can receive packets back, compared to the other server that can only receive packets and should not send any. What can also be observed is that, for the client, the manner of communication does not differ between a virtual server and a physical server, and both are interchangeable.

4.1 Server

The server acts as a communication partner for a client. Servers are independent of any specific client instance and can stay running through multiple client lifetimes. Each server is defined by its context. The context of a server is defined in the configuration file and consists of two interfaces, with which it is connected to a client: the testing interface, used for sending packets that are a part of a test, and the communication interface running a TCP server, that is used to synchronise with a client.

Before the start of the server, the testing interface used by the server is configured to attempt prevention of arrival and generation of packets unrelated to testing. This is done by disabling several functionalities of the kernel networking stack. Using the `sysctl` command, certain attributes are changed, such as `mldv2_unsolicited_report_interval`, `mcast_solicit`, or `accept_ra`.

When the server is started, it waits for a client to connect to it. After a client establishes a connection with the server, the client can then request the server to either send information about its testing interface, send a list of packets through the testing interface, or to capture any packet arriving into the testing interface.

4.1.1 Interface attributes sharing

Sharing information about the testing interface is useful so that the client can create packets containing the correct addresses of the server's testing interface since the server's interface could discard the incoming packets otherwise. Since auto-configuration is disabled on the testing interfaces, the client may not acquire this information the usual way.

4.1.2 Sending testing packets

The client may provide a server with packets and request the server to send them out through the testing interface. After receiving this request, the server firsts starts an asynchronous sniffer on the testing interface and afterwards sends the packets, using *Scapy*'s `AsyncSniffer` and `sendp`. Since *Scapy*'s sniffing by default contains both incoming and outgoing due to its configuration, in order to sniff only the outgoing packets, the `AsyncSniffer` is given a custom wrapper around *Scapy*'s socket class. This wrapper activates an option to ignore the outgoing packets. After sending all the testing packets, the server stops the sniffer, notifies the client about finishing and sends the sniffed packets.

4.1.3 Listening for testing packets

The mechanism of listening is similar to the one described in the previous subsection, with the difference being that sniffing is stopped after receiving a command from the client, instead of when all packets are sent out. This is useful when using the interface connected to the server as a target for redirecting.

4.2 Client

The primary task of the client is to execute test cases. This is done using the `XDPCase` class for implementing the tests and *unittest* for running these tests.

A client is configured by using a list of contexts. Each context contains a definition of a local interface and a definition of a remote interface connected to it or a communication interface of a server. The

configuration can also specify virtual servers to be created. Virtual servers are defined using both context of the client and the server.

The abstract class `XDPCase` provides an interface adjusted for testing of XDP. This is done primarily by providing method `attach_xdp` for selecting XDP function to test with, and method `send_packets` process packets by the selected XDP function. Results of `send_packets` are provided as lists of packets that arrived or departed from each interface. Other than these, there are also helper methods for getting the attributes of testing interfaces, used for building and verifying packets, a method for generating packets, that can be used when the content of packets does not matter for testing, and assert methods for packets and packet containers, providing an output adapted for easier readability of packet header attributes.

Currently, there are two modes the client can run in: the network mode, testing XDP on interfaces with the help of servers, and the offline mode, testing XDP using a special command.

4.2.1 Network mode

Implementation of `XDPCase` for this mode provides `XDPCaseNetwork`. Method `send_packets` uses servers in order to receive packets on a client interface or get packets transmitted from the client interfaces to server interfaces. While using this mode, the XDP mode to be tested is specified in the configuration, and as long as the mode is supported by the driver, the testing process remains the same.

Before running the tests, the environment needs to be prepared. This is done in three phases. In the first phase, the testing interfaces are configured so that no packets unrelated to the testing arrive. This is done the same way as done by the server in section 4.1. The second phase creates requested virtual servers. This is done by creating a network namespace, connected by virtual ethernet pairs to the namespace the client is running in, for each server. The final phase acquires interface attributes from the remote servers. The reason for this is that with disabled functionalities of the networking stack, auto-configuration might not be possible. Other than filling missing information, this also verifies whether servers are ready to be used by the tests.

In order to log packets that both arrived at the local machine and those that were redirected into a server, a sniffer is run locally and

a command to sniff is sent on non-main servers. After the sniffer initialisation is done, the packets are then relayed to the main server to be transmitted. When the packets are sent-out, the server sends a message to the client. Subsequently, the sniffers are stopped, all the sniffed packets are collected and returned.

Thanks to the fact that packets are delivered to the interface from outside and may be processed by the entire network stack, this mode may be used to test all the current XDP modes.

4.2.2 Offline mode

Implementation of `XDPCase` for this mode provides class `XDPCaseBPTR`. This mode uses a function `bpf_prog_test_run`, introduced in section 3.3.3, to simulate running of an XDP program on an interface.

Using this mode offers a baseline for testing, since the function `bpf_prog_test_run` provides only a wrapper around the execution of the program, without engaging the rest of the network stack. This minimalises the risk of a bug in driver implementation failing the test.

As seen in figure 3.1, `bpf_prog_test_run` provides a return value of the XDP program, the final state of packet and length of the program execution. Unfortunately, using only these alone, it is not possible to extract all the information required to make `XDPCaseBPTR` keep the same interface for sending packets as `XDPCaseNetwork`. One missing piece of information is the target of redirection. A tracing program is used to fill this information. Sadly, `bpf_prog_test_run` does not trigger required tracepoints and so kprobes are used. An eBPF program is attached using kprobes to functions `bpf_xdp_redirect` and `bpf_xdp_redirect_map`. When activated, the program saves information about the target from the function into a map. After each execution of `bpf_prog_test_run`, if the program was activated, the maps are checked and with the use of the information acquired this way the packet is treated as if it would arrive at the specified target. That means, if for example `bpf_xdp_redirect` or `bpf_xdp_redirect_map` with a device map was used, the packet is treated as if it would depart from the target interface.

Compared to the network mode, there is no need for using servers, creating virtual interfaces or making sure no third-party packet arrives at the testing interface since all the testing is done offline. On the other

hand, due to the fact that this mode in many ways substitutes the driver, new bugs may arise, and the scope of the features may not be complete.

5 Tests cases

This chapter describes the standard structure of a test in this test suite, shows an example of such a test and lastly describes the test cases implemented.

Before using the method `send_packets`, test cases should call both methods `attach_xdp`. If a test case uses an external loading mechanism, it can specify so with a `usingCustomLoader` decorator, and it will be skipped if it is not compatible with the testing mode.

An example of how a typical test might look like is shown in 5.1. Presented is a simple test class, consisting of three methods.

`setUpClass` The first method is the class method `setUpClass` on line 3. Beginning with the necessary call to the parent's overridden method, what follows are time-consuming operations, due to this method being called only once. In this case, this is the creation of packets and the compilation of the XDP program to be used. Since the content of the packets does not matter for this tests, generation of packets on line 6 uses `generate_default_packets`, a method of `XDPCase` that returns several unspecified packets, containing the correct addresses. After that, on line 8, the source file containing the XDP programs is compiled. This file specifies XDP programs `pass_all` and `drop_all`, both consisting of only one statement. Program `pass_all` contains a statement to return `XDP_PASS` and `drop_all` contains a statement to return `XDP_DROP`.

`test_pass_all` Next method is a test method `test_pass_all` on line 10. This method verifies that while an XDP program, always returning `XDP_PASS`, is attached to the testing interface, all packets are received. As a first thing, the program is attached, on line 11. Since the program was already selected when it was compiled, all that is left is to select the program. In this case that is `"pass_all"`. After attaching the program, the packets can be sent. This is done on line 13 by calling `send_packets` with the earlier generated packets as an argument. This method returns a `SendResult` object, containing variables `captured_local` containing packets captured on clients interface and `captured_remote` containing packets captured on interfaces of servers.

```
1  class ReturnValuesCase(XDPCase):
2    @classmethod
3    def setUpClass(cls):
4      super().setUpClass()
5
6      cls.packets = cls.generate_default_packets()
7
8      cls.prog = cls.load_bpf("progs/example.c")
9
10   def test_pass_all(self):
11     self.attach_xdp("pass_all")
12
13     result = self.send_packets(self.packets)
14
15     self.assertPacketsIn(
16       self.to_send,
17       result.captured_local
18     )
19     for i in result.captured_remote:
20         self.assertPacketContainerEmpty(i)
21
22   def test_drop_all(self):
23     self.attach_xdp("drop_all")
24
25     result = self.send_packets(self.packets)
26
27     self.assertPacketContainerEmpty(
28       result.captured_local
29     )
30     for i in result.captured_remote:
31         self.assertPacketContainerEmpty(i)
```

Figure 5.1: Example of test case code

The last thing left to be done is to check whether the packets arrived at the correct interface unchanged, this is done on line 15 to 18, and to make sure none of the servers received any packets from the client, this is done on lines 19 and 20.

`test_drop_all` The last method is also a testing method and is similar to the previous one, with the difference being that now, the XDP program instead of letting the packets continue, it drops them all. Changes in this methods are located on lines 23, where the XDP program `"drop_all"` is used, and on lines 27 to 29, where instead of checking whether all packets arrived, it is checked that no packets arrived at the client's interface.

5.1 Response verification

In this subsection, tests testing the correct response of XDP implementation to the XDP program's return value are described.

Implemented here are tests for the all current possible return values of an XDP program, excluding the `XDP_REDIRECT`. That is `XDP_PASS`, `XDP_DROP`, `XDP_ABORTED`, and `XDP_TX`. The test for `XDP_REDIRECT` is excluded, since calling it directly is not a correct usage, and one of its two helper functions should be used instead. These helper functions are described in the following section 5.2.

Testing of most of the return values is, thanks to the testing harness short and straightforward. All tests in this subsection differ only in used XDP programs and the interfaces to which assertions expect the packets arrived. The tests are implemented in the class `ReturnValuesBasic`. Program of a test of each return value consists of only a return instruction, with the value dependent on the test. Methods implemented in this class are similar to those depicted in figure 5.1.

5.2 Helper functions

In this section, tests verifying the correct functioning of selected helper functions, introduced in subsections 2.1.2 and 2.2.2, are described. The section is divided into two parts. The first part describes helper

functions used for redirecting packets, and the second part describes helper functions for changing the size of a packet.

5.2.1 Redirecting of packets

Two helper functions for redirecting packets are tested: `bpf_redirect` and `bpf_redirect_map`. The implementation of these tests is divided between two classes. The class `HelperFunctionsRedirectToDevice` contains tests for redirecting into a network interface and the class `HelperFunctionsRedirectToCPU` contains tests for redirecting into a CPU.

Redirecting into a device The testing of the two helpers, used for redirecting packets, `bpf_redirect` using an index of the device, and `bpf_redirect_map`, using a device map, is accomplished by the class `HelperFunctionsRedirectToDevice`. For `bpf_redirect_map`, the testing is done by using a map that is changed from the user space program. For `bpf_redirect`, the target is chosen by a macro definition, specified while compiling the program, instead of a map, in order to decrease the risk of a bug with map breaking this test.

The structure of this test class can be seen in figures 5.2 and 5.3. These two code snippets show a testing class focused on the redirection of packets. The structure of the class is similar to the one introduced earlier in figure 5.1, with some changes, described in the following text.

The first snippet from figure 5.2 starts with a *unittest*'s `skipIf` decorator. This decorator causes the test runner to skip the class if the number of the servers available to the context is lesser than required, in which case the test case would fail. On line 7 starts the initialisation of the testing class using the method `setUpClass`. The content of this method is similar to the example case's one, with the difference of saving a target variable determining the interface into which the received packets are going to be redirected and defining a value of a macro in for the compilation, containing the target on top of that. The target is specified using two variables, one containing the index in the harness's context and the other one the index of the device used by the kernel.

```
@unittest.skipIf(
  XDPCase.get_contexts().server_count() < 2,
  "Requires a second server to redirect to."
)
class HelperFunctionsRedirectToDevice(XDPCase):
  @classmethod
  def setUpClass(cls):
    super().setUpClass()

    cls.target = 1
    cls.target_index = cls.get_contexts() \
        .get_local(cls.target).index
    cls.prog = cls.load_bpf(
        b"progs/helper_functions.c",
        cflags=["-DREDIRECT_TARGET=" +
                str(cls.target_index)]
    )
    cls.to_send = cls.generate_default_packets()
```

Figure 5.2: Snippet of redirection test – initialisation

```
20    def test_redirect_to_device(self):
21      self.attach_xdp("redirect_to_const")
22      self.check_result(
23        self.send_packets(self.to_send)
24      )
25
26    def test_redirect_map_to_device(self):
27      self.attach_xdp("redirect_to_devmap")
28      self.prog[b"device_map"][0] = ctypes.c_int(
29        self.target_index
30      )
31      self.check_result(
32        self.send_packets(self.to_send)
33      )
34
35    def check_result(self, result):
36      arrived_local = result.captured_local
37      arrived_remote = result.captured_remote
38
39      self.assertPacketsIn(
40        self.to_send, arrived_remote[self.target]
41      )
42      self.assertPacketContainerEmpty(
43        arrived_local
44      )
45      for i in arrived_remote[:self.target] \
46          + arrived_remote[self.target + 1:]:
47        self.assertPacketContainerEmpty(i)
```

Figure 5.3: Snippet of redirection test – testing methods

```
1 int redirect_to_const(struct xdp_md *ctx) {
2         return bpf_redirect(REDIRECT_TARGET, 0);
3 }
4
5 BPF_DEVMAP(device_map, 1);
6 int redirect_to_devmap(struct xdp_md *ctx) {
7         return device_map.redirect_map(0, 0);
8 }
```

Figure 5.4: Programs used by redirection test

The second snippet, shown in figure 5.3, consists of testing methods `test_redirect_map_to_device` and `test_redirect_to_device` and the method `check_result`. The first testing method in this class, named `test_redirect_to_device` and starting at line 20, tests the functionality of the XDP's redirection helper `bpf_redirect`, the second testing method `test_redirect_map_to_device`, starting at line 26, tests the functionality of the helper `bpf_redirect_map`. The structure of the two testing methods of this class is nearly identical, except for setting the correct target in the device map on line 29. Compared to the test example's testing method, these classes also differ in the assertion part. The assertion is this time extracted into the `check_result` method on line 35 since the same assertion is shared between the two testing methods. The difference is not only in the separation of assertions but also in the location where the targets were expected to arrive. This time, all the packets were expected to arrive at the targeted server.

The XDP program used by the earlier introduced tests is presented in figure 5.4. This figure consists of two functions `redirect_to_const` and `redirect_to_devmap`. Function `redirect_to_const` redirects all packets that arrive at the device with index determined by the value of the `REDIRECT_TARGET` macro, defined while the program is being compiled. The other function `redirect_to_devmap` redirects arrived packets to the index in the first value of the `device_map` variable, that is set after attaching the program.

Redirecting into a CPU The class `HelperFunctionsRedirectToCPU` tests only helper function `bpf_redirect_map`, using a CPU map, since `bpf_redirect` can target only network interfaces. The test itself is similar to the one described in `HelperFunctionsRedirectToDevice`, with the difference being that the assertions expect the packets to arrive at a local interface instead of to a remote interface.

5.2.2 Packet size modification

The helper functions enabling the changing of packet size tested are `bpf_xdp_adjust_head` and `bpf_xdp_adjust_tail`. Both of these tests are implemented in test case `HelperFunctionsAdjustSize`.

The correct functioning of these helper functions is tested by attaching an XDP program changing the size of the packet and passing it, sending packets to the client and ensuring that the size of arrived packets has been correctly changed.

Both for changing the head and the tail of the packet, there are two tests for each: one test for increasing the overall size of the packet and the other for decreasing the size. As of now, the tests for increasing size are marked as expected failures, since they are not yet implemented in XDP.

6 Conclusion

This thesis introduced and described tracing technology extended Berkeley Packet Filter and filtering technology eXpress Data Path. Also introduced were libraries used in the implementation, *unittest*, *Scapy*, *BPF Compiler Collection* and *Pyroute2*. Finally, a test harness and tests were implemented as the practical part of the thesis. This test harness is capable of testing both XDP implementation and XDP programs using either network interfaces or the command `BPF_PROG_TEST_RUN`. The tests are focused on the basic capabilities of XDP, such as verifying the response to the returned value of an XDP program, or proper behaviour of the packet size modification and redirecting helper functions, were implemented as the practical part of the thesis.

6.1 Evaluation

The test suite has already proved itself to be useful. Using this test suite, a bug in the implementation of XDP in the Linux kernel was discovered. This bug caused XDP programs attached to a veth interface using the generic mode to be ignored since programs using generic mode were not executed on cloned packets. Thanks to the cooperation with upstream Linux developers at Red Hat in verifying this bug, this bug has been fixed[1], and the fix is included in the Linux kernel since version 5.5 and it has also been backported into stable versions 4.19 and 5.4.

However, in its current state, the test suite can sometimes give false results. This is usually caused by services running on the machines and sending packets through the testing interface. One of such services is Avahi [14], a service providing DNS discovery, which can be disabled so that it does not send configuration packets using testing interfaces.

6.1.1 Future work

There are several possible follow-ups for this project. One of the possible follow-ups is an introduction of a method to remove the require-

1. Commit of this fix can be found at `git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ad1e03b2b3`.

ment for every server to be connected by two interfaces. A possible solution is to introduce a time multiplexing. The testing interface would be used for both synchronisation and traffic. This solution would, however, not only increase the time required for tests to run, due to the necessary timeout for every packet load but also might hide issues with the server as issues with XDP. Another possible extension of the test harness is to introduce ways to compile and attach an XDP program, other than *BCC*, for example by using the CO-RE (Compile Once – Run Everywhere) project. Finally, in order to keep the test harness up-to-date, it is necessary to follow new features of XDP, and if needed, update the test harness to support them. One of such features might the currently under development `BPF_XDP_EGRESS` program type for eBPF, enabling the filtering of outbound packets.

Bibliography

1. HØILAND-JØRGENSEN, Toke; BROUER, Jesper Dangaard; HERBERT, Tom; BORKMANN, Daniel; FASTABEND, John; AHERN, David; MILLER, David. The eXpress data path: fast programmable packet processing in the operating system kernel. In: *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. ACM, 2018. Available from DOI: 10.1145/3281411.3281443.
2. SCHULIST, Jay; BORKMANN, Daniel; STAROVOITOV, Alexei. *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*. 2019. Available also from: kernel.org/doc/Documentation/networking/filter.txt.
3. IOVISOR. *BCC source code repository* [online]. 2020 [visited on 2020-05-17]. Available from: github.com/iovisor/bcc.
4. PYTHON SOFTWARE FOUNDATION. *unittest — Unit testing framework* [online]. 2020 [visited on 2020-05-17]. Available from: docs.python.org/3/library/unittest.
5. BIONDI, Philippe; THE SCAPY COMMUNITY. *Scapy's documentation* [online]. 2020 [visited on 2020-05-17]. Available from: scapy.readthedocs.io.
6. KERRISK, Michael; STAROVOITOV, Alexei. *BPF(2) Linux Programmer's Manual*. 2018. Version 4.16. Available also from: man7.org/linux/man-pages/man2/bpf.2.html.
7. *BPF and XDP Reference Guide* [online]. Cillium Authors, 2019 [visited on 2020-04-27]. Available from: cilium.readthedocs.io/en/latest/bpf.
8. MILLER, David Stephen. *net: Generic XDP* [online]. 2017 [visited on 2020-04-24]. Available from: git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.
9. DESNOYERS, Mathieu. *Using the Linux Kernel Tracepoints*. 2019. Available also from: kernel.org/doc/Documentation/trace/tracepoints.txt.

10. KENISTON, Jim; PANCHAMUKHI, Prasanna S.; HIRAMATSU, Masami. *Kernel Probes (Kprobes)*. 2019. Available also from: `kernel.org/doc/Documentation/kprobes.txt`.

11. KERRISK, Michael; BORKMANN, Daniel; BRUIJN, Willem de; PRÉVOT, David. *PACKET(7) Linux Programmer's Manual*. 2017. Version 4.16. Available also from: `man7.org/linux/man-pages/man7/packet.7.html`.

12. SAVELIEV, Peter V. *Pyroute2 netlink library* [online]. 2013 [visited on 2020-05-17]. Available from: `docs.pyroute2.org`.

13. KERRISK, Michael; ØSTERGAARD, Bryan; GÍSLASON, Bjarni Ingi; EMELYANOV, Pavel. *RTNETLINK(7) Linux Programmer's Manual*. 2020. Version 4.16. Available also from: `man7.org/linux/man-pages/man7/rtnetlink.7.html`.

14. POETTERING, Lennart; LLOYD, Trent; SIMONS, Sjoerd. *Avahi* [online]. 2020 [visited on 2020-05-20]. Available from: `avahi.org`.

A Code structure

A.1 Harness

The implementation of the harness consists of files located in the folder `harness`, and the two files `run.py`, for parsing the user input, and `config.py`, a configuration file containing contexts used while testing, located in the root folder. While writing new tests or while running the tests, it is enough to only interact with the files in the root folder, whose usage is described in appendix B. The remainder of the files, located in the `harness` folder, contains several files used in the implementation. Those files are `setup.py`, containing functions for creating and configuring interfaces and network namespaces, `config_virtual.py`, providing function for creating virtual servers, `xdp_case.py`, containing implementations of the `XDPCase` class and its subclasses, `bptr_probe_counter.c`, used by `XDPCaseBPTR`, `context.py`, containing files for configuring and sharing information about interfaces, `client.py` and `server.py`, containing implementations of the client and the server part, and `utils.py` for small utilities.

A.2 Tests

The implementation of the tests is divided between two folders. In the folder `tests` the implementation of the tests are located and in the folder `progs` the source files of the XDP programs used tests are located. The `tests` folder currently contains a `test_general.py` file and `test_external.py`. The file `test_general.py` includes tests focused on the basic capabilities of the XDP, such as tests introduced in the chapter 5. The file `test_external.py` contains an example of a test case that does not use the attaching method of the test harness and instead uses an external script to attach the program.

B Usage and extension

B.1 Requirements

In order to run the test harness, *Python 3.5*, *BCC*, *Pyroute2* and *Scapy* are required to be installed.

B.2 Testing

B.2.1 Running

To start the test suite, start `./run.py` as a superuser. There are three commands that can be used:

`client` Start a client, running tests using network interfaces to process packets by an XDP program. One can further specify which tests to run, using `unittest`'s format. That is modules, classes and methods separated by dots. To run only the tests in `ReturnValuesBasic` class in the `test_general.py` file, one can start the test harness by calling `./run.py client test_general.ReturnValuesBasic`.

`bptr` Similar to the `client` command, with the difference that it uses the `BPF_PROG_TEST_RUN` syscall command instead of a server to process packets by an XDP program.

`server` Starts a server, used by `client` command to send packets.

B.2.2 Configuration

Configuration of interfaces to be used for testing is done in the file `config.py`. In the configuration file there are two variables:

`local_server_ctx` A variable specifying the interface of the server, used for testing, and the interface of the server used for communication with a client. An example of a configuration of a server follows:

```
local_server_ctx = ContextServer(
  ContextLocal("enp0s31f6"),
  ContextCommunication("192.168.0.106", 6555),
)
```

remote_server_ctxs List of contexts specifying one physical testing interface and one virtual testing interface. Elements of the list are either `ContextClient` objects, for physical interfaces, or objects created by `new_virtual_ctx` function, for virtual interfaces. An example of a configuration of a client follows:

```
remote_server_ctxs = ContextClientList([
  ContextClient(
    ContextLocal("enp0s31f6",
                 xdp_mode=XDPFlag.SKB_MODE),
    ContextCommunication("192.168.0.107", 6555)
  ),
  new_virtual_ctx(
    ContextLocal("a_to_b",
                 xdp_mode=XDPFlag.DRV_MODE),
    ContextCommunication("192.168.1.1"),
    "test_b",
    ContextLocal("b_to_a",
                 xdp_mode=XDPFlag.DRV_MODE),
    ContextCommunication("192.168.1.2", 6000),
  ),
])
```

B.3 Creating new tests

To create a new test, create a class inheriting from `XDPCase`. This class should be located in a file named with a `test_` prefix and placed in the `tests` folder. Each method of this class, that should be run while testing, has to be named with a `test_` prefix.

Each test should either call both `load_bpf` and `attach_xdp` methods in this order, before calling `send_packets`, or be decorated with `usingCustomLoader` and attach own XDP program to the interface.

After attaching attaching an XDP program, calling `send_packets`, returns a `SendResult` object, containing lists of packets that arrived to each interface engaged in testing.