Performance Implications of Packet Filtering with Linux eBPF

Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak and Georg Carle

Chair of Network Architectures and Services, Department of Informatics,

Technical University of Munich

{scholzd|raumer|emmericp|kurtz|lesiak|carle}@in.tum.de

Abstract—**Firewall capabilities of operating systems are traditionally provided by inflexible filter routines or hooks in the kernel. These require privileged access to be configured and are not easily extensible for custom low-level actions. Since Linux 3.0, the Berkeley Packet Filter (BPF) allows user-written extensions in the kernel processing path. The successor, extended BPF (eBPF), improves flexibility and is realized via a virtual machine featuring both a just-in-time (JIT) compiler and an interpreter running in the kernel. It executes custom eBPF programs supplied by the user, effectively moving kernel functionality into user space.**

We present two case studies on the usage of Linux eBPF. First, we analyze the performance of the eXpress Data Path (XDP). XDP uses eBPF to process ingress traffic before the allocation of kernel data structures which comes along with performance benefits. In the second case study, eBPF is used to install application-specific packet filtering configurations acting on the socket level. Our case studies focus on performance aspects and discuss benefits and drawbacks.

Index Terms—**Linux, eBPF, XDP, Performance Measurements**

I. INTRODUCTION

Controlling and monitoring network packets traveling from and to a program on a specific host and network is an important element in computer networks. In theory, the application itself could determine whether to interpret and process a packet received from the network and what information to send back, but current implementations lack the ability for four reasons. Our perspective is focused on Linux:

R1: Traffic should be filtered as early as possible to reduce unnecessary overhead.

R2: Application developers know best about the packet filtering configuration concerning their application and therefore should be able to ship policies together with their application in an easy way.

R3: Considering the system administrator would know the requirements of each application, she still would have to manage huge amounts of all slightly different and ever changing configuration files, complicating central policy implementation and verification.

R4: Orthogonal to these three policy related issues, modern applications have to cope with performance and latency requirements.

Over the past years, the state of the art was to have all of the hosts' packet filtering rules installed at a central location in the kernel. Configuring and maintaining a centralized ruleset using, e.g., iptables or nftables, requires root access, a scheme

not only used on UNIX-based systems. By the time a packet is filtered in kernel space, rejected traffic already caused overhead, as each single packet was copied to memory and underwent basic processing. To circumvent these problems, two trends for packet filtering can be observed: Either the filtering is moved to a lower level leveraging hardware support (e.g., offloading features or FPGA-based NICs), or breaking up the ruleset and moving parts to user space.

In this paper we show how eBPF can be used to break up the conventional packet filtering model in Linux, even in environments of 10 Gbit/s or more. Controlling the information flow only at the network endpoints is appealing in its simplicity and performance. Therefore, we use XDP (eXpress data path) for coarse packet filtering before handing packets to the kernel. This provides a first line of defense against traffic that is in general unwanted by the host, e.g., spoofed addresses [1] or Denial-of-Service flooding attacks. We contribute by comparing the achievable performance and latency for basic filtering operations to using classic approaches like iptables or nftables.

Additionally, policy rules become simpler if only one application on a host at a time has to be considered and the risk of leaving an application exposed to the network unintentionally is reduced if both packet filtering and application form a single unit. Default packet filtering rules could be shipped by application developers alongside their product, bringing us closer to secure-by-default systems while allowing techniques like port-knocking without root access. For that vision, we propose our solution for socket attached packet filtering. Because traffic unwanted by the host in general is already taken care of on a lower level, at socket level, per application decisions regarding the traffic of interest can be made. We show that both eBPF applications can be configured and used from user space even in high-performance environments of 10 Gbit/s networks.

This paper is structured as follows. In Section II, we present an overview of packet filtering on Linux machines at different points in the processing chain. A short history of eBPF in Linux is presented in Section III. In Section IV, we present measurements of common kernel packet filters as baseline for our case studies. Section V presents our XDP case study. Section VI describes our proposal for socket attached eBPF packet filtering and details the performance implications thereof. Related work is discussed as part of each case study, before we conclude with Section VII.
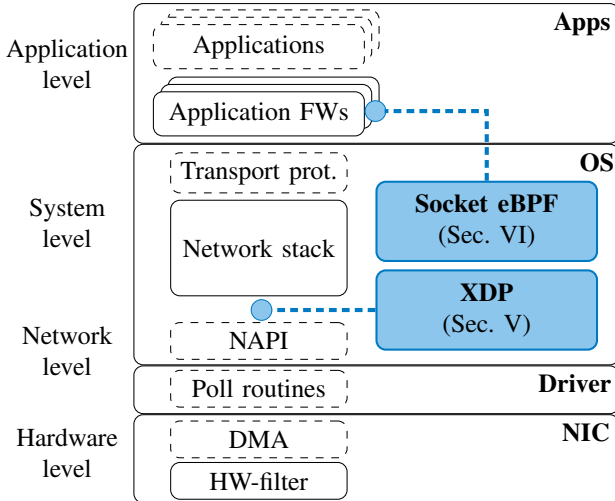
Figure 1: Different levels of packet filtering in Linux

II. PACKET FILTERING ON DIFFERENT LEVELS

Packets can be filtered at different stages on their way from the physical network interface until they reach the application. Figure 1 presents a rough overview of this; every non-dashed box presents a point where filters can be applied.

A. Hardware Level

The first, and from a performance perspective most attractive point for filtering as part of a firewall, is on the network interface card (NIC) itself. Modern server NICs offer hardware offloading or filter capabilities, which can be configured via driver parameters and tools like ethtool. Furthermore, dedicated platforms based on FPGAs or SmartNICs for packet filter offloading like HyPaFilter [2] have emerged. However, the functionality is limited and depends on the vendor and the specific NIC, resulting in a lack of general availability and no common interface for configuration.

B. Network Level

We refer to network level firewalls as any kind of packet filtering before the routing subsystem has started to process a packet. From performance perspective this is more attractive than a system level firewall as fewer CPU cycles are consumed for dropped packets.

In Section V, we analyze the benefits of early filtering via XDP, e.g., against DoS attacks.

C. System Level

Packet filters on system level like iptables or the newer nftables are widely used in Linux systems, achieving performance levels acceptable for today's applications. They hook into the processing at different locations, e.g., at the routing subsystem of the Linux kernel and therefore before the application. However, these firewalls require root access and system-specific knowledge; different rules may interfere and it is not possible to ship meaningful application-specific packet filters with the application.

In Section IV, we use both iptables and nftables as baseline for the performance measurements of our case studies.

D. Application Level

Application level firewalls look into traffic addressed for a specific application. In our second case study, we add application-specific eBPF-based packet filters to sockets using systemd's socket activation (cf. Section VI). This allows application developers to ship packet filtering rules that can be deployed together with their application. Those rules are specific to an application and simplify the central system-level firewall.

III. SHORT HISTORY OF BPF VMs IN LINUX

The increase of dynamic features is a general trend in the Linux kernel: Virtual machines, byte code interpreters, and JIT compilers help to abstract features and move complexity into the user space. Both case studies in this paper (XDP and our application level packet filters) are based on eBPF, which is available in Linux 4.x. eBPF allows for high-performance packet filtering controlled from user space. When filtering sockets of specific applications, root access is not required.

A. Berkeley Packet Filter VM

BPF, developed in 1992 for UNIX [3], has the purpose of packet filtering, improving the performance of network monitoring applications such as tcpdump. BPF has long been part of the Linux kernel. In the original paper [3], a BPF program can be found, which loads a packet's Ethernet protocol type field and rejects everything except for type IP, demonstrating the internal workings.

Since the release of Linux 3.0, the BPF VM has been improved continuously [4]. A JIT compiler was added for the filter, which enabled the Linux kernel to translate the VM instructions to assembly code on the x86_64 architecture in real time. In Linux 3.15 and 3.16, further performance improvements and generalizations of BPF were implemented, leading to the extended Berkeley Packet Filters (eBPF).

B. Extended Berkeley Packet Filters

eBPF is an extended form of the BPF VM with a network specific architecture designed to be a general purpose filtering system. While the features provided by or implemented with eBPF continues to grow [5], it can already be used for other applications than socket filtering, such as packet filtering, traffic control/shaping, and tracing. The IO Visor Project [5] lists eBPF features available by kernel version.

eBPF is a byte code with 64-bit instructions running on a machine with ten 32-bit registers, either dynamically interpreted or compiled just-in-time [5]. The instructions of the eBPF VM are usually mapped 1:1 to real assembly instructions of the underlying hardware architecture [6][7]. When the kernel runs an eBPF program, it sets up the memory this program can access depending on the type of the program (e.g., socket filtering or event tracing) and allows calling a predefined and type-dependent set of functions provided
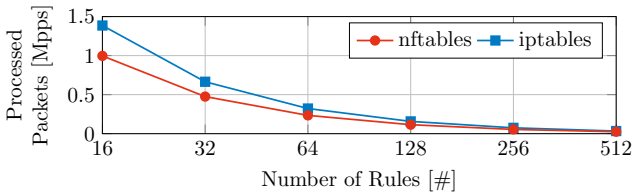
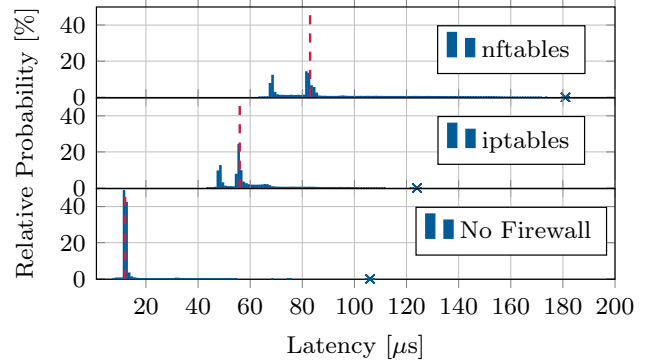Figure 2: Maximum throughput for iptables and nftables

Figure 3: Latency distribution for iptables and nftables at 0.03 Mpps. Maximum outlier (blue cross) and median (dotted red line) are marked for better visibility.

by the kernel [8]. eBPF programs are like regular programs running on hardware with two exceptions: First, when an eBPF program is loaded into the kernel, it is statically verified. This ensures that the program contains no backward jumps (loops) and does not exceed a maximum size of 4096 instructions. This verifier cannot be disabled without root access, impeding users from overloading the system with complex filtering rules. Thus, a malicious eBPF program cannot compromise or block the kernel, which makes it possible to allow non-root users to use eBPF [9]. All memory accesses are bound-checked.

The second difference are the key-value stores, coined *maps*. While regular programs can retain state and communicate with other programs using a variety of methods, eBPF programs are restricted to reading from and writing to *maps*. Maps are areas in memory which have to be set up by a user space helper before the eBPF program is loaded into the kernel. Both, the size of the key and value type, as well as the maximum number of entries are determined at creation time. Data can then be accessed in a secure manner by both user space and eBPF kernel space using a file descriptor [5].

C. eXpress Data Path

Linux 4.8 integrated eXpress Data Path (XDP) [5] into the kernel. XDP provides the possibility for packet processing at the lowest level in the software stack. Functions implemented in network drivers expose a common API for fast packet processing across hardware from different vendors. Through driver hooks, user-defined eBPF programs can access and modify packets in the NIC driver's DMA buffers. The result is a stateless and fast programmable network data path in the Linux kernel. It allows early discarding of packets, e.g., to counteract DoS, and bypassing of the Linux network stack to directly place packets in the egress buffers.

XDP is already actively used. Cloudflare integrated XDP into their DoS mitigation pipeline [10]. Incentives are the low cost for dropping packets and the ability to express rules in a high level language. In addition to deploying their own DoS protection solution based on XDP [11], Facebook has published plans to use XDP as layer 4 load balancer [12]. In their proposed scheme, each end-host performs the load balancing. This again is possible, because XDP hooks are performed before any costly actions.

IV. LINUX PACKET FILTERING PERFORMANCE

The iptables packet filter utility is part of Linux since kernel version 2.4 in 2001. It was introduced together with the netfilter framework that provides functionality to hook in

at different occasions in the Linux network stack. iptables rules trigger different behavior implemented in different kernel modules. This approach has shown to introduce drawbacks, like the need to implement additional modules for IPv6, which are, although copied in large parts from IPv4, a separate module in the Linux kernel.

In 2014, with Linux 3.13 nftables [13] was introduced. It also builds on the netfilter framework, but has only basic primitives like compare operations, or functionality to load information from the packets implemented in the kernel. The actual packet filters are created in the user context via translation of rules into a list of basic primitives that are evaluated via pseudo-machines whenever a packet needs to be checked. This for instance allows to extend packet filtering functionality for new protocols without kernel updates, by updating the userspace program for rule translation.

We consider both, iptables and nftables packet filters based on the netfilter framework of the Linux kernel as the default way of packet filtering in Linux. Therefore, we use them as baseline comparison for our subsequently described case studies of eBPF-based packet filtering approaches.

We perform basic tests to measure the performance of iptables and nftables by installing rules that do not apply to the traffic, but are sequentially checked before the packet passes. Figure 2 shows the maximum throughput for increasing rule set sizes. Our results coincide with related work, showing that iptables yields better performance [14]. Both approaches are able to process up to 1.5 Mpps for few number of rules, but quickly decline when increasing the number of traversed rules. The test traffic consists of a single flow, processing was hence restricted to a single 3.2 GHz CPU core. Figure 3 shows the latency distribution without firewall and with 3000 installed nftables and iptables rules at 0.03 Mpps which equals 15 Mbit/s at minimum packet size. iptables induces five times increased latency compared to not using any packet filters, while the median for nftables is further increased by roughly 20 μs. Both applications show two peaks and a long tail which can stretch up to 180 μs compared to 110 μs for no firewall.
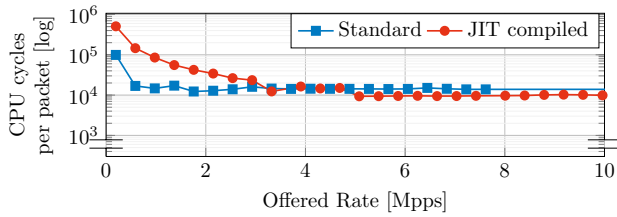
Figure 4: Lower bound of cycles per packet for an XDP application

V. PRE-KERNEL DoS PROTECTION

XDP yields performance gains as processing happens before allocation of meta-data structures like the kernel's `sk_buff` or buffering in software queues [15]. The programmability aspect is realized with eBPF. As result of the processing, a packet can either be dropped, passed to the Linux TCP/IP stack or transmitted back via the same NIC it was received on. Beside functionality that is usually deployed in firewalls on system level, this offers the possibility to use XDP for packet processing, coarse filtering, e.g., to protect against common DoS flooding attacks, stateful filtering, forwarding, load balancing tasks, etc. In other words, XDP is useful to protect the complete host from certain traffic, not to protect single applications.

A. Measurement Setup

For all measurements we connected two hosts directly with each other via a 10 Gigabit Ethernet (GbE) link. One host is used as load generator and sink using the traffic generator MoonGen [16]. The second host is the device under test (DuT) and runs the packet filtering program. The DuT is equipped with an Intel Xeon E3-1230 CPU (4 cores) @ 3.20 GHz and 16 GB DDR3 RAM and an Intel X540-T2 Network Adapter. The DuT runs a live boot Debian Stretch image with a Linux 4.12 kernel supporting XDP for Intel NICs. We used the Linux profiling utility perf for whitebox measurements of the DuT. All incoming traffic was pinned to one core to reduce the influence of jitter and side-effects on our measurements. Furthermore, we statically set the CPU frequency to 100 % and disabled Turbo Boost and Hyper-Threading. Each workload was tested for 30 s.

1) XDP Sample Program: The sample program we used for packet filtering with XDP is based on netoptimizer's prototype-kernel[1]. It consists of two parts: A command-line application to add and remove filtering rules and an application that attaches the XDP program to the network driver of the specified interface. The processing of an incoming packet is as follows. The XDP program first parses the Ethernet frame and performs sanity checks. If the parsing fails, the packet is passed on to the TCP/IP stack for further handling, otherwise it attempts to parse the IPv4 header and extract the source IP. If the extracted IP is contained in a blacklist, the packet is dropped. If not, execution continues to check the layer 4

header, extract the ports and look up another eBPF map that contains a list of blocked ports. If the port is blacklisted, the program returns `XDP_DROP`, otherwise, the packet is passed to the network stack.

We slightly modified the XDP program. Instead of passing valid packets to the TCP/IP stack, we swap source and destination MAC addresses and return the packet to the sender with `XDP_TX`. Including the network stack introduces complexity and potential side effects. Analysis of the network stack can be found in numerous related work [17], [18].

2) Filtering Using Kernel Bypass: In addition to comparing the results with the baseline measurements presented in Section IV we compare XDP with current state of the art kernel bypass technology. We use a packet filtering example based on the libmoon/DPDK framework [19]. The filter sample program was configured to align the number of threads and TX/RX queues to the same amount used by the XDP sample.

3) MoonGen Load Generator: We use the MoonGen [16] high-performance, software packet generator based on libmoon/DPDK for traffic generation. MoonGen is able to saturate a 10 Gbit/s link with minimum sized, user-customized packets (14.88 Mpps) using a single core and offers precise nanosecond-level hardware assisted latency measurements. All measurements were performed using an example script[2] sending minimum sized 64 B packets.

B. Performance Results

1) eBPF Compiler Mode: At the time of writing this paper (2018-03), the JIT compiler is still disabled by default as it is deemed experimental and not mature enough. To analyze the effect of the JIT compiler, we used the most basic XDP example, dropping every received packet. Our measurements show that enabling the JIT can improve the performance by up to 45 %, resulting in 10 Mpps total processed packets. As shown in Figure 4, the JIT increases the CPU cycles spent per packet for low packet rates. The optimizations through the JIT reverse this effect for higher rates. Processing packets with only 50 % of the cycles yields the observed increase in performance. With increased maturity and continuous optimization of the JIT this performance gain compared to static compilation is likely to increase further.

An XDP program uses the routines of the Linux network stack for packet reception and transmission. The costs for the operations of the NIC driver are known to be roughly 700 CPU cycles per packet [18]. Thus, we can estimate the lower bound for executing the XDP code to roughly 300 CPU cycles per packet when JIT compiled.

2) Maximum Processing Rate: Figure 5 shows the maximum performance when filtering packets with XDP with JIT enabled. XDP can process up to 7.2 Mpps in the case of 90 % packets dropped, a 28 % performance loss compared to the basic drop example. When reaching the point of full CPU utilization, the amount of processed packets remains constant, i.e., excess packets cannot be processed and are dropped

---

[1]https://github.com/netoptimizer/prototype-kernel/blob/master/kernel/samples/bpf/xdp_ddos01_blacklist_kern.c

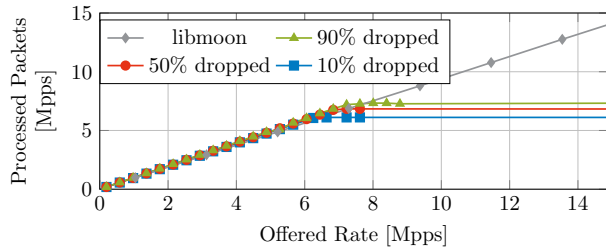[2]https://github.com/emmericp/MoonGen/blob/master/examples/l3-load-latency.lua

Figure 5: Packet filtering performance for different amount of filtered packets (JIT enabled)

(independent of eBPF rules and XDP actions). The 3 Mpps reduction in peak performance compared to the simple drop example is because the filter program performs additional tasks including the parsing and forwarding of packets. Depending on the percentage distribution of dropped and forwarded packets, the performance differs. Dropping packets costs less cycles than forwarding a packet. However, towards the worst case, i.e., all packets are forwarded, less than a 10 % reduction in performance is visible.

The results show a ~10-fold performance increase compared to iptables and nftables. As expected, the libmoon/DPDK-based kernel bypass approach was able to process at 10 Gbit/s line-rate for all scenarios.

In the following, our analysis is limited to the case of 90 % flows being passed. We chose this scenario to obtain more samples for our latency analysis, as only packets that pass the device can be timestamped.

3) Profiling: We use Linux' `perf record` profiling utility to analyze the costs of different internal processing steps. Linux' `perf record` allows us to count CPU cycles spent per function. As this results in hundreds of kernel functions, we group the functions by category. `bpf_prog` contains the eBPF program code itself (packet processing), while `bpf_helper` are functions that can be called from eBPF programs such as map lookups. Driver related functions of the NIC are grouped in `ixgbe`, while `kernel` denotes all other kernel functions. Finally, `idle` contains idle functions and unused CPU cycles. Note that in order to get meaningful output when profiling a JIT-compiled eBPF program, the `net/core/bpf_jit_kallsyms` kernel parameter has to be set, exporting the program as a kernel symbol to `/proc/kallsyms`.

Disregarding idle times for lower rates in Figure 6, the eBPF program's relative cost account for the highest percentage with approximately 60 % of CPU usage. The second highest is the ixgbe driver code, requiring close to 20 % of the resources, primarily for handling DMA descriptors (13 %). The BPF helper functions consume approximately 10 %. This is almost exclusively utilized for the execution of lookup functions. The processing performed by the kernel amounts to less than 5 % accounted to various utility functions.

4) Latency: Figure 7 shows latency percentiles up to the 99.99th percentile for the XDP packet filtering example with 90 % packets passed. We compare the cases with and without
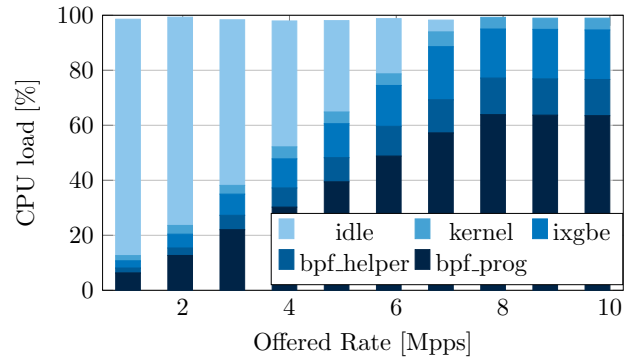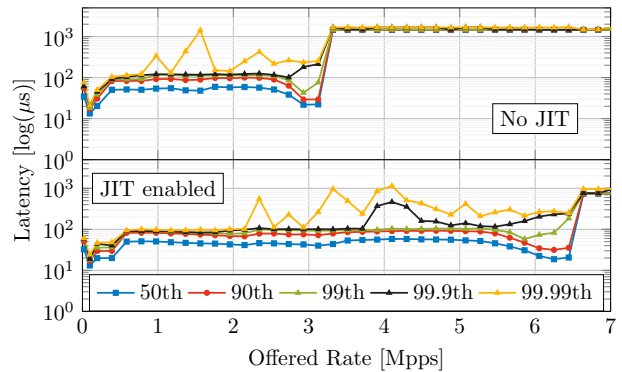

Figure 6: Profiling XDP

Figure 7: Latency percentiles for XDP packet filtering (90 % passing traffic)

JIT to analyze the effect of the JIT compiler on latency outliers.

As with the drop example, the JIT compiled code yields 3 Mpps more performance for the filter example. Two different areas of latency can be observed. Overloading the device leads to a high latency (800 μs to 1500 μs) caused by buffers. During normal operation, the program shows a median latency of roughly 50 μs. With increasing load the 99.99th percentile extends up to the worst case latency. At both edges, latency optima with median latency of 10 μs to 20 μs are observable. For very low data rates this extends throughout the 99.99th percentile.

Figure 8 shows the latency histograms for both optima and the average case during normal operation. These latency figures are in the expected and normal range for in-kernel packet processing with interrupt throttling (ixgbe ITR) [20], [21]. The histograms for the optima show similarities, independent of the JIT compiler. The difference is that the optimum for high data rates shows a long tail with outliers between 100 to 300 μs. During the average steady-state case (1.4 Mpps/3.9 Mpps) the median latency raises to roughly 50 μs. Both cases have a long tail, however, enabling JIT causes more and higher outliers up to 900 μs.

In comparison to the iptables and nftables baseline measurements, XDP achieves slightly better median latencies for the steady-state. However, the downside is the long tail, which can appear for all data rates beyond 0.5 Mpps and
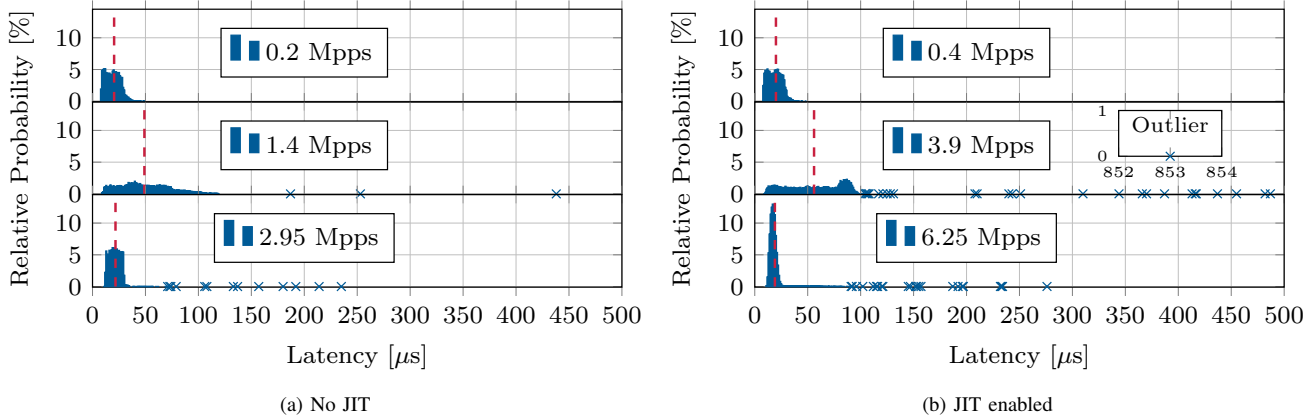
(a) No JIT

(b) JIT enabled

Figure 8: Histograms for XDP packet filtering with 90 % passing traffic. Median (red dotted line) and outliers (blue cross) are marked for better visibility.

can induce 20 times increased latencies. The JIT compiler increases this effect. This is a significant problem for modern high-performance applications with high demand to low and stable latency [22]. Such outliers do not appear when running an application on DPDK [21], the trade-off is that the CPU is always fully utilized by the poll-mode driver.

C. Discussion

Our measurements have shown that XDP provides a significant performance increase in comparison to iptables or nftables. Enabling the eBPF JIT compiler further increases the performance up to 10 Mpps. However, the line rate performance of a kernel bypass application like DPDK cannot be reached.

The mean latency of XDP is comparable to in-kernel filtering applications. Latency is dominated by interrupt throttling in the driver (ixgbe ITR) and dynamic interrupt disabling and polling in NAPI [20]. DPDK-based applications with a pure poll-mode driver provide a consistent low latency at the cost of 100% CPU load regardless of the offered load.

XDP offers a tradeoff. While performance is not as good as dedicated high-performance frameworks that bypass the kernel, it offers flexibility. It is fast enough to be deployed as DoS protection, but also offers kernel integration, i.e., packets can be passed through the network stack with all its benefits. While being able to process 7.2 Mpps might seem insufficient as it is only 50 % of 10 GbE line rate, this represents the performance of a single core, i.e., it scales with the number of CPU cores.

VI. SOCKET-ACTIVATED FILTERING

Our second case study aims at fine-grained filtering before packets reach the application. The purpose is not DoS protection. This should be handled in general for the complete host on a lower level, e.g. with XDP. Allowing to filter packets on a per-socket basis has clear advantages. Application developers can bind to the wildcard address, circumventing startup problems in case an address or interface is not yet available. The developers can ship their program with packet filters to
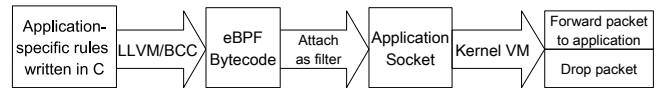

Figure 9: Steps from C based packet filter definition to effective socket attached packet filter

restrict the network exposure according to their requirements. Also, the user can define application specific firewall rules without privileges. This does not introduce security problems, i.e. one application can not mess with the domain of another, as rules can only be attached to a specific socket, only impacting the application listening on the socket. Lastly, network tool developers can implement complex firewall tools on top of this approach, such as custom port-knocking solutions or traffic analysis programs.

As a result, the complex firewall is decentralized, consisting of smaller, easy to maintain rulesets per application. This approach is less prone to errors and requires less administration.

A. Technical Overview

We combined the eBPF features in the Linux kernel and the systemd init daemon. systemd creates the socket, adds an eBPF machine and then passes it to the application via systemd socket activation. We implemented a proof-of-concept, available as open source [23], for demonstration and use it to analyze what performance penalty is inherited when attaching an eBPF filter to an application socket. Our implementation offers a command line interface, enabling to quickly write applications requiring even complex support from the firewall. We demonstrate this with an implementation of port-knocking using our tool [23].

Figure 9 shows the steps to instantiate application specific packet filtering rules via eBPF. In the first step, C code is compiled with clang into eBPF programs, before being attached as filter to an application socket. The kernel VM now runs the program to determine which packets are to be dropped and which to be forwarded to the application.
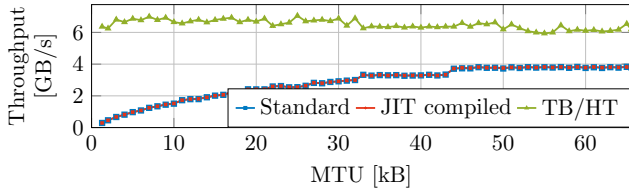
Figure 10: Baseline transmission speeds at different MTUs

1) BPF Compiler Collection: As programming with the low-level eBPF instruction set (cf. [24]) can be cumbersome, the BPF Compiler Collection (BCC) [25] provides the C compiler clang to compile restricted C code into eBPF programs. The restrictions with byte code verifier in the kernel are: no loops, limited number of instructions, no function calls, etc. BCC wraps LLVM/clang and enhances it with support for special syntax for defining eBPF maps inside the actual C program code. The eBPF maps are used to store state across multiple invocations of the eBPF program and for communication with the user space.

2) Socket Activation: Socket activation allows to spawn services on arrival of incoming connections. This allows the application to only start if and when a client actually connects to it, and to only stay active for the duration of the connection. While the first popular implementation, inetd, spawned a process for new requests, the systemd socket activation protocol only starts the main application once and passes the listening socket to it via a file descriptor with a specified number. We take advantage of this capability to pass sockets in our implementation and use this to implement application-level packet filtering. The server socket is created before the filter program is attached to the socket using the BCC library. The socket is then passed to the application via systemd socket activation. As both systemd and socket activation have become popular, many applications support preconfigured socket file descriptors. For those, our application firewall can be added without changes.

3) Data Availability: When transferring a stream of data using TCP sockets, the user space only sees one packet containing the TCP/UDP payload. The lack of Ethernet and IP headers makes packet filtering impossible.

Fortunately, eBPF allows negative memory addresses to implement certain Linux-only extensions such as getting the current time or determining the CPU core the filter program is currently running on [26]. Our implementation uses this to read from memory addresses containing the layer 2 and layer 3 headers, even if the filter is attached to a UDP or TCP socket.

B. Measurement Setup

The measurement setup differs in comparison to Section V-A. As benchmarking an application-level packet filter requires all packets to pass the kernel, it is expected that the performance is clearly below 10 GbE line rate on the same hardware. Initial measurements using a simple echo daemon without any packet filters confirmed that the peak throughput is below 4 Gbit/s. To maximize the impact of our socket-activated packet filters, we decided to measure the performance using the loopback interface of the DuT. Using a virtual link accomplishes this, as the results will be largely dominated by how efficient the eBPF filters (and the TCP/IP stack) work on Linux, assuming that processor and memory speeds are constant. Furthermore, we decided to use the Linux utility iperf instead of MoonGen as traffic generator and sink. This approach requires no dedicated networking hardware, allowing for simpler to reproduce experiments at the cost of not being able to analyze the latency behavior. However, we expect that latency will be primarily influenced by the same mechanisms discussed previously.

All measurements are performed using IPv6/TCP traffic generated by iperf. Instead of the packet rate, we measure the throughput of the application, as it is the important metric for application developers. We tested three different configurations. As in Section V-B1, we evaluated the impact of enabling the JIT compiler for eBPF. Furthermore, we analyzed the performance when also enabling dynamic frequency scaling of the CPU, Turbo Boost, and Hyper Threading (referred to as TB/HT). Note that these are usually disabled for performance measurements as they introduce jitter and for instance raise the CPU frequency above 100 %. However, we argue that it represents a realistic scenario for an application using a socket attached filter as they are enabled per default on Linux-based systems.

C. Performance Measurements

The following discusses the performance results of our socket-activated packet filtering tool.

1) Baseline: To determine maximum achievable transmission speed of our setup, we ran a performance test with iperf serving as both the client and server at the default MTU of the loopback interface (65536 bytes). The measurement showed a maximum transmission speed of roughly 3.8 GB/s as the limit.

Figure 10 shows the peak throughput for increasing MTU when using our client and server programs without attaching a socket filter. Instead, they use classic systemd socket activation to create and pass the socket.

The results show that our program without any filtering rules is able to reach the maximum throughput starting with approximately 42 kB. As we are not using socket filters, this is independent of the usage of the JIT compiler. For smaller MTU the correlation between MTU size and transmission speed appears sub-linear. This suggests a slightly increasing overhead with growing packet size. We attribute the visible steps of performance to side effects of the Linux memory management.

2) Subnet Filtering: The first scenario configures a socket filter to compare all ingress traffic against a configurable set of IPv6 subnets and only allow the packet to pass if a match was found, i.e., a whitelist filter. The matching rule is put increasingly further in the back of the list of subnets given to the socket filter, starting from index 0 up to index 22, which is the biggest index our filter supports, because of the maximum size of an eBPF program. We ran the test for
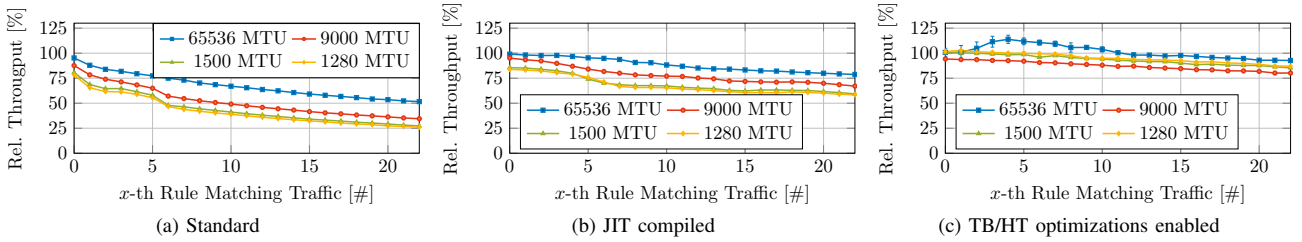
Figure 11: Subnet filtering performance at various MTUs and matching rule indices
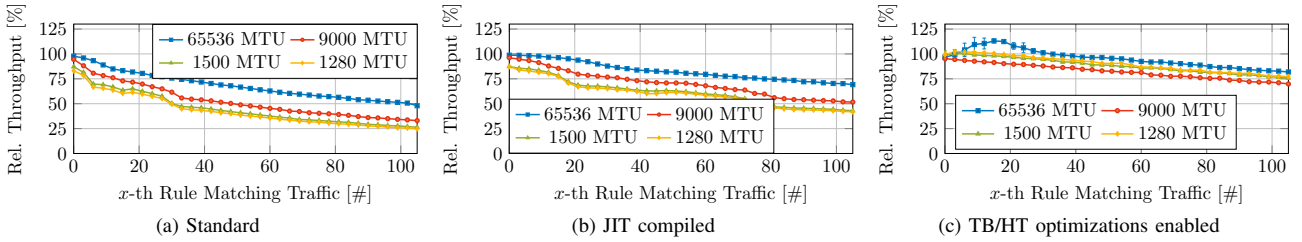

Figure 12: Interface filtering performance at various MTUs and matching rule indices

four different MTUs typically encountered. 1280 bytes is the minimum MTU allowed with IPv6 [27], 1500 bytes the default MTU for Ethernet [28], 9000 bytes a commonly supported size for Jumbo frames [29], and 65536 bytes is the default MTU of the loopback interface on Linux [30]. The results relative to the baseline speed at the corresponding MTU are displayed in Figure 11.

For all configurations, the performance degrades the further back the matching rule is in the set of whitelisted subnets. This is expected as rules are matched sequentially, i.e. non-matching rules before a match do cost performance. Higher MTUs generally lead to better performance, even though the performance is relative to the baseline performance of that MTU. This is because a higher MTU not only means fewer packets for the TCP/IP stack to handle, but also implies fewer invocations of the socket filter.

Enabling JIT (cf. Figure 11b) yields significant performance gains up to 100 % for the worst case of lowest possible MTU and highest possible index for the matching rule. Enabling further optimizations (cf. Figure 11c) further increases the performance, such that the minimum performance is 80 % of the baseline. Due to the baseline being slightly lower for the 1280 and 1500 bytes MTUs, the relative throughput for the 9000 bytes MTU is worse than the other measured MTUs.

3) Interface Filtering: The second scenario filters ingress packets depending on their incoming interface. The difference to the first scenario is that the (byte-) code for checking whether a packet arrived via a particular interface (32 bit integer) is shorter than the code required for doing a full IPv6 subnet matching (128 bit integer + logic for subnet matching). Consequently, the eBPF program can contain four times as many rules, allowing to further investigate the performance of the match and action processing.

The results of the worst case without JIT compilation in Figure 12a are almost identical to Figure 11a, despite the increase in the number of rules. Only when enabling JIT

(cf. Figure 12b) or all optimizations (cf. Figure 12c), the performance degrades to 50 % and 70 % of the baseline, respectively, when having to process more than 100 rules. In fact, in this scenario, when having to process the same number of rules as in the previous scenario (up to 22), the performance is equal or slightly better. This is due to the simpler packet filter, i.e., the interface ID in comparison to an IPv6 subnet.

D. Discussion

Making the packet filtering or general firewall decisions, at least partially, at the application level instead of the network or system level provides both better isolation between applications and also more freedom for application developers and users. While not all applications may require that freedom, there are examples (such as local file sharing solutions or remote management interfaces) whose overall security could be increased, if their network exposure could be limited in a flexible and configurable way. Additionally, an application level firewall does not have to care about the system-level configuration and its limitations, thus reducing the risks of creating an error-prone ruleset affecting other applications on the host.

The idea of filtering network traffic at the application level has in fact been implemented before with for instance TCP Wrappers [31]. However, these implementations typically require explicit application support (linking against a shared library [32]), and, more importantly, do not allow applications to ship their own, arbitrary, filtering rules. Instead, they are restricted to what the system administrator has set up in the global TCP Wrappers configuration file.

Our approach circumvents these issues. We have shown that this can be done at high data rates in both scenarios, once with few complex rules, and in the second case with one hundred simple rules. When using optimizations available and enabled per default in modern systems, the performance loss is below 30 % on a single core.

We argue that the bottleneck is the limit of the eBPF program size, i.e., the number and complexity of rules that can be attached to the socket. Considering machines with a realistic number of interfaces and therefore required socket filter rules, this is likely to be acceptable for packet filtering at application level. Besides packet filtering, our tool can also be used for more complex firewalling operations, which we demonstrated with a port-knocking application [23].

VII. CONCLUSION

As shown by our two case studies, eBPF can be used for versatile and high performance packet filtering applications.

XDP, hooking at the lowest level before the network stack, is well suited for coarse packet filtering such as DoS prevention. Our measurements have shown that XDP can yield four times the performance in comparison to performing a similar task in the kernel using common packet filtering tools. While latency outliers exist, which will likely be improved with increasing maturity of the XDP code base, the median latency also shows improvements. JIT compiled code yields up to 45 % improved performance at the cost of more and higher latency outliers. Furthermore, eBPF and XDP are constantly being improved or extended, e.g., eBPF hardware offloading or redirecting packets to another NIC for XDP, which will likely improve the performance and support more use cases.

Our approach for socket-activated packet filtering shows that eBPF provides flexibility. Application specific firewall rules can be set by each application individually without requiring root access. Our tool grants the freedom to the application developer to restrict the network exposure of the application to its needs. This can improve the classic, error prone, centralized configuration schemes used with iptables and nftables. The complexity of the global, system level firewall is reduced while security is improved through better isolation between applications. Performance losses are below 20 % for the worst case of maximum number of rules with minimum MTU when enabling JIT compilation and modern performance features like Turbo Boost.

The code of our eBPF demo application is available as free and open source [23].

REFERENCES

[1] P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing," Internet Requests for Comments, RFC Editor, RFC 2827, May 2000.

[2] A. Fiessler, S. Hager, B. Scheuermann, and A. W. Moore, "Hypafilter—a versatile hybrid fpga packet filter," in *Architectures for Networking and Communications Systems (ANCS), 2016 ACM/IEEE Symposium on.* IEEE, 2016.

[3] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93. Berkeley, CA, USA: USENIX Association, 1993.

[4] Q. Monnet. (2017, 10) Dive into BPF: a list of reading material. [Online]. Available: https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

[5] IO Visor Project. (2017, 09) BPF Features by Linux Kernel Version. [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

[6] J. Corbet, "BPF: the universal in-kernel virtual machine," May 2014. [Online]. Available: https://lwn.net/Articles/599755

[7] J. Corbet, "A JIT for packet filters," Apr. 2011. [Online]. Available: https://lwn.net/Articles/437981/

[8] IO Visor Project. (2017, 09) bcc Reference Guide. [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md

[9] J. Corbet. (2015, 10) Unprivileged bpf(). [Online]. Available: https://lwn.net/Articles/660331/

[10] G. Bertin, "XDP in practice: integrating XDP into our DDoS mitigation pipeline," in *Technical Conference on Linux Networking, Netdev*, vol. 2, 2017.

[11] H. Zhou, D. Porter, R. Tierney, and N. Shirokov, "Droplet: DDoS countermeasures powered by BPF + XDP," in *Technical Conference on Linux Networking, Netdev*, vol. 1, 2017.

[12] H. Zhou, Nikita, and M. Lau. (2017) XDP Production Usage: DDoS Protection and L4LB. [Online]. Available: https://www.netdevconf.org/2.1/slides/apr6/zhou-netdev-xdp-2017.pdf

[13] netfilter.org. The "nftables" project. [Online]. Available: http://netfilter.org/projects/nftables/

[14] P. Sutter. (2017) Benchmarking nftables. [Online]. Available: https://developers.redhat.com/blog/2017/04/11/benchmarking-nftables/

[15] Linux Foundation. (2017, Jul.) eXpress Data Path. [Online]. Available: https://www.iovisor.org/technology/xdp

[16] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Internet Measurement Conference (IMC) 2015*, Tokyo, Japan, Oct. 2015.

[17] R. Bolla and R. Bruschi, "Linux software router: Data plane optimization and performance evaluation," *Journal of Networks*, vol. 2, no. 3, pp. 6–17, 2007.

[18] D. Raumer, F. Wohlfart, D. Scholz, P. Emmerich, and G. Carle, "Performance exploration of software-based packet processing systems," *Leistungs-, Zuverlässigkeits-und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen, 6. GI/ITG-Workshop, MMBnet*, 2015.

[19] P. Emmerich. (2017) libmoon git repository. [Online]. Available: https://github.com/libmoon/libmoon

[20] P. Emmerich, D. Raumer, A. Beifuß, L. Erlacher, F. Wohlfart, T. M. Runge, S. Gallenmüller, and G. Carle, "Optimizing Latency and CPU Load in Packet Processing Systems," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2015)*, Chicago, IL, USA, Jul. 2015.

[21] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis," Jul. 2017.

[22] G. Tene, "How not to measure latency," 2013.

[23] A. Kurtz. alfwrapper – Application-level firewalling using systemd socket action and eBPF filters. Github project page. [Online]. Available: https://github.com/AlexanderKurtz/alfwrapper

[24] python sweetness. (2015, 07) Fun with BPF, or, shutting down a TCP listening socket the hard way. [Online]. Available: http://pythonsweetness.tumblr.com/post/125005930662/fun-with-bpf-or-shutting-down-a-tcp-listening

[25] BCC Project. (2017) Main repository. [Online]. Available: https://github.com/iovisor/bcc

[26] A. Starovoitov. (2017, 03) [iovisor-dev] Reading src/dst addresses in reuseport programs. [Online]. Available: https://lists.iovisor.org/pipermail/iovisor-dev/2017-March/000696.html

[27] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," Internet Requests for Comments, RFC Editor, RFC 8200, Jul. 2017. [Online]. Available: http://www.rfc-editor.org/rfc/rfc8200.txt

[28] IEEE, "Standard for Ethernet," *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, March 2016.

[29] I. Cisco Systems. (2017) Jumbo/Giant Frame Support on Catalyst Switches Configuration Example. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/switches/catalyst-6000-series-switches/24048-148.html

[30] Linux 4.13 Source Code. (2017) drivers/net/loopback.c. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/loopback.c?h=v4.13

[31] W. Venema, "TCP Wrapper - Network monitoring, access control, and booby traps." 1992. [Online]. Available: http://http://static.usenix.org/publications/library/proceedings/sec92/full_papers/venema.pdf

[32] D. J. Barrett, R. E. Silverman, and R. G. Byrnes. (2017) SSH Frequently Asked Questions: I'm trying to use TCP wrappers (libwrap), but it doesn't work. [Online]. Available: http://www.snailbook.com/faq/libwrap.auto.html