Eberhard Karls Universität Tübingen
Faculty of Science
Department of Computer Science
Chair of Communication Networks

Master thesis Computer Science

Adaptation of the VPP Firewall for Real-Time Packet Processing in Industrial Environments

Markus Schramm

Mar. 15, 2022

Reviewers

Prof. Dr. habil. Michael Menth
Department of Computer Science
Chair of Communication Networks
Eberhard Karls Universität Tübingen

Prof. Dr. rer. nat. Tobias Heer
Faculty Computer Science and Engineering
University of Applied Sciences Esslingen

Supervisor

Lukas Wüsteney, M.Sc.

Markus Schramm:

Adaptation of the VPP Firewall for Real-Time Packet Processing in Industrial Environments
Master thesis Computer Science
Eberhard Karls Universität Tübingen
Thesis period: Aug. 15, 2021 - Mar. 15, 2022

Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Master thesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, den Mar. 15, 2022

(Markus Schramm)

Contents

Abstract

Standard Ethernet is on the march in industrial environments to interconnect devices like industrial robots. Thanks to Time-Sensitive Networking (TSN), Ethernet can guarantee the deterministic and real-time transmission of packets as it is possible with the previously used bus systems. With the emergence of Industry 4.0 and cloud computing, the interconnection between industrial devices and controllers is continuously growing, leading to an increased network size.

On the other hand, the number of cyberattacks on companies is increasing. As a consequence, companies should pay particular attention to network security. Firewalls are one essential building block of network security. They are placed between two subnetworks to block traffic that is not explicitly allowed between the subnetworks. Unfortunately, firewalls, especially software firewalls, introduce a high latency and high jitter to packet forwarding. Such non-deterministic behavior is unacceptable when using TSN.

In this thesis, we propose three ideas that aim to reduce the latency and jitter of software firewalls to make them suitable for TSN. We implement these ideas into FD.io VPP, an existing high-performance software firewall. After that, we analyze our implementation regarding latency and jitter to evaluate whether the ideas are suitable for TSN.

Acknowledgment

I would like to thank everyone who supported me in writing this master thesis.

First, I would like to thank Prof. Dr. Michael Menth and Prof. Dr. Tobias Heer for making this thesis possible and for their support.

Moreover, I would like to thank Lukas Wüsteney M.Sc. for his support throughout the time writing this thesis and his helpful and critical tips to improve this thesis.

Finally, I would like to thank Hirschmann Automation and Control GmbH for providing the devices required to perform the performance measurements.

1 Introduction

Traditionally, networks in industrial environments consist of multiple parallel networks. Each network transmits data for different applications. Some networks transmit time-critical data, for example, for control industrial robots. Other networks, on the other hand, transmit general-purpose data, such as monitoring information. Thanks to the combination of standard Ethernet and Time-Sensitive Networking (TSN), these networks converge into one single network nowadays. TSN is part of the IEEE 802.1 standards. It is a mechanism that reduces the latency and the variation in latency (*jitter*) of time-critical packets. In other words, TSN enables time-deterministic forwarding of packets in converged networks. Converged networks enable interconnecting different industrial devices, leading the way to Industry 4.0. Furthermore, additional infrastructure from the cloud integrates into the network to make the plant more intelligent and flexible.

However, an increasing network size and additional (cloud) services connected to the network increase the number of vulnerable points. To make an intruder's life harder, the network is segmented into multiple zones between which firewalls are placed. The firewalls block all packets that are not allowed by the policy. The policy itself is defined by an administrator and only allows packets that need to pass to the other zone.

Imagine two devices, A and B, within the same zone exchanging control information. An intruder in another zone can manipulate device A by sending fake information. Even if device A filters incoming packets by the source IP address, the intruder can spoof the source IP address. Then, the packet looks like it originated from device B. A firewall prevents such an attack by blocking all packets that are not explicitely allowed. As packets from device B should not be allowed to arrive from other zones, the firewall protects device A from the above attack.

However, combining firewalls and TSN is currently a challenge because there are, to the best of our knowledge, currently no TSN-capable firewalls available. Wüsteney *et al.* [1] evaluated the filtering performance of firewalls regarding latency. They compared firewalls filtering in hardware with firewalls filtering in software. According to them, firewalls with hardware filtering satisfy the latency constraints of TSN, whereas firewalls filtering in software do not satisfy the latency constraints. However, using firewalls with hardware filtering in combination with TSN is not always a solution. They have disadvantages like less flexible rules (no support for stateful traffic) and limit the total number of filtering rules.

Because of these limitations, our goal is to modify FD.io VPP, an existing high-performance software firewall with software filtering, to satisfy the latency constraints of TSN. We propose three ideas, timebound, passive, and priority that we implement into the firewall. Furthermore, we measure and evaluate the performance of the modified firewall regarding latency and discuss the security impact.

2 Motivation

In this chapter, we first explain the challenges in industrial environments. Based on these challenges, we define our goals for the modified firewall regarding latency and jitter. Then, we explain the reasons for high latency and jitter of software firewalls. All this knowledge helps us to introduce our ideas to reduce the latency and jitter on software firewalls at the end of this chapter.

2.1 Challenges in industrial environments

The transmission of time-critical packets (in the following called *high-priority* packets) follows a strict schedule in industrial environments. TSN reserves recurring time slots on a forwarding device, like a switch, to transmit high-priority packets. During this time, the switch interrupts the transmission of other packets (*non-priority* packets). Thus, high-priority packets do not have to wait for other packets when they arrive at the switch. A waiting time is not tolerable for high-priority packets because it increases the latency and the jitter.

There are different types of high-priority traffic in TSN networks. Based on requirements contributions to the IEC/IEEE 60802 [2] and Wüsteney *et al.* [1], we distinguish three different types:

Isochronous traffic Isochronous traffic is cyclic, or in other words, the packets repeat in fixed intervals. The cycle times (intervals) are short, usually between 1 µs and 1 ms. Additionally, a switch must delay isochronous traffic by not more than 3 µs. Packet loss is not allowed. An example use case are synchronized motion applications.

Cyclic traffic The requirements of cyclic traffic are less strict in comparison to isochronous traffic. Common cycle times are between 2 ms and 20 ms. Moreover, cyclic traffic weaker requirements on delay and tolerates packet loss.

Acyclic traffic As the name suggests, acyclic traffic does not repeat in intervals. It occurs spontaneously, for example, caused by events or alarms. There are no requirements regarding the delay. To prevent the loss of event or alarm notifications, packet loss is not allowed.

The recurring time slots of a TSN switch are long enough that the switch can process and transmit the high-priority packet within this time. However, if the network consists of multiple switches, the time slots must compensate for the jitter caused by the other switches. We define jitter as the difference between the average latency and maximum latency and the average latency and minimum latency, respectively. For example, if the first switch introduces a jitter of 10 µs, the time slot must be 10 µs longer than the average time required to process and transmit the packet. The reason for this is that the packet can arrive up to 10 µs later at the second switch due to the jitter. To keep the time slots short, the switches must introduce only a low jitter. Long time slots decrease the throughput of the switch because the switch cannot process other packets during this time.

2.2 Goals

As we mentioned in the last section, TSN switches should only introduce a low jitter to keep the time slots short. Since network segmentation using firewalls plays an increasing role, we try to achieve the same with our modified firewall. Thus, our goal in this thesis is to reduce the jitter of the firewall as much as possible so that the latency is as constant as possible. We aim to meet the requirements of cyclic traffic and acyclic traffic because isochronous traffic is expected to stay inside one network. However, the low maximum delay of 3 µs for isochronous traffic is impossible to achieve, as our measurement results in Chapter 7 show.

We do not improve the general packet processing of the firewall in this thesis. We focus on improving the firewall implementation. Therefore, we chose to modify a software firewall that already processes packets very fast.

2.3 Impact of software firewalls on latency

In this thesis, we focus on software firewalls. Software firewalls are a firewall subcategory. In contrast to many hardware firewalls, they do not rely on special hardware like Application-Specific Integrated Circuits (ASICs) or Field-Programmable Gate Arrays (FPGAs). Instead, software firewalls run on commodity hardware like server hardware used in data centers or embedded systems. Commodity hardware brings more flexibility because features are not bound to special hardware with limited capabilities. For example, if a firewall uses an ASIC to filter packets, the number of rules and their structure is limited by the capabilities of the ASIC. In contrast, a general-purpose CPU does not limit the number of rules and their structure. Furthermore, with software firewalls, there is no dependency on a hardware manufacturer and commodity hardware is often cheaper than specialized hardware. On the other hand, commodity hardware can degrade the performance because it is less efficient than hardware that is optimized for a special purpose. One task that suffers from performance degradation on commodity hardware are table lookups (e.g., a routing table or a table/list with firewall rules). They can be implemented with a constant-time delay in hardware while the delay varies per lookup on a general-purpose CPU. In this section, we show how table lookups on software firewalls influence packet latency.

We start with a brief introduction to firewalls in order to explain why such table lookups are required (we give a complete overview of firewalls in Section 3.1). Firewalls are configured by an administrator who adds a set of rules that determine

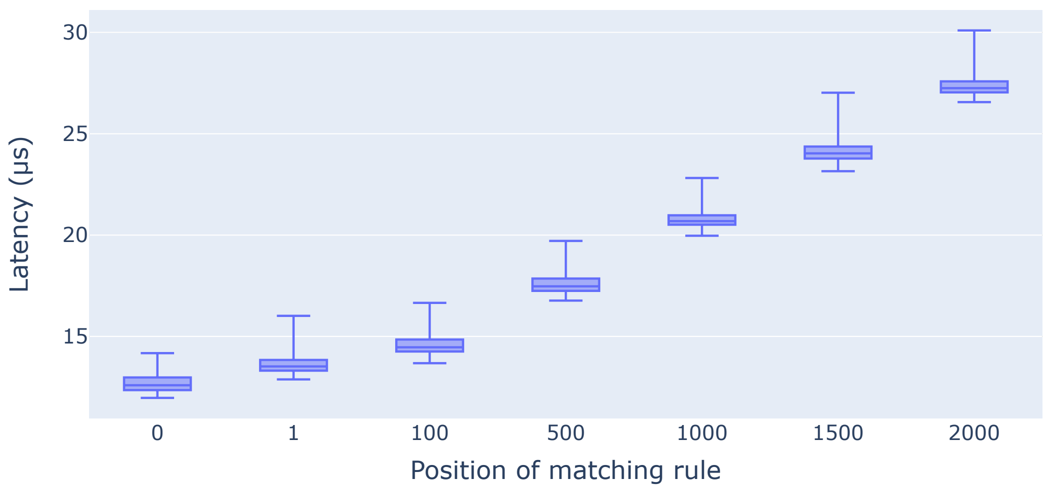


Figure 2.1: Latency for different matching positions (data rate: 1 Mbit/s; packet size: 64 B)

whether a packet is allowed to pass through the firewall or whether it is blocked. Such a set of rules to filter packets is called Access Control List (ACL). A rule consists of fields and an action. The firewall compares the fields against each packet arriving at the firewall. If all fields match the packet (i.e., the rule matches), the action defined in the rule is taken. Valid actions are *allow* (forward the packet to the destination) and *block* (drop the packet). If no rule matches, the firewall takes a default action. In most cases, the default action is to drop the packet.

Software firewalls process ACLs on the CPU. They match the rules in a linear way, meaning the rules are processed one after another. As a result, the duration to match the ACL rules is dependent on the number of rules that need to be matched. This can slow down packet processing drastically.

For example, Figure 2.1 shows the increasing latency of a packet depending on the position of the matching ACL rule[1]. We see that a packet where the 100th rule matches has a median latency of 14.5 µs while a packet where the 1000th rule matches has a higher median latency of 20.7 µs. Figure 2.1 also shows that the latency increases linearly based on the number of rules. In our example, we identify a linear increase of the latency by around 3.2 µs per 500 rules (this value varies on different firewalls and hardware).

With an increasing number of ACL rules, the firewall needs more time to process the ACL rules (the *ACL processing takes longer*). Thus, less time remains for remaining packet processing before the next packet arrives. To give an idea of how much time a firewall has to process a packet (e.g., ACL matching and forwarding the packet), we show the frame interarrival time for different packet sizes[2] and data

[1]The whiskers on the box plot denote the minumum and maximum latency observed during the measurement. The boxes are drawn from the first to the third quartile and the horizontal line in between denotes the median of the latency.

[2]Note that we use the term "packet" as a synonym for the OSI layer 2 "frame" throughout the

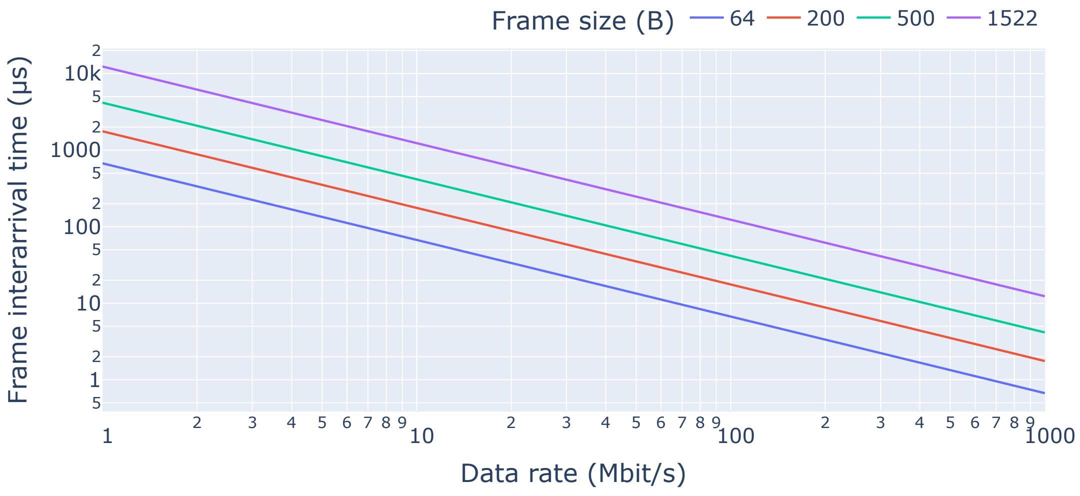


Figure 2.2: Amount of time between the arrival of frames for different packet sizes and data rates

rates in Figure 2.2. The interarrival time is the amount of time that passes until the next packet arrives, assuming a constant packet size and data rate. We can calculate the interarrival time A in seconds with Equation (2.1) using the packet size s_p in bytes and the data rate r_d in Gbit/s. In the equation, we add 20 to the packet size because of the OSI layer 1 overhead of 20 bytes (preamble, start frame delimiter, and interpacket gap). At the smallest possible packet size (64 bytes) and a data rate of 1 Gbit/s, only 0.67 µs remain to process a packet. In contrast, at the maximum possible (regular) packet size of 1522 bytes and a data rate of 1 Mbit/s, the firewall has 12 336 µs to process a packet.

$$A = \frac{(s_p + 20) \cdot 8}{r_d} \tag{2.1}$$

At an increasing rate of arriving packets, the firewall cannot keep up checking the ACL anymore if too many rules need to be checked. The time required to match the rules exceeds the time available to process the packet. In that case, one or more packets arrive at the firewall while the firewall still processes another packet. To avoid that the firewall must drop these packets, it stores arriving packets in a queue, more specifically, in the *ingress queue*. Each port of the firewall has such an ingress queue. When the firewall has processing resources available, it fetches the next packet from the ingress queue to process it. However, an increasing number of packets in the ingress queue leads to congestion, meaning the latency increases up to milliseconds. The queue can only store a limited number of packets. As a consequence, the firewall must drop packets if the queue is full, which leads to packet loss.

thesis.

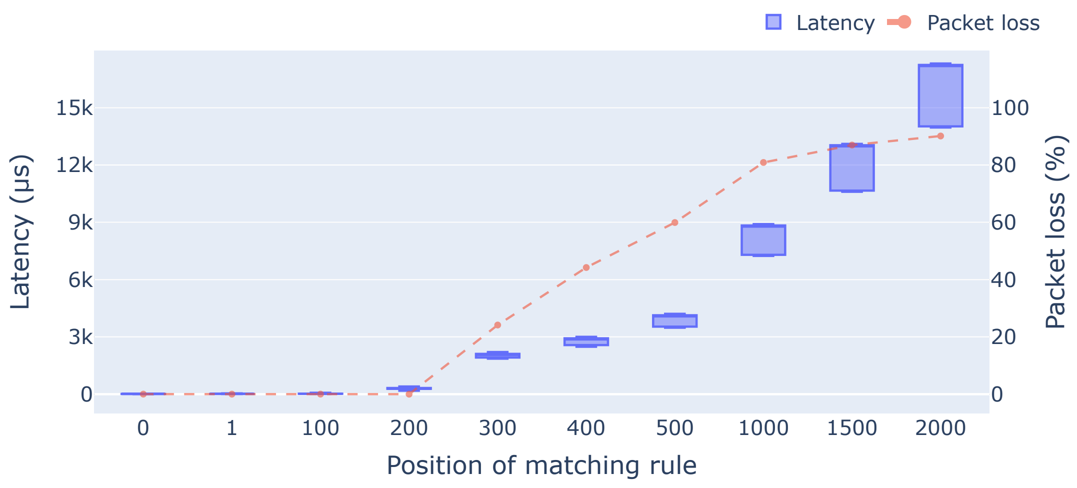


Figure 2.3: Latency and packet loss for different matching positions (data rate: 500 Mbit/s; packet size: 64 B)

The low data rate of 1 Mbit/s in Figure 2.1 offers enough time (672 µs) for the firewall to process 2000 rules. Thus, the ingress queue is empty. This changes in Figure 2.3, where the data rate is 500 Mbit/s. The ingress queue fills completely up, starting at slightly over 200 rules to process per packet. This is visible in the increasing packet loss that occurs due to the full queue. The high data rate of 500 Mbit/s does not leave enough time to match all rules before the next packet arrives.

2.4 Ideas to achieve low latency and low jitter

To meet the low latency and low jitter requirements, we propose three ideas that we design (see Chapter 5), implement (see Chapter 6) and evaluate (see Chapter 7) in this thesis. The first two ideas change the way how the firewall performs ACLs checks and the third one changes the way how the firewall processes high-priority packets.

2.4.1 Timebound

With the timebound idea, we aim to reduce the ACL processing time below the packet interarrival time. If the ACL processing time stays below the packet interarrival time, the ingress queue does not fill, preventing high latency, jitter, and packet loss. This is achieved by limiting the ACL runtime to a configurable duration (time limit, e.g., 2 µs). When the time limit exceeds, the firewall stops ACL processing of the packet and forwards the packet as if the ACL check yielded the allow action.

The time limit is configurable (e.g., by an administrator) on a per-flow basis so that packets with higher priority can be assigned a shorter time limit. For example, packets with a different source IP address or packets with a different priority code point field in their VLAN tag can be assigned different time limits. Note that we do not recommend configuring time limits with the same granularity as the ACL rules. The number of time limit rules would grow too fast, leading to the same performance issues as the ACL check. Instead, the time limits are intended to configure a single time limit rule for a set of different connections. For example, a time limit rule applying to all packets with the same priority stored in the VLAN tag.

One problem remains if the time limit is only applied only to a few flows: packets of other flows can still increase the ACL processing time above the packet interarrival time. Using timebound, we can solve this problem by limiting the ACL processing time of all packets. However, a static time limit for all packets does not scale with the rate of arriving packets. In dynamic load scenarios, the packet rate can rise to any rate. Thus, the interarrival time can further decrease while the time limit is static, leading to packet loss at higher packet rates. As a solution, we let the firewall automatically determine a suitable time limit for each packet. This allows the firewall to match as many ACL rules as possible without causing the ingress queue to fill up.

Stopping ACL checks before they are actually finished worsens the security because the firewall cannot match all rules. For example, the result of the ACL check could have been to block the packet but the rule responsible for this decision was never reached because the packet was already forwarded. Therefore, the decision which packets are configured for time-limited ACL checks or the decision for the value of the time limit should be made wisely.

To provide at least some level of security, the firewall continues the ACL check in the background after it forwarded the packet without delaying other packets. As a result, the packet is not further delayed, but we still get the result of the ACL check. This allows to report or log wrong decisions (i.e., cases where a packet was forwarded although it should have been dropped). Logging wrong decisions is a tradeoff between latency and jitter reduction and security. Possible logging targets are, for example, syslog or a monitoring service.

In summary, timebound consists of three components that can but do not have to be used in combination:

- A *configurable time limit* to configure a custom time limit per flow (to limit the processing time of high-priority packets).
- An *adaptive time limit* that applies to all packets.
- *Analyze later* to complete the remaining ACL checks in the background to get notified about packets that should have been blocked by the ACL.

2.4.2 Passive

With the passive idea, we completely avoid the delay caused by ACL checks by forwarding the packets without ACL check. Whe firewall performs the ACL checks in the background using analyze later from the timebound idea. Whether the passive idea should be applied to a packet is configurable on a per-flow basis equally to the timebound idea.

2.4.3 Priority

Our goal with the priority idea is to prioritize the processing of high-priority packets to forward them as fast as possible at the cost of non-priority packets. Processing of non-priority packets, on the other hand, is less time-critical so the latency can be higher.

To forward high-priority packets as fast as possible, the firewall must be able to classify arriving packets as high-priority or non-priority. Whether the firewall should classify a packet as high-priority is configurable on a per-flow basis. Moreover, the firewall must be able to spend as much processing power as possible on high-priority packets as soon as they arrive at the firewall, neglecting non-priority packets if needed. This ensures that the firewall processes high-priority packets as fast as possible.

In contrast to the other two ideas, timebound and passive, the priority idea is not limited to ACL processing. Instead, the firewall classifies packets at the beginning of processing, for example, when the firewall fetches the packets from the queue. This ensures that we can prefer high-priority packets throughout the whole forwarding and filtering process.

With the priority idea, we effectively reduce the packet rate which leads to a higher interarrival time. We can ignore the effect of non-priority packets on the packet rate and only take high-priority packets into account. Thus, the firewall has more time to process a high-priority packet until the next high-priority packet arrives (assuming the rate of high-priority packets is lower).

3 Background

Firewalls and packet processing play a fundamental role in this thesis. Therefore, we first give an overview of firewalls. After that, we explain how packet processing in the Linux kernel and the user space works. There are some differences between packet processing in the kernel and the user space that make a significant difference in performance. In between, we present several firewalls that we compare with each other in Section 5.1 because we considered using them to implement our ideas. Lastly, we explain the architecture of DPDK and FD.io VPP. DPDK is a framework on which the firewall we modify, FD.io VPP, relies.

3.1 Firewalls

A firewall plays an essential role in network security. It is usually placed at the border of a network where it is connected to another network that is considered untrusted. An example of an untrusted network is the Internet. Likewise, two networks with different security levels may be divided using a firewall. The firewall filters packets flowing between the two networks based on a policy (configured by the administrator) to prevent malicious packets from entering the trusted network and to prevent internal data from flowing into the untrusted network.

In the following, we show the different deployment locations and deployment types of firewalls (software or hardware). Then, we explain and differentiate the different firewall categories. The segmentation into different firewall types and categories is not always unambiguous because the transitions in between are smooth. However, some literature [3–6] helped us to find a common overview of firewalls that we present in this section.

3.1.1 Personal firewall vs. Network firewall

There are two main locations where a firewall can be installed. First, a firewall can be installed on a client, for example, an office computer. Common operating systems, like Windows, already come with a preinstalled firewall to prevent attacks on the client. Firewalls running on such clients are called *personal firewalls* or *host firewalls*. Figure 3.1 shows an example topology where all internal peers (clients) come with their own personal firewall.

Second, a firewall can be a dedicated device that is placed in the network. It protects all hosts and servers connected to the network from attackers in the untrusted network. This firewall type is called *network firewall*. Figure 3.2 shows an example topology where a network firewall protects all internal peers. Note that the peers can still be equipped with their own personal firewall, which we omitted in the example topology.

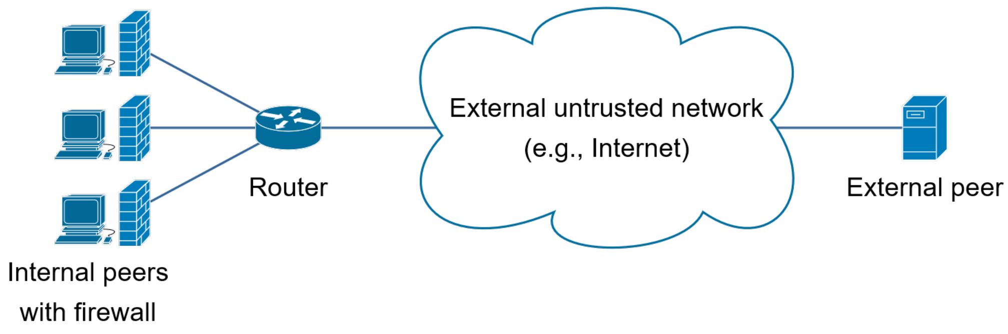


Figure 3.1: Example topology with personal firewalls

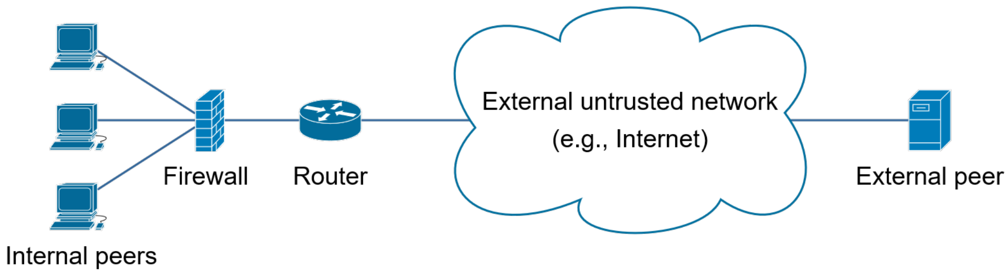


Figure 3.2: Example topology with a network firewall

	Layer	Data unit
7	Application	
6	Presentation	Data
5	Session	
4	Transport	Segment, datagram
3	Network	Packet
2	Data link	Frame
1	Physical	Bit

Table 3.1: OSI model (layers based on ISO/IEC 7498-1:1994 [7], data units based on RFC 1122 [8])

3.1.2 Software firewall vs. Hardware firewall

Network firewalls come in two different forms: a *software firewall* or a *hardware firewall*. A software firewall runs on top of a common Operating System (OS) like Linux, that is, it is installed after installing the generic OS. The OS runs on commodity hardware. This means that the software firewall performs packet processing on the general-purpose CPU.

A hardware firewall comes as a dedicated device on which the firewall system is installed. The operating system is often based on Linux or FreeBSD. Usually, a hardware firewall includes custom hardware, like ASIC, that speeds up the packet processing because the ASIC is highly optimized for its purpose. For example, the ASIC stores the routing table, allowing the hardware firewall to look up the next hop fast and in constant time. A software firewall uses the CPU to search the routing table that is stored in the memory instead. This usually takes longer compared to a lookup using the ASIC.

3.1.3 Firewall categories

The different firewall categories that we present in the following can be implemented by both, personal firewalls and network firewalls, as well as in both appearances, software firewalls, and hardware firewalls. Multiple categories can also be (and often are) combined into one single firewall product.

3.1.3.1 Stateless packet filter

A *stateless packet filter* is the most basic firewall category and usually operates on the data link layer, network layer, and transport layer of the OSI model (see Table 3.1 for an overview of the OSI model). It analyzes incoming packets (flowing from the untrusted network into the trusted network) as well as outgoing packets (flowing from the trusted network into the untrusted network) and subsequently allows or blocks them based on a configurable policy.

The policy is implemented with one or more ACLs that need to be assigned to each network interface if the packet filtering functionality should be enabled on that interface. The ACLs can be assigned per direction, *ingress* (packets arriving on the interface) and *egress* (packets leaving the interface). An ACL consists of one or

Src IP	Src port	Dst IP	Dst Port	Protocol	TCP flags	Action
192.0.2.0/24	-	-	443	TCP	-	Allow
192.0.2.0/24	-	-	8080-8085	UDP	-	Allow
192.0.2.0/24	-	198.51.100.20	25	TCP	-	Allow

(a) Stateless egress (outgoing) ACL

Src IP	Src port	Dst IP	Dst Port	Protocol	TCP flags	Action
-	443	192.0.2.0/24	443	TCP	ACK	Allow

(b) Stateless ingress (incoming) ACL

Table 3.2: An example of stateless ACLs. "`Src`" and "`Dst`" are abbreviations for source and destination.

more rules. Each rule consists of several fields whose values can be compared with the values of the corresponding fields in a packet. Table 3.2 shows a typical example of the structure of an ACL. Each table column stands for one field in the ACL. In this case, the fields consist of a *5-tuple* (source IP address, source port, destination IP address, destination port, protocol), and additionally, a `TCP flags` field. There is also an `Action` field that defines how the packet is handled by the packet filter if a rule matches the packet. Possible actions are `Allow` to let the packet pass through the packet filter and `Block` to drop the packet. Additionally, other typical fields are, for example, source and destination MAC address as well as fields to filter Internet Control Message Protocol (ICMP) packets based on the ICMP type and ICMP code.

When a packet arrives on an interface to which an ACL was assigned, the firewall compares the packet with each ACL rule until one rule matches. The order in which the rules are compared is equal to the order defined by the administrator during configuration. The comparison (matching) of a packet with a rule works as follows: each field of the rule is compared with the corresponding field in the packet. If the values of all corresponding fields are equal or included in the range, the packet matches the rule. As a consequence, the action defined by the rule is taken. A rule field can also be empty (indicated by "`-`" in the table). Then, the field is ignored and not compared with the packet. If no rule matches, a default action is taken which is usually to block the packet.

Example: imagine a peer in the trusted network (internal peer) talks to an external server, that is, the UDP packet in Table 3.3 arrives at the egress ACL in Table 3.2a. First, the packet is matched against the rule in line one. The source IP address matches because the address `192.0.2.5` is part of the network `192.0.2.0/24` but the protocol (and the port) do not match. The fields indicated with "`-`" are ignored because they are undefined. Because there was no match, the packet is matched against the rule in the second line. Again, the source IP address matches as well as the protocol and the destination port because `8081` is in the range `8080-8085`. As a result, the action `Allow` is taken, which means the packet is forwarded.

One problem arises if a peer in the trusted network tries to establish a connection to an external peer (e.g., a server) that involves bidirectional communication and therefore requires packets flowing in both directions. One may configure the stateless packet filter to allow packets flowing to external peers but disallow packets flowing from external peers to internal peers (i.e., configure an egress ACL rule with allow

Source IP	192.0.2.5
Destination IP	198.51.100.20
Protocol	UDP
Source port	58952
Destination port	8081

Table 3.3: Example IP packet that arrives at the egress ACL in Table 3.2a

action and block ingress packets). As a consequence, packets can only flow from the trusted to the untrusted network, and therefore, a bidirectional packet flow between an internal peer and an external peer is not possible. There is a solution for this problem as long as TCP is used as protocol between the internal peer and external peer: an ingress ACL rule can be created to allow TCP segments having the ACK flag set. An example rule is shown in Table 3.2b. This works because every TCP segment has the ACK flag set except for the initial packet opening the connection. As a result, external peers can send packets that are part of an established TCP connection but cannot establish a connection to a peer inside the trusted network themselves. Only internal peers are allowed to send the initial TCP segment for connection establishment that has the SYN flag set. A disadvantage is that the packet filter also allows packets that have the ACK flag set but are not part of an open connection because the packet filter does not know whether the packet belongs to an open connection. In general, applications running on peers should ignore such packets not belonging to an open connection but there can be vulnerabilities that can be exploited by attackers. As already mentioned, the presented solution only works with TCP as a transport protocol. Because of the statelessness of UDP, there is unfortunately no such solution for UDP packets on stateless packet filters.

3.1.3.2 Stateful packet filter

The *stateful packet filter* is similar to the stateless packet filter but extends the latter by connection tracking. This means, it recognizes when a peer opens a TCP connection and stores the information about the connection in a *connection table*. The rough structure of a connection table is shown in Table 3.4.

When a packet arrives at the stateful packet filter, it checks whether the packet belongs to an active connection by comparing the packet's fields with the entries in the connection table. If a matching entry was found, the packet is allowed and the table is updated with the latest information about the connection, that is, the `Last active` column of the corresponding entry is updated with the current timestamp in our example. If the packet does not belong to an open connection, the packet is processed by the ACL and allowed or blocked based on the result of the ACL check. If the packet is the initial packet of a connection (i.e., opens the connection) and is allowed by the ACL, the new connection is added to the connection table. Inactive connections where no packets were transmitted for a period of time are deleted from the connection table. Whether a connection is old and can be deleted can be recognized by comparing the timestamp in the `Last active` column with the current time. The termination of a TCP connection using the FIN flag can also lead to the deletion of the corresponding table entry.

Whether connection tracking should take place for certain connections can usually

Src IP	Src port	Dst IP	Dst Port	Protocol	Last active
192.0.2.2	59853	233.252.0.128	443	TCP	524 587 468
192.0.2.5	58952	198.51.100.20	8081	UDP	550 000 001
192.0.2.26	61589	198.51.100.20	25	TCP	529 815 954

Table 3.4: Example of a connection table

be defined with an additional ACL action that can, for example, be called "reflect". This action is used instead of the allow action. UDP packets can also be considered for connection tracking, though, UDP is stateless and a termination of the connection cannot be detected. A timeout using the `Last active` column is the only way to "terminate" a UDP connection in the view of the stateful packet filter.

Effectively all firewalls today combine a stateless and a stateful packet filter, even if a stateful packet filter has the disadvantage of a higher performance and memory impact due to the connection tracking and the connection table.

3.1.3.3 Circuit-level gateway

The *circuit-level gateway* operates on the session layer. It monitors handshaking between two peers and decides whether the connection is legitimate. In addition, it modifies the packets in a way that they appear to the external peer as they originated from the circuit-level gateway. This is done by, for example, replacing the source IP address on packets originated from the internal peer with the circuit-level gateway's IP address. On packets sent from the external peer to the internal peer, the destination address of the internal peer is restored. To keep track of the established connections and to modify the packets correctly, a connection table, similar to the one shown in Table 3.4, must be used. But additional columns are required to remember the values that need to be replaced in the packet. Hiding the internal peer from the external peer has the advantage that the external peer does not receive any information about the internal peer as well as the internal network. Since a circuit-level gateway only monitors the handshaking, it should be used in combination with a packet filter to filter individual packets too. For example, SOCKS [9] is a well-known protocol that can be implemented by circuit-level gateways.

3.1.3.4 Application-level gateway

Similar to a circuit-level gateway, the *application-level gateway*, also called *application proxy*, sits between two peers to monitor the exchanged packets and hide the internal peer from the external peer. But unlike the circuit-level gateway, it operates on the application layer to check whether an application is allowed by the policy. For example, applications like File Transfer Protocol (FTP) or Hypertext Transfer Protocol (HTTP) can be completely blocked or filtered based on the application layer payload and regardless of the used port. Filtering based on the requested Uniform Resource Locator (URL) in the case of HTTP is one example.

While the application-level gateway offers great flexibility, it requires much more processing power than the circuit-level gateway. Another disadvantage is that not all applications may be supported because there are so many of them.

3.2 Kernel space networking

In a Linux-based distribution, the Linux kernel takes over packet processing. Only the packets destined for the host are passed to the corresponding user space application by the kernel. In the following, we explain how the Linux kernel processes packets and why the performance is not optimal.

3.2.1 Packet arrival

When a packet arrives, the Network Interface Controller (NIC) copies it to a predefined region in the memory. This has the advantage that the kernel does not have to copy the packet from the NIC itself and therefore saves processing time. The capability of other hardware to access the memory independently of the CPU is called Direct Memory Access (DMA).

However, the kernel must recognize that a packet arrived and was copied to the memory in order to start processing it. The usual way to do this is using interrupts. This means, the NIC sends an interrupt to the CPU whenever a packet arrives. Consequently, the CPU interrupts processing to notify the kernel about the arrival of a new packet. With an increasing rate of arriving packets, the interruption of the kernel on the arrival of each packet introduces a significant processing overhead. As a result, more processing is spent with interrupt handling than with actual packet processing. Because of this, the New API (NAPI) was introduced in the kernel version 2.5 [10].

The NAPI switches to polling mode (instead of interrupt mode) if the rate of arriving packets is high. Depending on the packet rate, only a few or no interrupts are triggered by the NIC. Polling means that the kernel checks the ingress queue of the NIC by itself for the arrival of new packets. Polling increases the CPU utilization because the CPU permanently polls the ingress queue if the CPU has no other processing tasks. Without polling, the CPU goes to sleep as long as no packets are left to process. The NAPI tries to find a balance between interrupts and polling by combining them. At a low packet rate, the overhead of interrupt handling is no problem because enough resources are available. In contrast, polling would lead to an unnecessary high CPU utilization in that case. At a high packet rate, polling reduces the overhead caused by interrupt handling while it does not unnecessarily increase the CPU utilization.

3.2.2 Packet processing

The packet processing in the kernel is divided into three layers [10]. These three layers correspond to the OSI model: layer 2, layer 3, and layer 4. Figure 3.3 shows how an IP packet traverses the layers.

Layer 2, the data link layer, is implemented by the network driver. It interacts with the NIC to receive and send packets. When a packet arrives, the network driver allocates an instance of the `sk_buff` structure. The `sk_buff` structure represents a packet in the kernel. It stores a reference to the packet data and additional information about the packet. For example, a reference to the device on which the

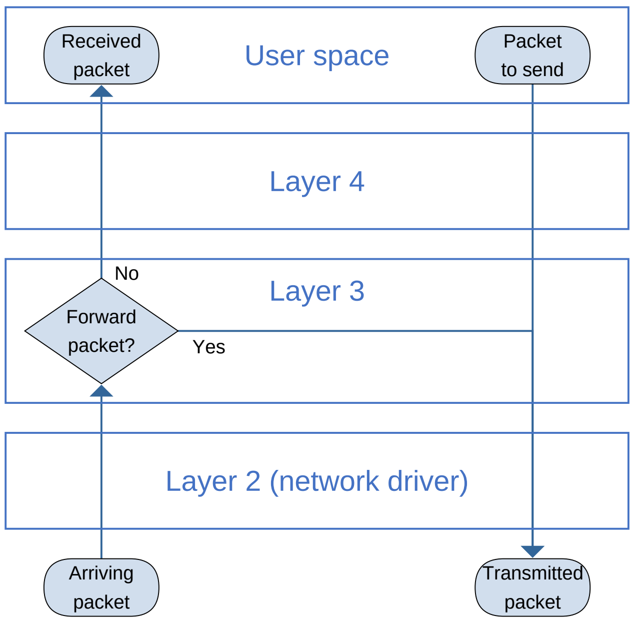


Figure 3.3: Example how a received/sent IP packet traverses the Linux network stack

packet was received, and the EtherType of the packet (e.g., IPv4 or IPv6). Some fields of the `sk_buff` are filled by the subsequent layers. When layer 2 finished processing the received packet, it passes the packet to layer 3.

The third layer validates the IP header and decides whether and to which destination the packet must be forwarded. If the packet must be forwarded, layer 3 modifies the packet as needed (e.g., decrement the TTL) and passes it back to layer 2 in order to transmit it. If the packet is for the local host (i.e., it must not be forwarded), layer 3 passes the packet to layer 4.

Layer 4 is the transport layer. It analyzes the TCP or UDP header to match the socket that is responsible for the packet. Each user space process that is listening for packets opens a socket so that the kernel can keep track of the connection. Based on the matching socket, the kernel passes the packet to the corresponding process in the user space.

Note that packets that are only forwarded by the kernel do not leave the kernel space. Sending packets from local processes works in the opposite direction where layer 4 receives the packet from the user space process. Layer 4 then passes the packet to layer 3 which in turn passes the packet to layer 2.

3.2.3 Bottlenecks

The kernel developers try to keep the networking stack in the Linux kernel as generic as possible so that it fits all needs. For example, the `sk_buff` structure contains metadata that is only needed by a few protocols [10]. As a result, there is a performance overhead that makes the network stack slower.

García-Dorado *et al.* [11] say that the allocation and deallocation of the `sk_buff` structure takes a large amount of time. According to them, the `sk_buff`-related operations consume 63 % of the CPU usage during the reception of a packet that is 64 bytes large. Moreover, the kernel copies the whole packet before modifying it, for example, to decrement the TTL [10].

If the kernel passes a packet to the user space for processing, additional overhead occurs. To pass data to the user space, the kernel must copy the packet. As a result, the kernel copies a packet at least twice. First, the network driver copies the packet after its arrival from the DMA-able memory region to a packet buffer in the kernel. Then, the kernel copies it from the packet buffer to a buffer of the user space application [11]. To run the user space application, context switches are required that produce additional overhead [11].

3.2.4 Netfilter

Netfilter is one of the firewalls that we compare in Section 5.1. Thanks to Netfilter, the Linux network stack also provides firewall functionalities. Netfilter is a framework in the Linux kernel that provides hooks to which other parts of the kernel can subscribe. These hooks trigger when a packet arrives at certain points in packet processing. Two popular projects build upon Netfilter: iptables and its successor nftables.

3.2.4.1 Hooks

Netfilter provides multiple hooks that trigger at different points in packet processing [10]. For example, there is a hook that triggers before the kernel makes a routing decision (prerouting hook). Another hook triggers only for packets that are forwarded (forward hook). Figure 3.4 shows the available hooks for IP packets.

Other parts of the kernel can subscribe to these hooks by defining a callback function. If a hook triggers, the kernel executes all callback functions that have subscribed to the hook. Consequently, the callback functions can react to the event. For example, a callback function can manipulate (mangle) the packet or tell Netfilter to drop the packet.

3.2.4.2 iptables and nftables

iptables [12] is a popular project to configure packet filtering rules. It is a stateful packet filter (see Section 3.1.3.2) and therefore allows to configure ACL rules. However, iptables has more features than a stateful packet filter. For example, it supports configuring Network Address Translation (NAT) rules and it can mangle packets.

For its operation, iptables subscribes to the hooks from Netfilter. As a result, filtering (as well as NAT and mangle) happens at multiple points, the hook points, in packet processing as shown in Figure 3.4. The filtering rules can be configured individually for each hook. iptables calls the different sets of filtering rules *chains*. There are five chains: `PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT`, and `POSTROUTING`.

iptables consists of a user space application for configuration and a kernel part to subscribe to the hooks and process the configured rules. Besides the iptables user

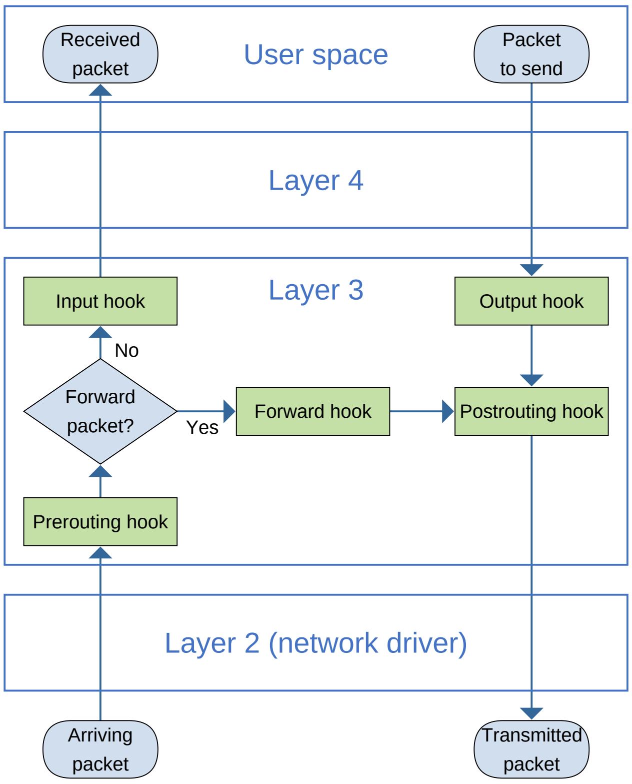


Figure 3.4: Netfilter hooks (green) in the Linux network stack at the example of an IP packet

space application, there are more applications, for example, ip6tables to configure IPv6 rules.

The successor of iptables is nftables [13]. It also relies on Netfilter and its hooks. As well as iptables, nftables is split into two parts: A user space application called nft and a counterpart in the kernel. One advantage of nftables is the unification of IPv4, IPv6, and other protocols in one user space application (as opposed to iptables where multiple applications exist). To achieve this, nft compiles the rules into bytecode and passes rhe bytecode to nftables in the kernel. When a Netfilter hook triggers, a virtual machine from nftables in the kernel executes the bytecode that contains the rules configured for the triggered hook.

3.2.5 eBPF

The Linux kernel offers a flexible technology called eBPF (extended Berkeley Packet Filter) [14] that allows running custom programs in the kernel. An example of an application that makes use of such eBPF programs is bpf-iptables, which we introduce in the next section. The kernel loads eBPF programs without the need of changing the kernel source code or the need to load kernel modules. eBPF programs run in a sandbox in the kernel and are written in a “pseudo-C code” [15]. They can be used to extend the kernel functionality during runtime and without changing the kernel source code. In addition, ePFB provides the functionality to exchange data between the eBPF program and a user space application.

eBPF programs do not run the whole time after they were added to the kernel. Instead, they are triggered by hooks. In contrast to Netfilter, these hooks are not limited to the network stack. eBPF hooks can be found all over the kernel, for example, it is possible to hook system calls.

There are three eBPF hooks in the network stack [16, 17]. Two of these hooks are part of traffic control (tc). The first tc hook triggers just after an arriving packet leaves the network driver (i.e., the `sk_buff` was already allocated). The second tc hook triggers just before a departing packet enters the network driver. The third hook is an eXpress Data Path (XDP) hook. It triggers right after the packet was copied into the DMA memory region. At this point, the kernel did not allocate an `sk_buff` and did not perform any expensive processing. This makes the XDP hook suitable for high-performance applications, such as Distributed Denial of Service (DDoS) mitigation, because packets can be dropped before they cause a high amount of processing. For example, Bertin [18] integrated XDP into their DDoS mitigation infrastructure.

However, to ensure that an eBPF program is safe to run, the kernel validates the eBPF program when loading it. As a result, there are some limitations of the eBPF programs [16]. First, the eBPF program must not contain loops (except loops with a static number of iterations). Second, the length of the eBPF program is limited. These limitations constrain the applications where eBPF can be used. Sometimes, workarounds are possible. For example, multiple eBPF programs can be chained together if one program gets too large.

3.2.6 bpf-iptables

Based on eBPF, Miano *et al.* [17] developed bpf-iptables. We considered using bpf-iptables as firewall for this thesis. Therefore, we explain the advantages and disadvantages of bpf-iptables in Section 5.1.

bpf-iptables emulates the iptables syntax and functionality using eBPF. However, it only supports a subset of the iptables features. For example, bpf-iptables does not support NAT and mangling packets.

To emulate iptables filtering, bpf-iptables provides a user space application that accepts commands in the iptables syntax to configure the filtering rules. The user space application transforms the configured filtering rules into multiple eBPF programs. After that, the user space application loads the eBPF programs into the kernel.

iptables allows configuring rules in different chains (see Section 3.2.4.2). These chains correspond to different Netfilter hooks. However, eBPF does not provide these hooks (see Section 3.2.5). To reproduce the iptables chains, bpf-iptables must emulate the Netfilter hooks. bpf-iptables does that by using the tc and XDP hooks of eBPF. Additionally, the XDP hook brings the advantage that it is more efficient than Netfilter hooks. When the XDP hook triggers, the kernel has not yet allocated an `sk_buff` for the packet. If a packet matches a filtering rule in the `FORWARD` chain with the action allow, bpf-iptables directly passes the packet to the target NIC. In that case, the packet skips the rest of the Linux network stack. Therefore, the kernel does not allocate an `sk_buff`.

bpf-iptables not only emulates iptables using eBPF. It also improves the matching algorithm. iptables traverses the filtering rules linearly for matching. To reduce the duration for rule matching, bpf-iptables uses a Linear Bit-Vector Search. According to Miano *et al.* [17], the improved matching algorithm enables a 64x speedup on common CPUs.

3.3 User space networking

In the last section, we explained how networking in the kernel works and pointed out multiple bottlenecks. To overcome those bottlenecks, user space I/O frameworks emerged. Their goal is to increase the throughput and reduce the latency. For this, the user space I/O frameworks completely bypass the kernel network stack and perform all packet processing in user space applications.

We use two user space I/O frameworks in this thesis: DPDK and FD.io VPP. FD.io VPP is the firewall we modify in this thesis. DPDK is the framework that FD.io VPP uses to fetch packets from the NIC. To give an overview of user space I/O frameworks, we first explain the general techniques of user space I/O frameworks in this section. After that, we introduce DPDK and FD.io VPP.

3.3.1 Techniques to improve the performance

User space I/O frameworks use several techniques to increase the throughput and to reduce the latency compared to the kernel network stack. In the following, we explain the most important techniques that were identified and evaluated on their performance by Barbette *et al.* [19]. The frameworks we use in this thesis (DPDK and FD.io VPP) implement all of these techniques.

Kernel bypass User space I/O frameworks can bypass the kernel network stack completely. They can use their own, optimized networking implementation to achieve better performance. This way, they also prevent expensive system calls that are required to transfer packets from the kernel space to the user space.

Zero-copy As we already explained in Section 3.2.3, the Linux kernel copies packets multiple times. Such copy operations take a large amount of time. Therefore, many user space I/O frameworks do not copy packets from the DMA memory region. Instead, the frameworks use references to the packet data in the DMA memory region during the whole packet processing.

Polling The usual way to get notified about new packets is to wait for interrupts from the NIC. The Linux kernel extended this mechanism with the NAPI by using polling at high packet rates to reduce the overhead caused by the interrupts (see Section 3.2.1). To further improve the performance, some user space I/O frameworks completely avoid interrupts. Instead, they poll the ingress queue all the time to fetch packets as soon as they arrive. This comes with the disadvantage of a constant CPU utilization of 100 % because the CPU is always busy polling the ingress queue.

Batched I/O Many user space I/O frameworks process packets in batches to reduce the processing overhead. For example, they reduce the overhead for accessing the NIC this way. Moreover, batching allows using the vector instructions of modern CPUs. We discuss this topic more detailed at the example of FD.io VPP in Section 3.3.3.5.

Multiple ingress queues Modern NICs support multiple ingress/egress queues. Thus, user space I/O frameworks can instruct the NIC to distribute arriving packets on multiple ingress queues. A common technique to distribute packets on multiple ingress queues is called Receive Side Scaling (RSS). With RSS, the NIC hashes the 5-tuple of the packet. Based on the resulting hash, the NIC selects a queue in which it places the packet. As a result, the user space I/O framework can distribute the packet processing on multiple CPU cores (one CPU core per ingress queue).

3.3.2 DPDK

In this thesis, we modify FD.io VPP to implement the ideas we described in Section 2.4. FD.io VPP, in turn, relies on DPDK to communicate with the NIC. Hence, we give a brief overview of DPDK in this section and introduce the generic flow API that is important for the priority idea.

3.3.2.1 Overview

DPDK stands for Data Plane Development Kit. Intel created DPDK in 2010. Since 2017, it is part of the Linux Foundation. However, Intel is still the main contributor. DPDK is available as open source under the BSD-3 license[1].

DPDK runs in user space. It applies the techniques that we explained in the last section to increase the throughput and reduce the latency compared to the Linux kernel. It bypasses the kernel and communicates directly with the NIC. Since DPDK polls the ingress queue all the time, it causes a CPU utilization of 100 %.

Essentially, DPDK provides efficient NIC drivers and abstracts different NIC models (see [20] for supported NICs). Applications using DPDK can configure various NIC models using the same interface. Moreover, an application using DPDK can receive packets and send packets regardless of the NIC model. DPDK also abstracts the metadata that a NIC creates when a packet arrives. The metadata contains, for example, information whether the NIC validated the IPv4 checksum. As a result, an application using DPDK can analyze the metadata independently of the NIC model. The same happens when an application transmits a packet using DPDK. The application sets metadata for a packet, for example, to instruct the NIC to set the VLAN tag. DPDK translates this metadata into metadata for the specific NIC model.

3.3.2.2 Generic flow API

Many NICs that DPDK supports can match packets in hardware based on configurable rules. DPDK offers libraries to configure such rules on the NICs. One of these libraries is the generic flow API [21]. We use the generic flow API for hardware classification on the NIC as part of the priority idea, as we explain in Section 6.4.3.

The generic flow API allows configuring so called *flow rules* on the NIC. A flow rule consists of *attributes*, a *matching pattern*, and *actions*. The NIC matches the packets based on the attributes and the matching pattern. If a packet matches the matching pattern, it executes the defined actions. Attributes can be used, for example, to group flow rules to a table or to specify the direction in which a flow rule is applied (ingress or egress).

Actions instruct the NIC on how to process the packet if it matches. There are many actions available, for example:

- `QUEUE`: Place the packet in the given queue
- `DROP`: Drop the packet
- `COUNT`: Count the number of packets matching the flow rule
- `RSS`: Enable Receive Side Scaling for matching packets using the given hash function and queues

[1]Some components, such as Linux kernel components, are licensed under the GPL-2.0 or LGPL-2.1 license

Each flow rule includes a matching pattern that the NIC uses to match the packets. A matching pattern consists of one or more *pattern items* of which many different types are available. Some *pattern item types* are, for example:

- `INVERT`: Match all packets that do not match the pattern
- `ETH`: Match the Ethernet header
- `VLAN`: Match the VLAN tag
- `IPV4`: Match the IPv4 header
- `UDP`: Match the UDP header
- `RAW`: Match a given byte string

Each pattern item type usually accepts fields that correspond to its type. For example, the `ETH` type accepts, among others, a source MAC address and a destination MAC address for matching. The pattern item types can be combined to a matching pattern as required. For example, they can be combined in the order `[ETH, IPV4, UDP]` to match a 5-tuple as with an ACL. If the predefined item types, like `ETH`, are not sufficient, the `RAW` item type can be used. The `RAW` item type accepts a byte string that the NIC matches with the packet. In addition to this byte string, the `RAW` type accepts the following values:

- A start position that defines at which byte in the packet the NIC starts matching
- A search area around the start position if the exact position is unknown
- A bitmask to exclude certain bits from matching
- A second byte string (called "last") to match ranges, for example, a port range (the main byte string defines the start port and the last byte string defines the end port)

3.3.2.3 Generic flow API limitations

However, the generic flow API comes with some limitations. These limitations arise because not every NIC supports all features that the generic flow API offers. In the following, we discuss these limitations at the example of the Intel I210 Ethernet controller that the NIC model of our firewall uses.

The Intel I210 controller offers a smaller feature set compared to other controllers. It utilizes DPDK's IGB poll mode driver, so we can look at the source code of the poll mode driver to see which generic flow API features the controller supports [22]. In addition, we looked at the data sheet of the Intel I210 controller [23].

The source code of the IGB poll mode driver boils down all supported features of the generic flow API to five filters. These filters are configured with the same attributes, pattern item types and actions from Section 3.3.2.2, but the I210 controller is very picky regarding the combination of attributes, pattern item types, and actions it accepts. As a result, all of these filters are very restricted in their feature set. In the following, we give an overview of all five filters.

2-tuple filter The 2-tuple filter matches the protocol field in the IP header and the destination port (only one port, no port range). To create a 2-tuple filter flow rule, the pattern item types must be combined as in the following example: `[ETH, IPV4, UDP]`. All fields other than the protocol field and destination port field must be left empty.

Ethertype filter The ethertype filter filters the EtherType field in the Ethernet header, therefore the `ETH` pattern item type is used (with the EtherType field filled).

TCP SYN filter The TCP SYN filter matches packets that have the TCP SYN flag set. To create a TCP SYN filter rule, the pattern item types must be combined as in the following example: `[ETH, IPV4, TCP]`. All fields must be left empty except the TCP flags field where the TCP SYN flag must be set.

Flex filter The flex filter uses the `RAW` pattern item type and is therefore equal to its description in Section 3.3.2.2. However, there are a few limitations. The I210 controller does not allow real bitmasks. Instead, the bitmask is actually a bytemask to define whether a byte of the pattern is matched or ignored. Additionally, it is possible to concatenate multiple `RAW` pattern items (`[RAW, RAW, ...]`), but the I210 controller can only match the first 128 bytes of a packet. Lastly, the I210 controller does not support ranges. This means that, for example, only one port number and no port range can be matched.

RSS filter The RSS filter is no actual filter on the I210 controller. Instead, it is only the `RSS` action that must be configured without any pattern items. As a result, the NIC distributes all arriving packets among the given queues (i.e., no filtering is possible).

In addition to the above limitations, there are a few limitations that apply to all these filters. The I210 controller only allows to configure eight flow rules per filter and port. For the TCP SYN and RSS filter, the I210 controller allows one flow rule (this is no limitation in practice because more than one of these flow rules do not make sense anyway). According to the data sheet, the I210 controller only supports four flow rules for the ethertype filter [23] while DPDK allows configuring eight of them. We did not try what happens if we configure more than four flow filter rules for the ethertype filter. Another limitation is that all filters only support the `QUEUE` action, except the RSS filter, which is configured using the `RSS` action. Lastly, all flow roles can only be configured in ingress direction because the I210 controller does not support egress flow rules.

In conclusion, we realize that the I210 controller limits the feature set of the generic flow API significantly. We cannot even configure ACL rules anymore because the I210 controller only supports 2-tuples. As workaround to enable the configuration of ACL rules, we use the flex filter later in this thesis (see Section 6.4.3). With the flex filter, we can recreate an ACL with a limited feature set (e.g., no IP and port ranges).

3.3.3 FD.io VPP

FD.io VPP is the firewall we chose to implement the ideas described in Section 2.4. Therefore, this subsection is of special importance because the technical details explained here are relevant throughout the thesis. All information about FD.io VPP and our modifications discussed in this thesis are based on version 21.10. The corresponding code is available in FD.io's self-hosted Git repository [24] or as a mirror on GitHub [25].

3.3.3.1 Overview

FD.io VPP [26], in the following called VPP, is an "extensible and modularized software framework" [27]. It runs in user space, bypassing kernel networking, and is written in C. VPP can be used for almost any networking application, for example, as a router, switch, virtual network function, or in our case, as a firewall. Even commercial products like Netgate TNSR [28] and Cisco IOS XRv 9000 [29] use VPP internally. It is part of the Fast Data I/O project (FD.io, usually pronounced "Fido") that in turn is part of the Linux Foundation.

VPP is an open source project, licensed under the Apache 2.0 license. Cisco started its development as a closed source project in 2002 to use it in Cisco products [30, 31]. In 2016, the FD.io collaborative open source project was launched under the Linux Foundation with VPP being a part of it [27, 32]. Since then, VPP is open source but Cisco is still the main contributor.

VPP is an abbreviation and stands for "Vector Packet Processing" or "Vector Packet Processor". Vector processing is the novelty of VPP [31]. Instead of processing one single packet at a time, VPP processes packets in batches, so called vectors, to increase the packet throughput. The goal of batching packets in vectors is to use the available processing power more efficiently. It does so by leveraging the instruction and data caches as well as vector instructions (e.g., Intel SSE). Using the caches as much as possible ensures that the pipelines of a CPU are always full and do not have to wait for data to arrive from the memory.

3.3.3.2 Features

VPP is meant to be a *data plane*, that is, its purpose should be primarily packet forwarding [27]. A dedicated controller should take over the *control plane* tasks like the configuration of forwarding rules. For remote configuration, VPP exposes a binary API for which client libraries are available in different programming languages like C or Python. In addition, VPP offers a so called "debug CLI" that is primarily meant to be used for development or quick testing. The CLI lacks some features that are only accessible via API (e.g., deleting ACL rules).

By default, VPP brings many features like NAT, Internet Protocol Security (IPsec), or Dynamic Host Configuration Protocol (DHCP). If a feature is missing, one can add it using the flexible plugin system. Many native features are plugins themselves, as well as the firewall functionality that is part of the ACL plugin.

Plugins can make use of many available data structures and functions offered by VPP. For example, VPP offers dynamic arrays (also called vectors) that are used all over in the VPP code. Some other data structures are built on top of these vectors, such as bitmaps and ring buffers. They can also be used by plugins.

NIC drivers are implemented as plugins too. One of them is the DPDK plugin that uses DPDK to fetch packets from the NIC using its poll mode drivers. But there are also other drivers available since VPP existed before DPDK was released. For example, there are also drivers for virtual interfaces that exchange data using shared memory (memif). In this thesis, we use the DPDK driver because it works well together with our NICs and because it supports polling. VPP also supports interrupt-driven packet processing and an adaptive mode. The latter switches between poll and interrupt mode automatically based on the number of arriving packets.

3.3.3.3 Packet processing graph

Another fundamental idea of VPP, besides vector packet processing, is its graph architecture. Each feature, like packet reception, IPv4 packet validation, or IPv4 forwarding is an individual graph node. The packets traverse different graph nodes based on their content (e.g., IPv4 packets traverse the IPv4 nodes). VPP tries to keep the packets batched as vectors all the time but splits them as required to traverse the graph. Figure 3.5 shows an extract from VPP's packet processing graph. VPP differentiates between four node types: pre input nodes[2], *input nodes*, *internal nodes*, and *process nodes*.

Input nodes interact with the NIC drivers to fetch the packets from the NIC. In other words, they are the starting node for most vectors (except if VPP generates packets itself). The only input node in Figure 3.5 is the `dpdk-input` node that abstracts DPDK.

Internal nodes are traversed after an input node or after another internal node. Each performs small portions of packet processing like IPv4 packet validation (`ip4-input` node in the graph). All nodes in Figure 3.5 except the `dpdk-input` node are internal nodes. They can also be leave nodes (i.e., output nodes) that hand over packets to the NIC to transmit them. But not every leave node transmits packets, such as the `error-drop` node that drops packets.

Process nodes are independent of the packet processing graph. They do not take part in packet processing. Instead, they perform organizational tasks like controlling the CLI and API. Because of that, they are also not shown on the graph in Figure 3.5.

The edges between graph nodes and the nodes themselves are dynamic. During compile time and also during runtime it is possible to register new graph nodes using plugins. Plugins can insert nodes at any position to implement a new packet processing feature. For example, the ACL plugin inserts several nodes, as we show later in Section 3.3.3.8.

3.3.3.4 Graph traversal example

As an example, we traverse the packet processing graph in Figure 3.5. Initially, no vectors are in the graph and therefore, VPP continuously runs the `dpdk-input` node

[2]Pre input nodes are rarely used and therefore not discussed here.

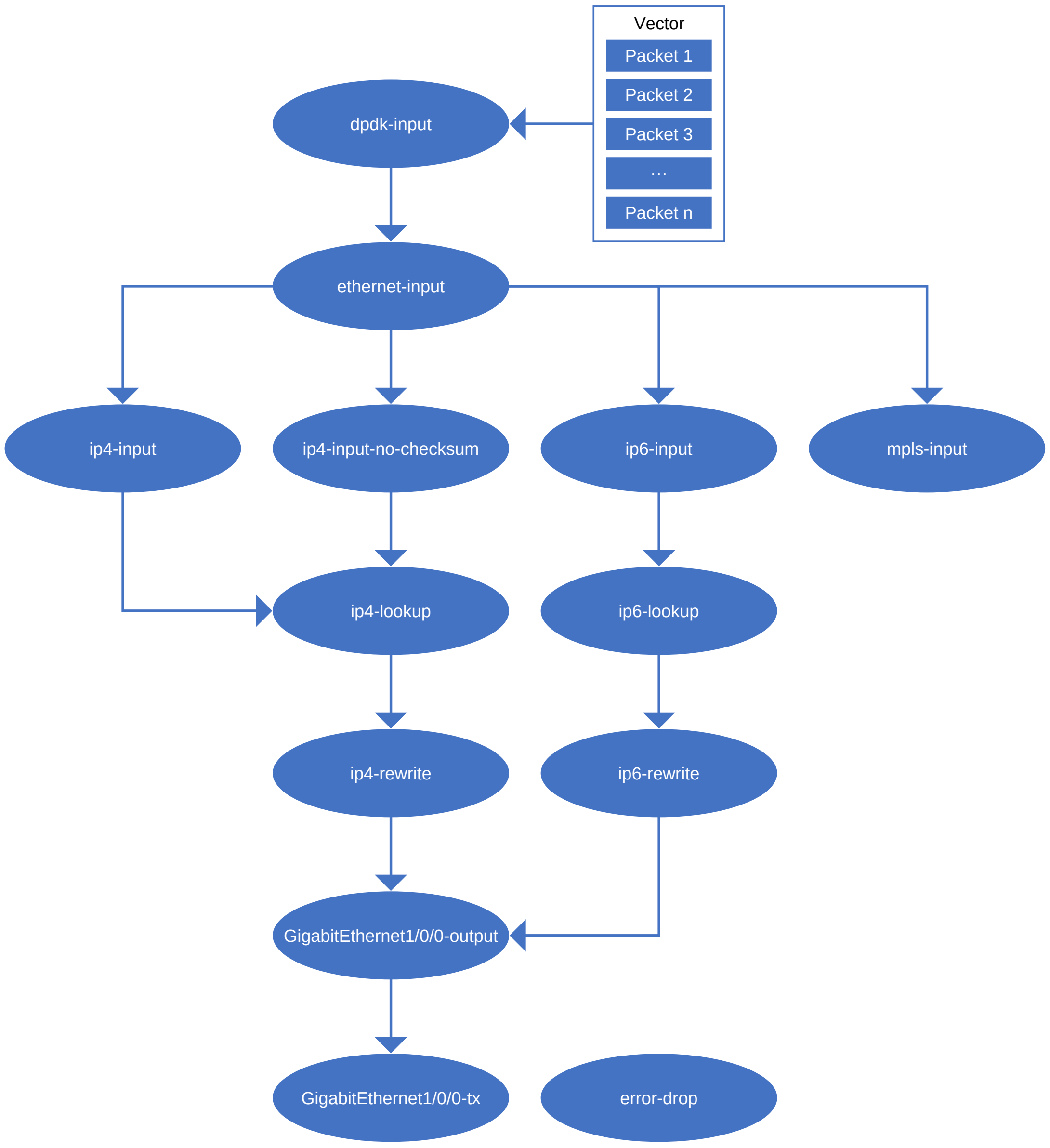


Figure 3.5: Extract from VPP's packet processing graph (note that the `mpls-input` node subgraph is incomplete and that every node can have an edge to the `error-drop` node)

to poll the ingress queue. In practice, `dpdk-input` can poll multiple interfaces (one after another). But to keep it simple, we assume we have one interface and one queue in this example. Every time, `dpdk-input` runs, it tries to fetch packets from the ingress queue. If packets are available, `dpdk-input` fetches up to 256 of them per queue as this is the maximum allowed vector size. According to Barach *et al.* [31], a vector size of 256 is optimal for a balance between throughput and latency. `dpdk-input` then performs some preprocessing like converting DPDK-specific information about each packet to VPP data structures. This allows other nodes to read that information (e.g., whether the checksum of a packet is valid). The `dpdk-input` node then decides to which node each packet must be forwarded. If the NIC does not already determine the protocol carried in the frame, `dpdk-input` forwards the packets to the `ethernet-input` node. All packets that are destined for the same node are packed into the same vector. We assume for our example that all packets are packed into one vector and forwarded to the `ethernet-input` node.

The `ethernet-input` node checks the EtherType field in the Ethernet header of each frame to see which protocol it carries. It also checks whether the NIC already validated the packet's IPv4 checksum. If the NIC did not validate the checksum, the next node is the `ip4-input` node, otherwise the `ip4-input-no-checksum` node. Based on the result, `ethernet-input` places the packets in different vectors and forwards them to the appropriate nodes. Thus, vectors must not traverse the whole graph in one piece. They can be separated into multiple vectors and traverse the graph independently. For example, the vector can contain IPv4 and IPv6 packets. Hence, `ethernet-input` creates two vectors, one for the `ip4-input-no-checksum` node and another one for the `ip6-input` node. When `ethernet-input` processed all packets from the vector, VPP runs the next node to which a vector got forwarded.

In our example, VPP runs the `ip4-input-no-checksum` node to process the vector forwarded to that node. When all packets were validated successfully, `ip4-input -no-checksum` forwards the vector to the `ip4-lookup` node. Since we still have a pending vector at the `ip6-input` node, VPP runs this node next. VPP continues to run the nodes with pending vectors until all vectors traversed the graph. At this point, all packets were either transmitted or dropped and VPP can run the input nodes again.

3.3.3.5 Advantages of vector packet processing

The goal of vector packet processing is to make packet processing more efficient in order to increase the packet throughput. Some of the main VPP developers, Barach *et al.*, discuss and evaluate the advantages in detail in their article [31]. In this section, we want to give a brief overview of the advantages.

Less instruction cache misses The breakdown of features into nodes splits packet processing into small tasks. These small tasks fit into the instruction cache of the CPU if they are small enough. When VPP is running the node, the CPU loads the instructions from the slow memory into the instruction cache while processing the first packet. The next packets require exactly the same instructions. Thus, the processing of the following packets is faster because the CPU can load the instructions from the cache instead of the memory. Effectively, VPP distributes the overhead caused by loading the instructions on multiple packets and therefore reduces the overhead in total.

Less data cache misses If VPP processes packets in batches, it knows which packet comes next. This allows prefetching the data of the next packet while the current packet is still processed. Data prefetching means that data is loaded from the slow memory into the fast cache before it is needed.

For example, VPP can use prefetching when it verifies the IPv4 checksums of all packets in a vector. While verifying the checksum of the first packet in the vector, it can already load the checksum of the next packet in the vector from memory into the cache. And while checking the checksum of the second packet, it can load the checksum of the third packet and so on.

Prefetching ensures that the data for the next packet is available as fast as possible so that the CPU does not have to wait. Because batch processing of packets is implemented using loops in the code, the compiler can automatically insert prefetching instructions. However, in some cases, prefetching calls are also manually inserted in the VPP code.

Mitigate node switching overhead VPP's graph architecture introduces additional processing overhead. When a node finished processing, VPP must determine the node to run next. In addition, VPP must call the node's entry point function to run it which introduces an additional function call. VPP uses inline functions in place of normal functions as often as possible to prevent processing overhead caused by function calls. But this is not possible here because we cannot determine at compile time which node is run after another node. Vector packet processing requires only one call of a node's entry point function for multiple packets. As a result, the overhead caused by a node switch is distributed on multiple packets, which reduces the total overhead.

Other optimizations Processing similar data in batches allows some general optimizations on today's CPUs. The first optimization is manual loop unrolling. For example, when iterating over 100 elements, VPP does not iterate over one element at a time. Instead, VPP iterates over two or four elements at a time in some loops. Figure 3.6 shows an example without and with loop unrolling. To use loop unrolling, the code must be organized manually as shown in the example. It is not done automatically by the compiler. Loop unrolling ensures that the CPU pipelines are always full. Otherwise, it can happen that the pipelines have to wait because of the branching caused by the loop.

The second optimization is the use of vector instructions. If an operation must be applied on many or all packets, it is inefficient to perform the operation for each packet one by one. Instead, VPP uses Single Instruction, Multiple Data (SIMD) instructions supported by today's CPUs. These instructions perform the operation in parallel for multiple packets at once. The number of supported vector operations is limited. But examples are assigning a constant value to multiple variables at once or performing arithmetic operations for multiple variables at once. In Intel CPUs,

```
int arr[100];

for (int i = 0; i < 100; i++) {
  arr[i] = i;
}
```

(a) Without loop unrolling

```
int arr[100];

for (int i = 0; i < 100; i += 4) {
  arr[i] = i;
  arr[i + 1] = i + 1;
  arr[i + 2] = i + 2;
  arr[i + 3] = i + 3;
}
```

(b) With loop unrolling

Figure 3.6: Example loop with and without loop unrolling

such SIMD instructions are called SSE and AVX.

All the advantages above come into play if the vectors are large. If the load is low (i.e., VPP does not have to process many packets), the vector size is small because the queues are empty. With increasing load, the vector size increases because VPP is still busy with processing packets while new packets arrive. The savings through batch processing prevent further fill up of the queue and the vector size settles at a certain value.

Thanks to vector packet processing, VPP achieves a high packet throughput. Barach *et al.* measured a maximum throughput of over 8 Mpps[3] [31] using one core of an Intel Xeon E52690 processor running at 2.60 GHz. Their scenario was to forward IP packets using a longest prefix matching lookup with a routing table size of 130 000 entries. The packet size was 64 bytes.

3.3.3.6 Node scheduling

VPP uses its own scheduling mechanism to decide which graph node should run next. The flowchart in Figure 3.7 shows the functionality of VPP's scheduling. We use it in the following to explain each step that is involved in scheduling.

Essentially, the scheduling consists of one loop that runs over and over. During the loop, VPP runs all nodes that have pending work in a certain order. Except for tasks that are required for scheduling, the main loop has no additional tasks. The real work (i.e. packet forwarding) is done in the nodes that are scheduled in the main loop.

[3]Mpps = Million packets per second

Input nodes First, the loop runs all input nodes one after another to fetch packets from the ingress queues of all NICs. As already explained in Section 3.3.3.3, the input nodes decide to which internal nodes the packets must be forwarded. Every input node creates the number of vectors that are needed to forward the packets. This means, at least one vector for each node to which a packet must be forwarded is created.

Until now, we just called the process of moving packets to the next node "forwarding". But it is no actual forwarding. Instead, the node places the vector in a "pending vectors" array that stores all existing vectors destined for any node. To remember to which node a vector belongs, vectors have an additional field to store the identifier of a node. In other words, the pending vectors array stores all vectors that VPP must process.

Inernal nodes When all input nodes finished, VPP iterates over the pending vectors in the pending vectors array. For each vector, VPP identifies the node to which the vector belongs, passes the vector to the node, and runs the node. The node can then process the packets in the vector. Most nodes forward the packets to other nodes because each node only takes over a small portion of packet processing. To forward the packets to other nodes, a node creates the required number of vectors and stores the packets in them. Then, the node stores the identifier of the next node in each vector and appends the vectors to the pending vectors array. VPP continues to iterate over the pending vectors until the array is empty and no more vectors are added. Nodes stop appending vectors when they handed the packets over to the NIC or when they dropped the packets. This means, all packets received from the input nodes were forwarded or dropped when the pending vectors array is empty and no packets remain unprocessed (except for packets that arrived at the ingress queue in the meantime).

Process nodes At this point, all input nodes and internal nodes had the chance to run (depending on whether vectors were forwarded to them). Only the process nodes did not have the chance to run yet. Process nodes do not process packets like input nodes and internal nodes. Instead, they perform organizational tasks during the runtime of VPP. Because of this, they consist of an infinite loop that prevents them from stopping in contrast to the input nodes and internal nodes.

The question is how process nodes can be paused after they were started if they loop for an infinite time. The answer is cooperative multitasking. In a cooperative multitasking environment, there is no external entity that stops tasks if they run for a too long time. Instead, a task must ensure to pause (*suspend*) itself when it is running too long. Exactly this is the case in VPP: process nodes must suspend themselves after a while to allow other nodes to run[4]. There are two ways how a process node can suspend. First, the process node can suspend for a certain period of time (the duration is arbitrary but must be longer than 1 µs). Second, the process node can suspend until it receives an event. Other nodes can signal an event to a

[4]Process nodes can really pause their execution anywhere within the loop and continue where they paused. But we do not discuss in this thesis how that works.

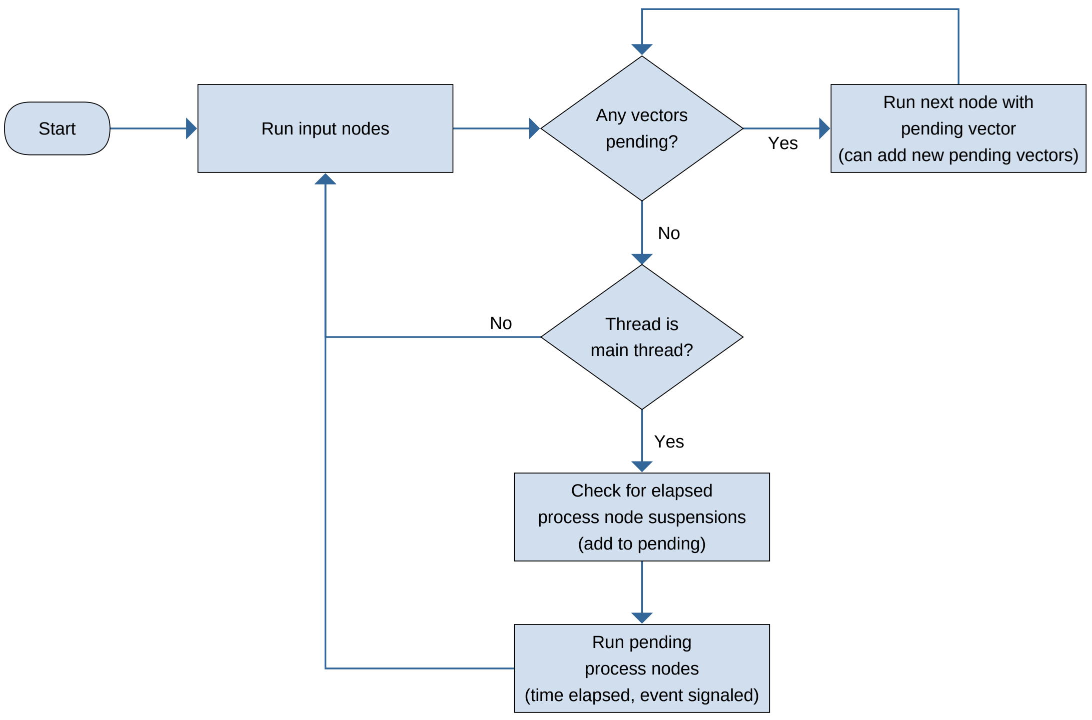

Start
Run input nodes
Any vectors pending?
Yes
Run next node with pending vector (can add new pending vectors)
No
Thread is main thread?
No
Yes
Check for elapsed process node suspensions (add to pending)
Run pending process nodes (time elapsed, event signaled)

Figure 3.7: Flowchart showing the VPP node scheduling mechanism

process node and also pass data together with the event. Such an event wakes the process node up again. But process nodes do not wake up immediately. Instead, the identifier of the process nodes is stored in a "pending process nodes" array to wake them up after running the internal nodes.

Now that we discussed how process nodes suspend and can be woken up again, we return to the flowchart in Figure 3.7. After no pending vectors are left, there is a check called "Thread is main thread?". We can ignore this check for now and assume that the answer is "Yes".

The next step is to check whether process nodes that suspended some time ago for a certain amount of time should wake up again. If a process node suspended for long enough, it is added to the pending process nodes array.

After that, VPP iterates over the pending process nodes array to run all process nodes that are scheduled to wake up again. Every process node runs until it suspends itself again. When all scheduled process nodes finished running, the loop starts from the beginning.

3.3.3.7 Multi-core support

In the last section, we explained how VPP schedules nodes. The scheduling and the nodes all run on the same CPU core in a single thread, meaning no nodes can run in parallel to process multiple packets at the same time. But VPP can also run on multiple CPU cores. In this case, VPP spawns multiple threads, one *main thread*, and multiple *worker threads*. The number of worker threads depends on the number of CPU cores. Every thread runs on a different CPU core and is independent of the other threads. They all run their own instance of the scheduling loop.

The main thread is responsible for organizational tasks like CLI and API while the worker threads perform packet processing. This brings the advantage that packet processing is not interrupted by organizational tasks, which reduces latency and jitter. Since organizational tasks run in process nodes, VPP does not run process nodes on the worker threads. VPP checks before scheduling the process nodes whether the scheduling loop runs on the main thread and only runs the process nodes in this case. The decision “Thread is main thread?” in Figure 3.7 shows this check.

VPP distributes the ingress queues of the NICs equally among the worker threads. This means, every ingress queue is processed by a different worker thread. If fewer worker threads than ingress queues are available, one thread processes multiple ingress queues. In that case, VPP tries to place at least the ingress queues that belong to the same NIC on different worker threads.

In addition, VPP assigns an egress queue of every NIC to each worker thread and to the main thread. If a NIC does not have enough egress queues, the egress queues are shared among multiple threads. This ensures that each worker thread can transmit packets over each NIC. However, shared egress queues reduce the performance because only one thread can access the egress queue at the same time. It is also possible to override the automatic ingress queue and egress queue placement. An administrator can manually specify the assignment of a queue to a thread using the CLI or the API.

Since no ingress queues are assigned to the main thread, there are no input nodes that the main thread must run. This means, the main thread must also not process any pending vectors and therefore only runs process nodes all the time. But there are some exceptions in which the main thread must still process pending nodes. For example, some CLI commands like the ping command lead to the transmission of packets. These packets are processed on the main thread because the CLI running on the main thread is responsible for the ping command.

3.3.3.8 ACL Plugin

Most of the modifications we make to the VPP code are located in the ACL plugin. Therefore, we want to give an overview of it in this section.

VPP offers multiple solutions that provide ACL functionality (ACL plugin, COP/ ADL plugin, Flow, Classifiers) [33]. But none of them except the ACL plugin support stateful rules which was a requirement. The other solutions also do not fit well together with our ideas (i.e., it would not be possible to implement them). Therefore, we will not further examine them.

Overview The ACL plugin supports stateful and stateless filtering. Unless stated otherwise, filtering functions like the stateless and stateful packet filters are described in Section 3.1.3.1 and Section 3.1.3.2. The ACL plugin can filter on the 5-tuple and TCP flags but also on the ICMP message and MAC address.

ACLs are assigned per interface and per direction. This means, we first create an ACL that contains any number of rules. Then, we assign the ACL to an interface in ingress or egress direction (an ACL can also be assigned to multiple interfaces/directions). Based on the direction, the ACL plugin filters packets when they arrive at the interface or when the packets are sent over the interface.

The ACL plugin supports three actions: permit, permit+reflect, and deny. To conform to our naming convention for ACL actions in the previous sections, we call these actions allow, allow+reflect, and block. Allow+reflect is used for the stateful feature of the ACL plugin. It tells the ACL plugin to track the connection of packets that match the corresponding rule. This is in contrast to Section 3.1.3.2 where the connections of all packets that have a matching rule with allow action were tracked.

ACL rules, their assignment to an interface and other options can be configured using the CLI and the API. But the CLI does not support all features (e.g., it is not possible to delete ACLs with the CLI).

Architecture The ACL plugin adds several internal nodes to the packet processing graph. Figure 3.8 shows the most important of them (they are highlighted in green). We omitted the nodes for layer 2 filtering because these are not relevant in this thesis. In the graph, we can identify two groups of ACL nodes. The first group consists of the `acl-plugin-in-ip4-fa` and `acl-plugin-in-ip6-fa` nodes. They perform ACL checks for ACLs assigned in ingress direction. The second group consists of the `acl-plugin-out-ip4-fa` and `acl-plugin-out-ip6-fa` nodes. They perform the ACL checks for ACLs assigned in egress direction. If no ACL rules were configured for a certain direction or protocol, the ACL plugin automatically disables the corresponding node. This reduces the node switching overhead because VPP must not run that node. In addition, the ACL plugin also adds a process node to maintain the connection table. For example, it deletes connection table entries if they time out.

All the packet processing graph nodes (except the layer 2 nodes) share the same code, even though they are different nodes. When a vector arrives at such a node, it traverses the node as the flowchart in Figure 3.9 shows. Essentially, the ACL plugin iterates over each packet in the vector. Each iteration begins with a check of the connection table to check whether the packet belongs to an active connection. If that is the case, the ACL plugin updates the corresponding connection table entry and allows the packet. The stateless ACL can be skipped.

If the packet does not belong to an active connection, the ACL plugin checks the ACL for a matching rule. After that and if the action of the matching rule is allow+reflect, the connection corresponding to the packet is added to the connection table. If no rule matches, the action is block which causes the packet to be dropped.

At the end of an iteration (stateful and stateless), the ACL plugin determines the node to which the packet must be forwarded next. If the action is block, the packet is forwarded to the `error-drop` node in order to drop the packet.

Stateful filtering The connection tracking mechanism is similar to the description in Section 3.1.3.2. Nevertheless, we want to mention one implementation detail because it is relevant for the performance: the connection table is stored as a hash table. This allows table lookups in a constant time because the ACL plugin must not iterate over every table entry. Instead, the ACL plugin extracts and hashes the

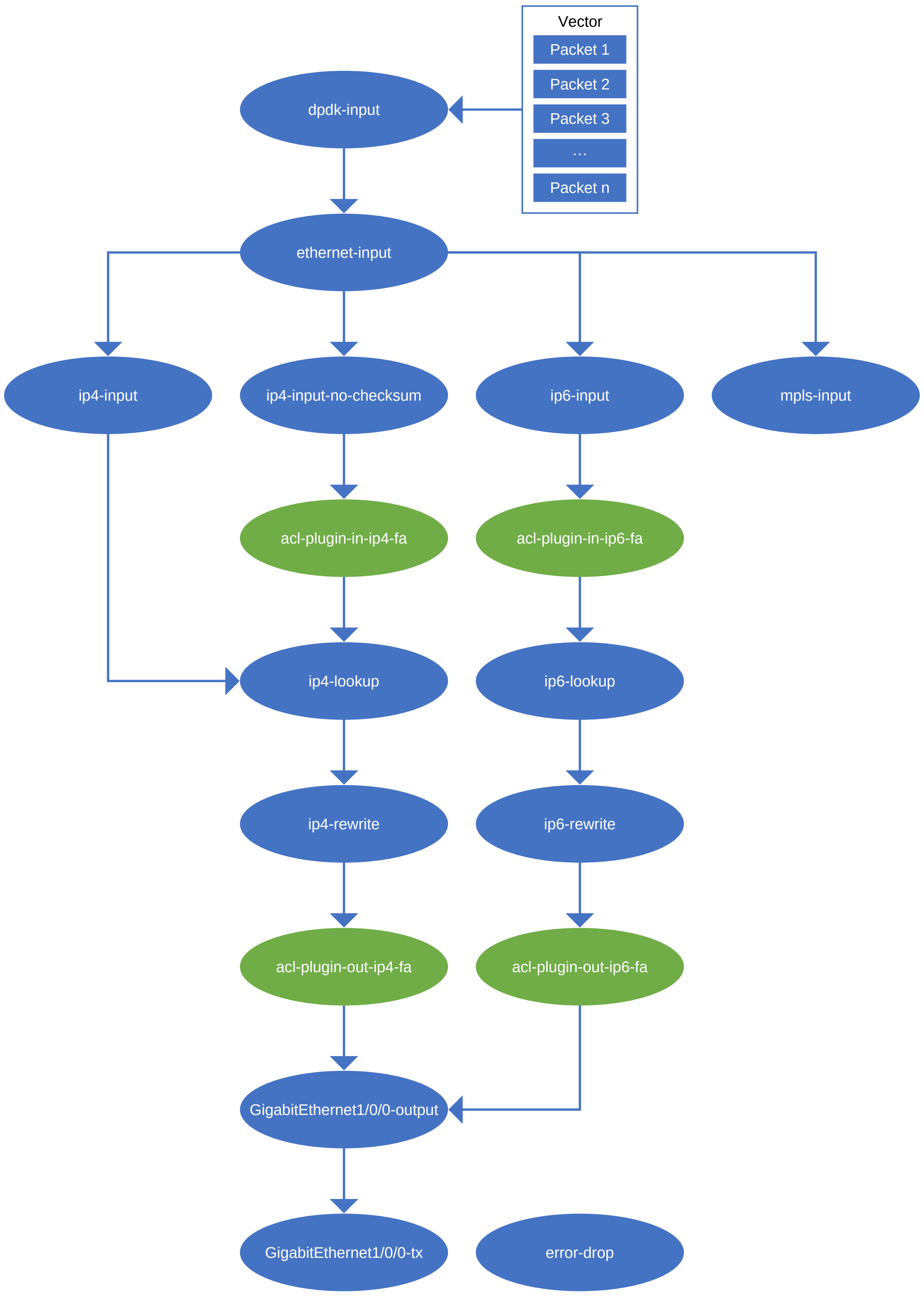


Figure 3.8: Extract from VPP's packet processing graph including a subset of the ACL plugin nodes in green (note that the `mpls-input` node subgraph is incomplete and that every node can have an edge to the `error-drop` node)

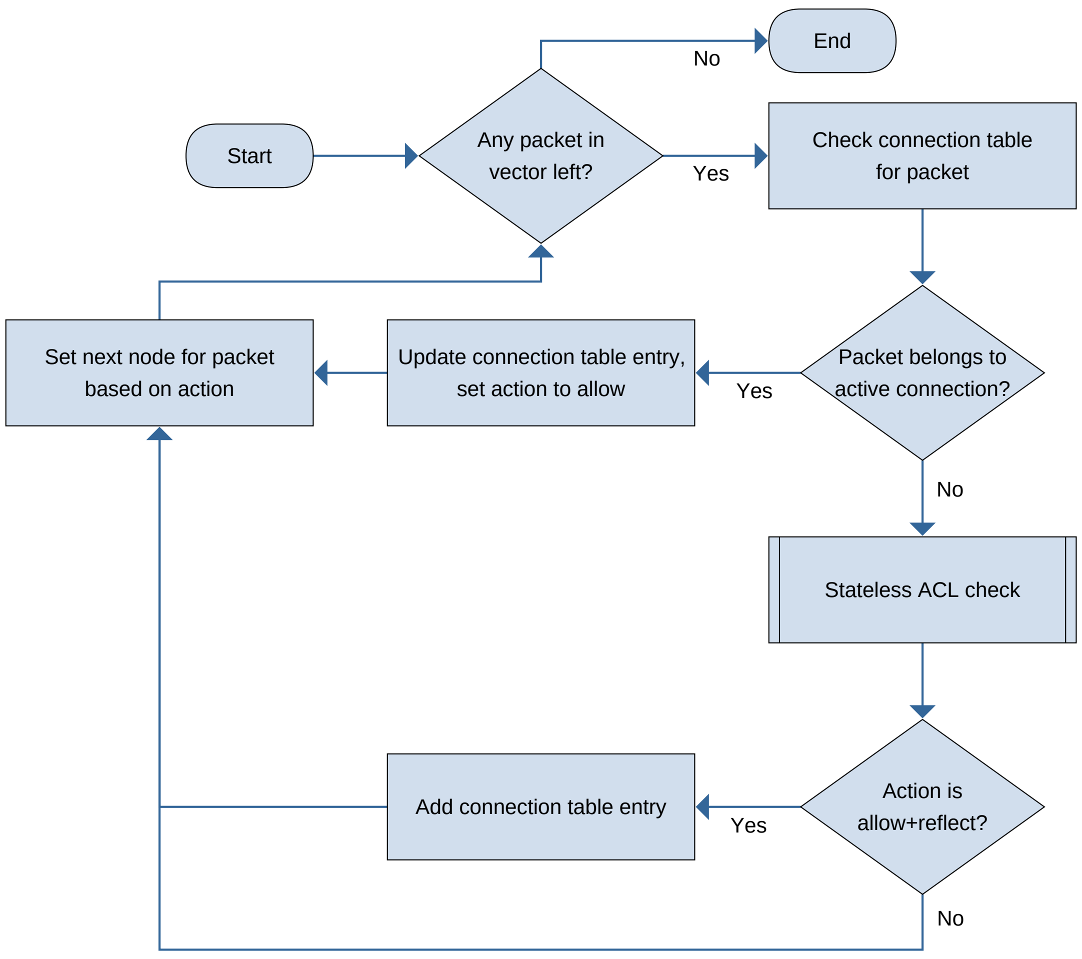

End
No
Start
Any packet in vector left?
Yes
Check connection table for packet
Set next node for packet based on action
Update connection table entry, set action to allow
Yes
Packet belongs to active connection?
No
Stateless ACL check
Add connection table entry
Yes
Action is allow+reflect?
No

Figure 3.9: Flowchart showing the ACL plugin structure

connection information from an arrived packet (the 5-tuple of the packet). It can then use the hash to directly access the corresponding table entry.

Stateless filtering If a packet does not have an entry in the connection table, it must be matched against the ACL rules. The flowchart in Figure 3.10 shows how the matching of the 5-tuple and TCP flags works. ICMP message filtering also takes place here but the protocol and the ports are replaced with ICMP-specific fields. MAC address matching, on the other hand, takes place elsewhere.

First, the ACL plugin gets all ACLs that are assigned to the interface in the corresponding direction. Then, it iterates over the ACLs and their rules. In each iteration over a rule, the ACL plugin checks whether the rule matches the packet. If the rule matches, the matching is finished and the action defined in the rule is applied. Otherwise, matching continues until no more ACLs and rules are left. In this case, the default block action is applied.

Next, we go into the details about how the ACL plugin matches a packet against an ACL rule. Since the flowchart in Figure 3.10 abstracts the matching of a single rule using the "Check ACL rule" process, we show the details of it in Figure 3.11. In short, all 5-tuple and TCP flags fields are compared one after another. If a field does not match, the rule does not match and the matching of the rule is stopped. If all fields match, the rule matches the packet and the action defined in the rule is used.

While we do not go through every field that is matched in detail because they are shown in Figure 3.11, we want to mention a few noteworthy matching operations. At the beginning of the matching, the ACL plugin compares whether the packet and the rule both have the same IP version (IPv4, IPv6) because IPv4 and IPv6 rules can be part of the same ACL. Since matching a rule with a different IP version does not make sense, it is checked first to cancel the matching.

After matching the source IP, the ACL plugin checks whether the rule defines a protocol to match (TCP, UDP, or ICMP). If that is not the case, the packet matches because the rule specifies no details about the protocol. Otherwise, if the rule defines a protocol, matching continues to match the protocol itself and the ports.

Rules that should not match a specific source and/or destination port must specify the maximum allowed port range (0 to 65 535). As a result, all source and/or destination ports match since they are within the allowed port range. The matching of TCP flags relies on a bitmask where ones mark the flag bits to match and zeroes mark flag bits to ignore during matching. If the rule does not define any required TCP flags, the bitmask is zeroed. In this case, the TCP flags matching is not omitted but always results in a match.

Hash-based vs. linear matching In the last section, we described a linear matching approach. This means, all rules are matched consecutively and the more rules are matched, the more time it takes. However, VPP also offers a hash-based matching approach. It is similar to the hash-based connection table but more complex because of the flexibility of the ACL rules (e.g., the ability to configure ranges of IP addresses and ports). The ACL plugin allows switching between the hash-based approach and the linear approach using a toggle during compile time and via API.

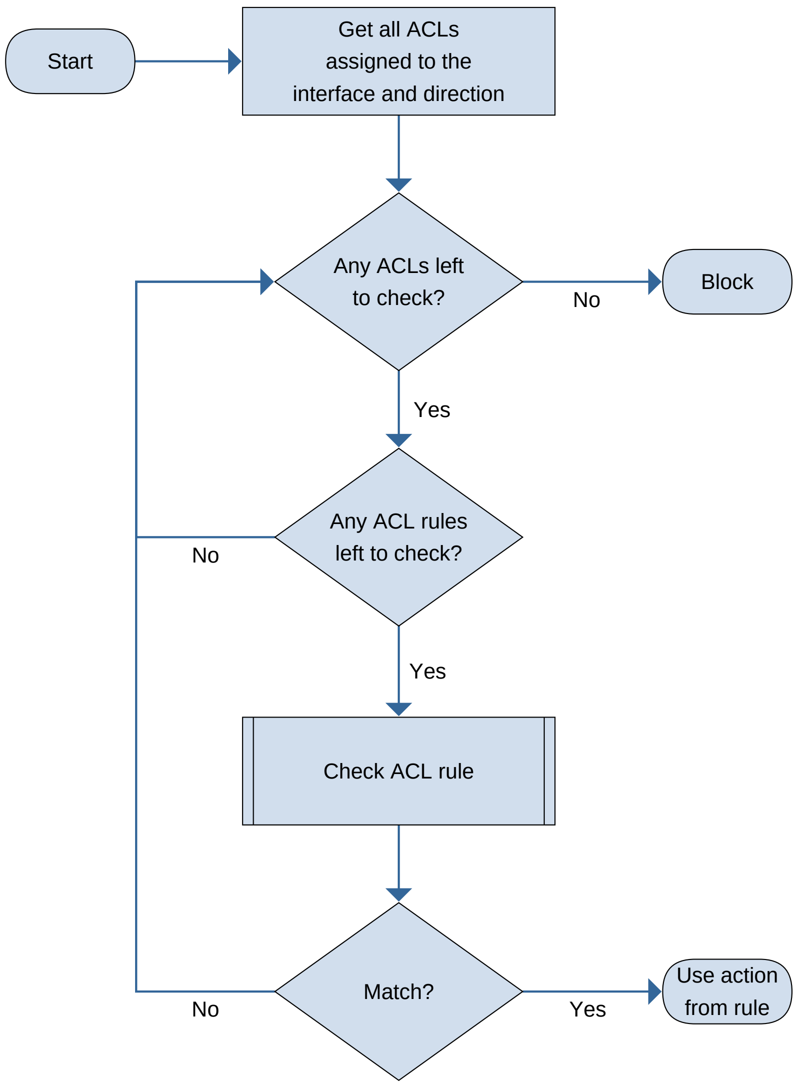


Figure 3.10: Flowchart showing the ACL plugin matching mechanism

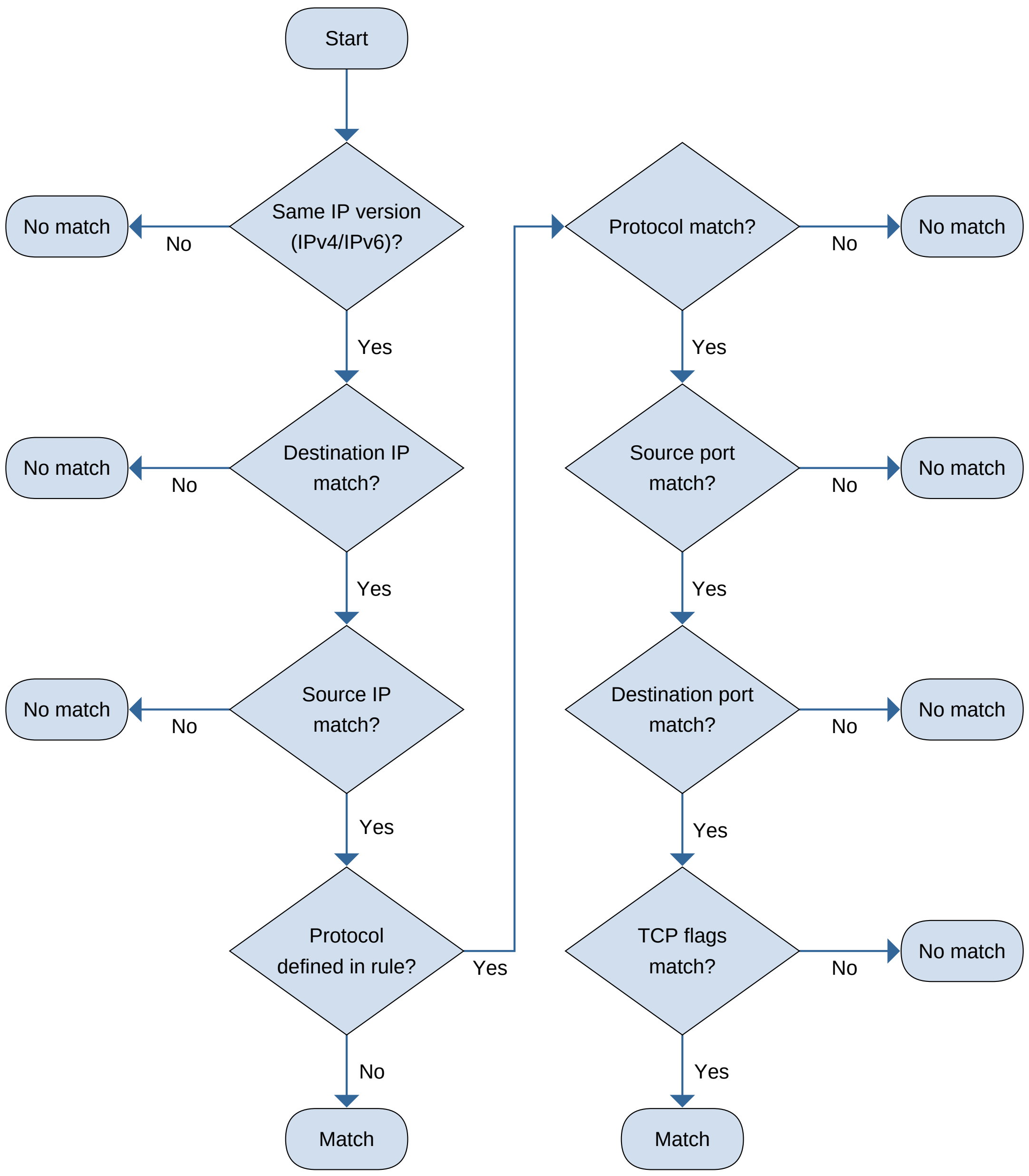


Figure 3.11: Rule matching mechanism of the ACL plugin

In this thesis, we use the linear matching approach because it is widely used in other firewalls (e.g., netfilter). This means, our solutions to the ideas are also usable with other firewalls if we rely on the linear approach (some modifications are required, though). In addition, it would not have been possible to implement the timebound and passive ideas with hash-based matching. Therefore, we set the flag at compile time to use linear matching.

4 Related work

In this chapter, we present related work that is about low latency and low jitter packet processing. Our firewall consists of two components that influence the latency and jitter. First, the user space I/O frameworks we use for packet processing in general (DPDK and VPP), and second, the firewall functionality itself.

4.1 Low latency packet forwarding in software

DPDK and VPP, both user space I/O frameworks, lay the foundation of our firewall. The main goal of such user space I/O frameworks is to achieve high throughput. They often process multiple packets in batches, which increases the latency and jitter of packets. However, it is still possible to achieve low latency and jitter using those user space I/O frameworks as we show in this thesis. A few researchers also evaluated the latency of user space I/O frameworks like DPDK.

For example, Gallenmüller *et al.* [34] compared the throughput and latency of multiple user space I/O frameworks, including DPDK. Moreover, they vary the batch sizes to find a tradeoff between latency and throughput. Lastly, they introduce a model to assess the performance of the frameworks based on several measurements.

Stylianopoulos *et al.* [35] go one step further and propose optimizations to the configuration of a Linux-based distribution to reduce latency and jitter of packet forwarding. They evaluate the improvements by measuring latency and jitter using the Linux kernel network stack as well as DPDK. We considered their suggestions in our testbed to get the lowest latency and jitter possible.

Emmerich *et al.* [36] did not evaluate user space I/O frameworks. They propose an improved algorithm for the NAPI in the Linux kernel. Their algorithm reduces the latency without increasing the CPU utilization too much. However, they did not consider jitter. Moreover, they cannot overcome the other bottlenecks in the kernel that limit throughput and latency (see Section 3.2.3).

4.2 Real-time packet processing on firewalls

A large part of the research on firewalls focuses on evaluating and improving the throughput. Only a few researchers consider latency and jitter.

For example, Cereia *et al.* [37] measured the latency and jitter of an industrial firewall in three scenarios. Decommissioned mode (only forwarding), ACL checks enabled, and Deep Packet Inspection (DPI) enabled. The DPI measurement is not relevant in our case because it filters Modbus TCP packets. The authors only configured up to 31 rules but still measured a high latency with their utilized firewall in the range of hundreds of microseconds. They only measure the latency at a low data rate (one TCP request every 100 ms).

Cheminod *et al.* [38] assumed a scenario where the industrial firewall is placed between an office network and an industrial control network. They measured how much office traffic they could transmit over the industrial firewall before the time constraints for the industrial traffic could not be met anymore. They used DPI to check their industrial traffic. DPI increases the time to check the packets and, therefore, reduces the performance. On the other hand, they checked the office traffic only using ACL rules.

In another paper [39], the same authors extended the measurements carried out in the paper above. They changed the testbed to a simple setup with a traffic generator, a firewall, and a receiver. Using this testbed, they measured the latency and the jitter with and without DPI.

Up to this point, the papers we mentioned in this section did not take the concept of zones and conduits according to IEC 62443 into account. Zvabva *et al.* [40], on the other hand, had the concept of zones and conduits in mind. They measured the latency and jitter in a network with multiple firewalls in a row. However, they did only measure up to 18 ACL rules.

Wüsteney *et al.* [1] also considered the concept of zones and conduits. Moreover, they analyzed the problems caused by the high latencies and jitter of industrial firewalls when using TSN. Based on their analysis, they present approaches to deal with the latency and jitter caused by firewalls. They conclude that current software firewalls are difficult to deploy in a TSN environment. Hence, we try to reduce the latency and jitter of software firewalls in this thesis.

Pesé *et al.* [41] developed a proof-of-concept firewall for automotive purposes. They had low latency and low jitter in their mind during development. However, their firewall is not a pure software firewall because they rely on an FPGA as additional hardware.

5 Design

In this chapter, we first discuss why we chose VPP as firewall. After that, we discuss how we design the ideas we presented in Section 2.4 to reduce latency and jitter as much as possible. Moreover, we present the testbed we used to measure the latency and to gather other information about packet processing. We use the testbed to analyze the effect of our implementation of the ideas on latency and jitter.

5.1 Firewall selection

To implement the ideas we presented in Section 2.4, we first need to choose a suitable firewall. We defined multiple requirements that the firewall must satisfy:

1. The firewall must be a software firewall without hardware dependencies (except a NIC that is compatible with DPDK)
2. The firewall must provide a stateful packet filter
3. The firewall must provide good overall performance (high throughput and, more importantly, low latency) because improving an inefficient firewall is not useful if better ones are available
4. We must be able to implement the ideas into the firewall (source code must be available, no technical restrictions)

We examined four software firewalls/filtering solutions that already satisfy the first, second, and fourth requirement:

- iptables/nftables (see Section 3.2.4.2)
- eBPF (see Section 3.2.5)
- bpf-iptables (see Section 3.2.6)
- VPP (see Section 3.3.3)

In the following, we evaluate their suitability in terms of performance and ease of implementation.

5.1.1 iptables/nftables

iptables is a well-known firewall shipped with the Linux kernel (in the following, we use the term iptables to address both, iptables and nftables). It is widespread and well maintained.

However, being integrated into the kernel also means that we need to modify the kernel. This makes our modifications harder to maintain and to deploy. We would have to keep up with kernel development to be able to use the latest kernel version including our modifications. To deploy our changes, we would have to compile and replace the kernel every time. In addition, the performance of iptables is not optimal because it brings along all the bottlenecks of the Linux network stack. For example, bpf-iptables achieves higher throughput than iptables as we show in Section 5.1.3.

5.1.2 eBPF

Just like iptables, eBPF is part of the Linux kernel and, therefore, well maintained. eBPF addresses the problem we mentioned in the last section about iptables: eBPF does not require recompiling the kernel to apply changes. Instead, we can use eBPF programs to implement the firewall functionality and load them into the kernel during runtime. Thanks to the XDP hook, eBPF is able to provide better performance than iptables.

The disadvantage of eBPF is that it does not provide any firewall features by default. We would have to implement packet filtering ourselves which would take a lot of time. As a solution, we discovered bpf-iptables that we discuss in the next section.

5.1.3 bpf-iptables

bpf-iptables implements a firewall based on eBPF programs. Miano *et al.* [17] achieved higher throughput with bpf-iptables compared to iptables and nftables. They measured the throughput with 50 rules in the `FORWARD` chain and 64-bytes-packets on a single CPU core[1]. With bpf-iptables, they measured a throughput of about 0.9 Mpps. In comparison, they only measured a throughput of about 0.5 Mpps with iptables and about 0.2 Mpps with nftables.

Unfortunately, Miano *et al.* did not measure the latency and jitter. In general, we assume that a higher throughput implies a lower latency and jitter. If a firewall has a higher throughput, the packets take less time for processing, implying a lower latency. Jitter is, besides the ACL check, often caused by overload situations which occur more often on firewalls with lower throughput.

While bpf-iptables is a better choice than iptables performance-wise, it has the disadvantage that it does not support all features of iptables. Moreover, the future development of bpf-iptables is unclear. Since the publication of the corresponding paper [17], bpf-iptables was not further developed (see the link to its source code at [42]).

[1]Note that their packets match the rules uniformly over whole rule set. In our measurements, only the last rule matches.

5.1.4 FD.io VPP

In contrast to the firewall solutions above, VPP is a user space application. As a result, it is easy to implement new features because no kernel modifications are necessary. VPP is actively developed, mainly by Cisco. Therefore, it is a future-proof choice. The performance of VPP is better than the performance of the other firewalls above. For example, VPP achieves a median throughput of about 7 Mpps with 50 ACL rules on one CPU core (with hyper-threading enabled) and 64-bytes-packets [43]. This is a much higher throughput than the one of bpf-iptables[2].

Because VPP offers the highest throughput and presumably the lowest latency, we chose to modify VPP in this thesis. In addition, its flexible architecture makes the implementation of our ideas easier. The advantages of VPP outweigh the fact that VPP is less widespread than iptables.

5.2 Timebound

We already explained the timebound idea in Section 2.4 and split it into three components: configurable time limit, adaptive time limit, and analyze later. In this section, we describe the design of these three components and the interaction between them. Additionally, we discuss alternative design approaches. Details about the implementation with VPP are covered in Chapter 6.

5.2.1 Configurable time limit

With the help of a configurable time limit, we can limit the duration of ACL processing to a configurable time period. The configuration and functionality is similar to ACLs, as we can set the time limit per flow. An administrator can configure rules that are placed in a list. We call this list *priority list*. Table 5.1 shows the structure of the priority list with example entries. The matching of packets works the same as in the ACL. The priority list contains fields for the 5-tuple and TCP flags. While a priority list rule does not have an action field like an ACL rule, it has a time limit field to store the time limit that should be applied to the matching packets. The priority list can be configured via API to be able to add and delete rules. Configuration using the CLI is not possible in our implementation, but CLI commands for that can be added in future. Our implementation matches the 5-tuple and TCP flags only. However, information from the layer 2 header and the VLAN tag can be used for matching. This is for example useful when used in conjunction with TSN because it uses the priority code point field in the VLAN tag. To demonstrate the functionality of the timebound idea, our implementation is sufficient.

The priority list is processed similar to the ACL. This means, it is traversed in a linear way until a matching rule is found. If a rule matches, the time limit stored by the rule is used to limit the duration of the following ACL processing. If no rule matches, the ACL processing time is not limited. However, to ensure that the

[2]We tried to match the CPU models of bpf-iptables and VPP measurements as close as possible. Even if we dvide the throughput of the VPP measurements by two to compensate the use of hyper-threading, VPP has still a higher throughput.

Src IP	Src port	Dst IP	Dst Port	Protocol	TCP flags	Time limit
10.0.1.0/24	-	10.0.2.0/24	8080-8085	UDP	-	0.6 µs
10.0.2.0/24	-	10.0.3.0/24	8098	UDP	-	1.0 µs
10.0.3.0/24	-	10.0.4.0/24	8099	TCP	-	1.5 µs
10.0.4.0/24	8099	10.0.3.0/24		TCP	-	1.5 µs
-	-	-	-	-	-	2.0 µs

Table 5.1: Example of a priority list with configured rules (the last rule is a catch-all rule)

ACL processing duration is limited for all packets, a catch-all rule can be added to the priority list. If the time limit exceeds during ACL processing, the packet is forwarded. In addition, the packet information required to continue ACL processing is forwarded to analyze later to complete the ACL check in the background.

5.2.2 Adaptive time limit

While the configurable time limit cannot prevent firewall overload in dynamic load scenarios, the adaptive time limit adapts the time limit dynamically based on the firewall load. This prenvents latency and jitter from increasing too much. The adaptive time limit can be used in combination with configurable time limit if there are packets with very strict time constraints that cannot be met from only using an adaptive time limit. The adaptive time limit does not meet the time constraint of a packet if its time constraint is lower than the highest adaptive time limit.

Adaptive time limit means that the time limit for ACL processing is chosen dynamically based on the load. This prevents the ingress queue from filling up with too many packets which would increase latency. For example, if the firewall is under low load, the time limit can be set to 2 µs and the queue will not fill with packets. But if the load is high, for example, 500 Mbit/s with a packet size of 64 bytes, 2 µs are too long because the packet interarrival time is lower (see Section 2.3 for an explanation of the interarrival time). In that case, the time limit must be reduced to a value so that the current packet has finished processing before (or just as) the next packet arrives. This ensures that the next packet must not wait in the ingress queue. Given the example data rate and packet size above and assuming the rate and the packet size do not change, the next packet arrives 1.34 µs after the last packet. As a result, the ACL processing time must be limited to a duration below 1.34 µs so that other forwarding tasks can also be completed within this time.

5.2.2.1 Measuring the firewall load

To make the time limit adaptive, the firewall must be able to adjust the time limit based on the current firewall load. There are several ways to measure the current firewall load. Some of them are easier to implement and more suitable while others are impossible to implement, depending on the utilized firewall.

CPU The most trivial way is to measure the CPU utilization. The CPU utilization directly relates to the firewall load, as the CPU needs to do more processing if the packet rate is higher. At a CPU utilization of 100 %, the firewall cannot keep up anymore and more packets arrive than it can process. However, in our case it is impossible to retrieve useful data from measuring the CPU utilization. The reason for this is the way how VPP (with the help of DPDK) fetches packets from the queue. VPP polls the ingress queues continuously for new packets if has no packets to process. As a result, we always read a CPU utilization of 100 %.

Packet rate Another way is to measure the rate of packets flowing through the firewall. The packet rate can be measured by counting the number of packets arriving at the firewall within a given time period. It can also be measured by determining the time interval between arriving packets. The disadvantage of this method is that there is no static relationship between the packet rate and the firewall load. A firewall with a better CPU could handle a certain packet rate better than a firewall with a bad CPU and would therefore not overload. This means that the relationship must be determined individually for each CPU model. Furthermore, the load on the CPU depends on the packet. For example, one packet requires checking 10 ACL rules and another packet requires checking 100 ACL rules. Thus, we cannot exactly determine the firewall load even with a CPU-dependent mapping between packet rate and firewall load.

Ingress queue A better way to determine the firewall load is to measure the ingress queue occupancy. In other words, we count the number of packets in the ingress queue. The queue occupancy is a good measure of the firewall load because the number of packets in the ingress queue increases if the firewall is overloaded. There is a limitation when using VPP, though. If multiple packets are waiting in the queue, VPP fetches up to 256 of them at once from the queue (see Section 3.3.3.3). This means, if we measure the queue occupancy after VPP fetched the packets, the queue is empty or contains far less packets than expected. Instead, VPP already stored the packets in a vector. Therefore, we must measure the queue occupancy before VPP fetches the packets from the queue.

Vector size Alternatively, we can measure the vector size after VPP fetched the packets from the ingress queue. The vector size is equal to the number of packets that VPP fetched from the ingress queue. Figure 5.1 shows why the vector size is a good indicator of the firewall load. If the data rate is low, there is much time available for ACL processing until the next packet arrives. As a result, the queue stays empty and the vectors only contain one packet. But at higher data rates, where less time is available until the next packet arrives, ACL processing can take longer than time is available. In that case, packets are already waiting in the queue when packet processing is finished. The vector size increases because all packets from the queue are fetched at once and stored in a vector. VPP stores packets in vectors because batch-processing multiple packets at once brings performance improvements (as explained in Section 3.3.3.5). Because of the performance gain, the vector size settles at a new value and the queue contains roughly the same number of packets every time when the packets in the vector have finished processing (with some outliers because of fluctuations in processing duration). If the firewall is overloaded for a longer time, the vector size settles at 256 packets (the maximum possible vector

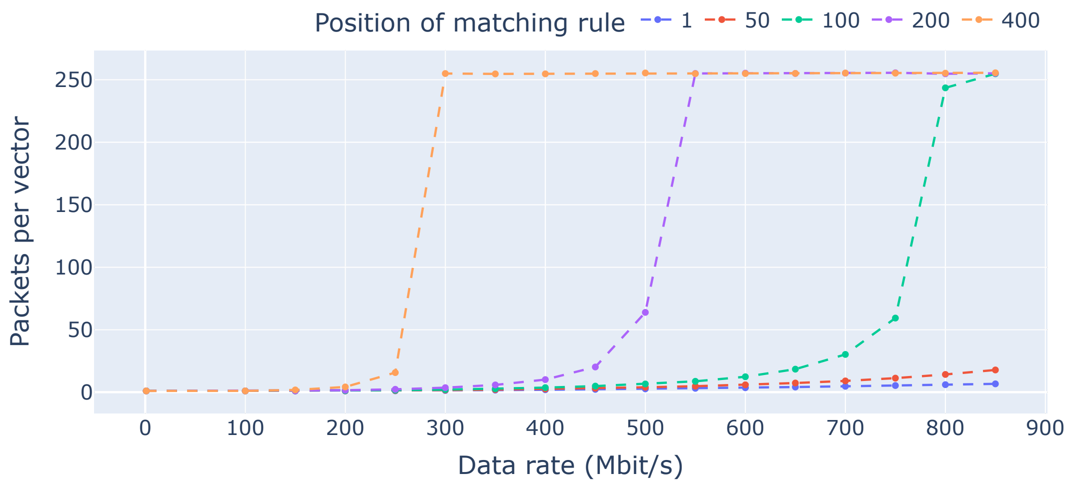


Figure 5.1: Mean vector size depending on the data rate and the position of the matching ACL rule (packet size: 64 B)

size). A vector size of 256 signifies that there are most likely still packets in the queue but VPP was not able to fetch all of them because of the maximum vector size. If the vector size is at its maximum value for a longer time, the queue is full and packet loss occurs.

In practice, measuring the vector size is similar to measuring the queue occupancy but yet contrary. The similarity is that they both measure the number of packets. But the difference is that the queue occupancy equals the number of packets in the queue while the vector size equals the number of packets fetched from the queue. Thus, we try to keep the queue occupancy equal to zero while we try to keep the vector size equal to one.

Up to this point, we assumed that the firewall has only one port. In this case, we must only measure the occupancy of one ingress queue. But this changes if the firewall has multiple ports on which packets can arrive (or if one port has multiple ingress queues). Then, we cannot measure the ingress queue occupancy of an arbitrary port but we must measure it for all ingress queues of all ports. After we measured the queue occupancy of all queues, we choose the largest queue occupancy of them. We choose the queue with the largest occupancy because only this one relates to the firewall load. The other queues have a smaller occupancy because they have a lower rate of arriving packets. The largest occupancy value can then be used to determine the firewall load.

If we measure the firewall load using the vector size and the firewall has multiple ports, we use a similar approach. In this case, VPP creates one vector per ingress queue. VPP can merge the individual vectors later on their way through the packet processing graph. But we cannot be sure that VPP merges them all into one single vector because the packets can traverse different nodes. Therefore, we measure the vector size of each vector directly after VPP fetched the packets from the corresponding ingress queue. We then choose the largest vector size and use it to determine the firewall load.

5.2.2.2 Deriving the time limit from the vector size

In the last section, we explained multiple ways to measure the firewall load. For our implementation, we chose the vector size because it is the most native way in VPP to measure the firewall load. Now, we explain how we determine the time limit based on the measured firewall load, that is, the vector size.

We use a PID controller for this because it is simple to implement and does not require much processing. PID stands for the three components that PID consists of: proportional, integral, and derivative. The input of the PID controller is an error value. It is the difference between the setpoint and the actual value that was, for example, measured. Each of the three components calculates an individual correction value out of the error value where the proportional component considers the current error value itself. The integral component considers the sum of the error over time and the derivative error considers the change of the error over time. The sum of all three correction values is the output of the controller.

In this thesis, we only implemented a P controller to show that it is possible to prevent firewall overload by adapting the time limit to the current firewall load. In future it is possible to improve the controller by tweaking the constants we show in the following and by testing a combination of P, I, and D controllers.

First, we must define the input of the controller. We want to control the firewall load, or more precisely, the vector size. But the controller expects an error value as input, so we cannot use the measured vector size as input. To calculate the error, we must define our setpoint (i.e., the vector size) first. Our goal is to keep the vector size small, ideally one. Hence, the setpoint of the vector size is one, and every value larger than one is an error. In Equation (5.2), we calculate the error e at a given time t by subtracting the setpoint of the vector size $V = 1$ from the vector size s_v at that time.

Next, we define the output of the controller. When the vector size increases, we want to reduce the time limit for ACL processing. Therefore, the output of the controller must change the time limit. We calculate the output of the controller P_{out} by multiplying the error e by the gain constant K_p as shown in Equation (5.3). We explain below why we chose the value $200{\cdot}10^{-9}$ for our gain constant.

The output of the controller is not the final time limit, it is only a correction value. We calculate the final adaptive time limit l_a by subtracting the controller output P_{out} from the constant $2 \cdot 10^{-6}$ as shown in Equation (5.4). We need this additional step because we want our upper bound of the time limit to be 2 µs ($= 2 \cdot 10^{-6}$ s). This means, the packet should not spend more than 2 µs with ACL processing. If we would not apply an upper bound, the ingress queue could fill up with an indefinite number of packets leading to an increased latency. The controller could then only counteract the increased latency at the next ACL check which is too late. P_{out} is just the amount of time in seconds that is subtracted from the upper bound. To prevent that the time limit gets negative or too low, we also set a lower bound, in this case, 0.2 µs ($= 200 \cdot 10^{-9}$ s). This ensures that at least a few rules are checked before the firewall forwards the packet. But the lower bound also limits up to which point firewall overload can be prevented.

In summary, each additional packet in the vector, reduces the time limit by 0.2 µs until the time limit reaches the lower bound of 0.2 µs. The reduction in steps of 0.2 µs comes from our gain K_p that we chose to calculate P_{out}. For example, if the vector size is 1, the time limit is 2 µs. And if the vector size is 2, the time limit is 1.8 µs.

$$V = 1, K_p = 200 \cdot 10^{-9} \tag{5.1}$$

$$\begin{aligned} e(t) &= s_v(t) - V \\ e(t) &= s_v(t) - 1 \end{aligned} \tag{5.2}$$

$$\begin{aligned} P_{out}(t) &= K_p \cdot e(t) \\ P_{out}(t) &= 200 \cdot 10^{-9} \cdot e(t) \end{aligned} \tag{5.3}$$

$$l_a(t) = \max\left(2 \cdot 10^{-6} - P_{out}(t), 200 \cdot 10^{-9}\right) \tag{5.4}$$

5.2.2.3 Combination of adaptive and configurable time limit

If we combine the adaptive time limit and the configurable time limit, we end up with two different time limits. In this case, we chose the lower time limit for two reasons. First, the configurable time limit must be the upper bound of ACL processing duration. No packet should take longer, even if the firewall load is low. Second, firewall overload should always be prevented, even if the configurable time limit is higher.

Choosing the lower time limit is achieved using the min function as shown in Equation (5.5) where l_a is the value of the adaptive time limit and l_c the value of the configurable time limit.

$$l(t) = \min\left(l_a\left(t\right), l_c\left(t\right)\right) \tag{5.5}$$

5.2.3 Analyze later

If ACL processing cannot be finished within the time limit, it is interrupted and the packet is forwarded to the next hop. Analyze later continues ACL processing of the packet in the background without interrupting packet processing. If the result of the background check yields that the packet should have been dropped, the result should be logged or reported. For example, we can log the result using syslog or report it to a monitoring service. Analyze later must meet the following requirements:

- It must not interrupt packet processing which would increase latency and jitter

- The ACL processing node must be able to notify analyze later about new packets that need to be checked
- ACL checks should be continued where they were interrupted instead of starting all over again (to save processing power)

5.2.3.1 Separation of packet processing and analyze later

To not interrupt packet processing, analyze later runs on a dedicated CPU core that is not responsible for packet processing. In case of VPP, this is the main thread that is also responsible for CLI, API, and other organizational tasks. The worker thread(s) can thus fully focus on packet and time-limited ACL processing.

Analyze later runs as individual processing node on the main thread and has to share its processing resources with other nodes whose functions were just mentioned. Because of this, analyze later can only run for a limited time until it must interrupt its work to give processing resources to other nodes (see Section 3.3.3.6 for an explanation of the reason for this). To prevent that analyze later blocks other nodes for too long because a large number of ACL rules have to be processed, analyze later also uses the timebound idea to pause ACL processing after some time. Analyze later then suspends and gives other nodes a chance to run. When analyze later can run again, it continues ACL processing where it has stopped.

Alternatively, analyze later can run on the same CPU core as packet processing. In that case, analyze later runs between packet processing, meaning, after the last packet was forwarded and before the next packet arrives. Running between the processing of packets (i.e., in an idle state) prevents that latency increases because of analyze later. But running analyze later just as long until a packet arrives is not easy to implement because we would have to detect the arrival of a packet in parallel (we discuss this later in the design of the priority idea in Section 5.4.1.1). Furthermore, running analyze later between packet processing is a contradiction because analyze later is especially needed under high load scenarios. But under high load, analyze later would not have enough time for background ACL processing due to the high rate of arriving packets. Hence, we recommend running analyze later on a different CPU core.

Instead of running analyze later on the same firewall, it can also be offloaded to another device like a server. This has the advantage that more processing power is available for background checks since we can choose a device that has a better CPU or is just under less load because it must not take care of packet processing. Whether packet processing benefits from offloading due to more available processing power on the firewall depends on the available processing power compared to the firewall. The disadvantage of offloading analyze later is that the delay increases until the results are available. A delayed analyze later result can have an impact on security, as wrong decisions are detected later. Fast analyze later devices can eventually compensate the delay caused by the transmission to the external device by faster ACL processing. But this largely depends on the number of rules to process and the delay to exchange data between the two devices. Currently, there is no protocol to exchange relevant data, but this is out of scope of this thesis. Additionally, there is the question how required information (the information about the packets to check) is exchanged between the devices (e.g., the utilized protocol). Required information encompasses the information about packets to check, the ACLs and to which interfaces the ACLs are assigned.

As we can see, offloading analyze later to another device requires further design considerations but is an approach worth considering. In this thesis, we implemented the approach where analyze later runs on the main thread, separate from packet processing.

5.2.3.2 Communication between ACL processing node and analyze later

Notifying analyze later about new packets that need to be checked is not trivial because it must be efficient to not slow down packet processing. We chose a buffer into which the ACL processing node inserts the information about the packets that need to be checked. Analyze later then reads the packet information from the buffer in the order it was inserted by the ACL processing node. Every time, analyze later finishes the background processing of a packet, it deletes the corresponding entry from the buffer and reads the information about the next packet from the buffer.

Due to hardware limitations that we discuss in Section 6.2.3.1, we define a maximum size for the buffer. If background processing is too slow, or in other words, if the ACL processing node inserts more packets into the buffer than analyze later can delete after it finished the background check of a packet, the buffer becomes full. There are two possibilities how we can handle a packet that cannot be inserted into the buffer: we drop the packet or we forward the packet without an additional background check. Since forwarding without check is insecure because it could be exploited by an attacker (Section 7.9.1.2), we recommend dropping the packet.

Another important point is the packet information that must be included in a buffer entry so that analyze later can check the packet. We only keep the necessary information and not the whole packet because copying the whole packet would require too much processing power (and memory) and therefore would increase the latency. Analyze later needs the following information:

- The header fields of a packet that correspond to the ACL fields (i.e., the 5-tuple and the TCP flags)
- The ACLs that need to be checked
- The rule and the corresponding ACL that has to be checked next (the previous ones were already checked)

The header fields that correspond to the ACL fields are the actual information that is extracted from the packet required for matching. The ACLs that need to be checked are needed because analyze later does not know the ingoing or outgoing interface of the packet and therefore cannot get the corresponding ACLs. To be able to continue where the ACL processing was interrupted, the rule and the corresponding ACL that has to be processed next must also be saved. Note that we do not copy the whole information. To save memory, we store pointers to the corresponding data.

5.2.3.3 Summary

Figure 5.2 shows the interaction between all components of analyze later. First, there is the worker thread including the ACL processing node. Every time, one or more packets are fetched from the queue by the RX node, they traverse the graph towards the TX node. They eventually arrive at the ACL node with enabled timebound ACL checks. If the time limit for the ACL checks of a packet exceeds, the packet information is inserted into the analyze later buffer. Packet processing can then continue in the other nodes until they arrive at the TX node that sends the packets to their destination.

Analyze later runs in parallel on the main thread and has to share its processing resources with other nodes that are, for example, responsible for the CLI. Every time the analyze later node is active, it runs its ACL processing loop. This means, it fetches the next packet information from the analyze later buffer to continue ACL checks where they were stopped by the ACL processing node. When analyze later finishes the ACL checks, it deletes the packet information from the buffer and takes action according to the matching result.

The processing time of the analyze later node is limited and therefore, it also assigns a time limit to its own ACL checks. If the time limit exceeds, analyze later suspends and gives processing resources back to other nodes. If analyze later is the active node again, it continues where it stopped.

5.3 Passive

With the passive idea, which we first introduced in Section 2.4.2, the firewall forwards packets without previous ACL processing. Instead, the firewall performs the whole ACL processing in the background using analyze later (see Section 5.2.3 for analyze later design). Since an administrator should be able to configure which packets the firewall forwards without ACL check, we use the priority list from the configurable time limit (see Section 5.2.1 for configurable time limit design). For this, the administrator configures a time limit of zero in the priority list. This causes the firewall to skip ACL checks and to pass the packet information directly to analyze later. More features are not required, we just use the features we already need for the timebound idea.

5.4 Priority

Our goal with the priority idea is to forward high-priority packets as fast as possible. Non-priority packets normally delay a high-priority packet (see Section 2.4.3). We came up with two approaches to overcome this. The first approach is a single-core approach where high-priority and non-priority packets are processed on the same CPU core. The second approach is a multi-core approach where high-priority packets and non-priority packets are processed on different CPU cores. In the following, we discuss both approaches, but due to the disadvantages of the single-core approach, we recommend and we also implemented the multi-core approach.

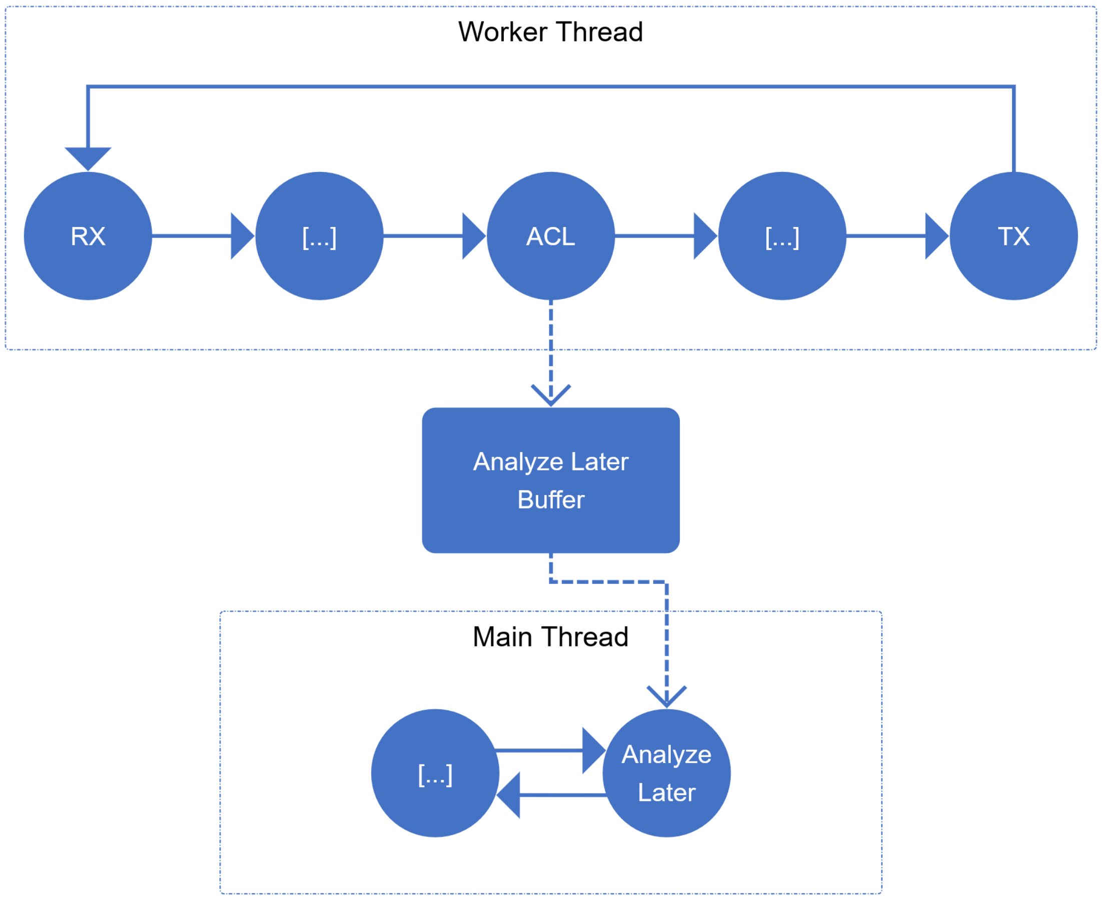


Figure 5.2: Architecture of analyze later

5.4.1 Single-core approach

If the firewall processes high-priority and non-priority packets on the same core, it must pause processing of non-priority packets when a high-priority packet arrives. This way, the firewall ensures that it can process and forward the high-priority packet as fast as possible. Therefore, the firewall must detect that a high-priority packet arrived and is waiting in the ingress queue for processing. We call this action *queue lookup*.

5.4.1.1 Detecting the arrival of high-priority packets

We have two possibilities to implement a queue lookup using a single CPU core:

- Pause packet processing and perform the queue lookup in regular intervals
- Delegate the queue lookup to the NIC and get notified by an interrupt

A trivial solution to pause packet processing for queue lookups is to wrap the queue lookup in a function and insert function calls all around the code in regular intervals. Listing 5.1 illustrates this solution by calling the `queue_lookup` function that is responsible for queue lookup in that example. However, we find that this is a dirty and unreliable solution since it is unclear in which time intervals these checks are performed and also the same function call is added everywhere in the code. In a large code base like the one from VPP with tens of thousands lines of code, this is not only dirty, copying and pasting the same function call over and over is unacceptable and tedious. Performing the queue lookups only during the time intensive ACL processing would be an acceptable solution because only one function call would be needed. The single function call would be placed in the loop that is responsible for ACL checks as illustrated in Listing 5.2. A drawback of both solutions is the overhead caused by the queue lookup because it slows down packet processing.

```
// Packet processing code

queue_lookup();

// Packet processing code

queue_lookup();

// Packet processing code
```

Listing 5.1: Very basic approach to pause packet processing and perform a queue lookup: insert function calls to queue lookup into the code in regular intervals

A much cleaner solution than inserting function calls everywhere is using VPP's node/graph architecture (see Section 3.3.3.3). We can place the queue lookup logic in a separate node that is called by VPP in regular time intervals. However, this solution introduces more overhead because VPP must suspend the currently active packet processing node before the queue lookup node can run (switching the active node requires additional processing). Another problem is VPP's scheduling architecture: VPP cannot call another node while a node is active because of the cooperative multitasking design. As a result, we cannot completely control the intervals in which VPP runs the queue lookup node. For example, VPP cannot call the queue lookup node while the ACL processing node performs ACL checks. But ACL checks can take a long time and thus, we would not detect arriving high-priority packets. Changing this would require substantial changes to the scheduling code of VPP (if there is even a way to implemnent this efficiently).

```
for (int i = 0; i < len(acl_rules); i++) {
  struct acl_rule rule = acl_rules[i];

  // Here, each rule is compared with the
  // header fields of the packet
  // (e.g., source IP or destination IP)

  queue_lookup();
}
```

Listing 5.2: Pause ACL checks and perform a queue lookup: only a single function call to queue lookup must be inserted

In addition to the overhead introduced by the solutions to call the queue lookup and the complexity to implement them, even the queue lookup itself is costly in terms of performance. To perform the queue lookup, the firewall must fetch all packets from the queue and then check them for high-priority packets. This means, the firewall must also fetch and check non-priority packets which increases the overall overhead and increases the delay for the high-priority packets.

A solution to overcome the overhead caused by the mixup of high-priority and non-priority packets in the queue is to offload packet classification to the NIC. This means, we configure two ingress queues (non-priority queue and high-priority queue) on the NIC and the NIC decides, based on prior configuration, in which queue a packet must be placed. This is an adequate solution that we discuss in detail in the multi-core approach (Section 5.4.2). However, the firewall must still poll the high-priority queue periodically for which it still must pause packet processing.

We can only avoid polling the high-priority queue by using interrupts. The NIC can trigger an interrupt so that VPP is notified about a new packet in the high-priority queue. But interrupts are where we started: we chose a user space firewall to bypass the inefficient and interrupt-based kernel space networking. The interrupts have to travel through the kernel to then notify VPP about the interrupt which would increase the latency until the notification about a new packet arrives in VPP. Since we did not investigate further on this solution, it is unclear what latency would be introduced exactly. But there was an approach by Intel to reduce interrupt latency in order to implement an interrupt-mode driver into DPDK in addition to the poll-mode driver. But except for one presentation [44] there is no more information about it and it seems like development did not continue.

5.4.1.2 High-priority packet processing

Besides the detection of the arrival of a high-priority packet, there is another problem: how can the firewall interrupt processing of a non-priority packet in favor of a high-priority packet? This is a similar problem as interrupting packet processing to detect the arrival of a high-priority packet above.

With VPP, the best solution would be to insert the high-priority packet as a vector into the pending vectors array and suspend the currently running node. Then, VPP switches to the initial node responsible for processing the high-priority packet. But as already written, VPP cannot interrupt the execution of a node in the middle of processing and therefore the high-priority packet would have to wait until the current node has finished processing. This can take a long time if, for example, the ACL processing node is the active node. As written above, changing this would require substantial modifications of the VPP scheduling code.

To sum it up, there are many disadvantages and complications with the single-core approach. The queue lookup is inefficient or comlicated to implement and it is complicated to implement fast switching from non-priority packet processing to high-priority packet processing. Consequently, we discuss the multi-core approach in the following.

5.4.2 Multi-core approach

We split the multi-core approach into two parts: the packet classification that is performed on the NIC and the packet processing that is performed by the firewall on multiple CPU cores.

5.4.2.1 Detecting the arrival of high-priority packets

We investigated two approaches to differentiate high-priority packets from non-priority packets if multiple CPU cores are available.

In the first approach, we use one CPU core to poll the ingress queue and one CPU core for packet processing. This allows one CPU core to permanently poll the ingress queue to fetch arriving packets. The polling CPU core first classifies the packets into high-priority and non-priority packets and then forwards them to the packet processing CPU core. The other core can process the packets and, if required, interrupt non-priority processing to process a high-priority packet.

This approach eliminates the need to interrupt packet processing in order to perform a queue lookup. In addition, more processing power is available for polling and classifying packets so a higher rate of packets can be processed without causing overload (as long as the classification does not get too complex).

The remaining problem is the interruption of non-priority packet processing when a high-priority packet arrives. This is the same problem as in the single core approach and cannot be solved easily. As a solution, we came up with another approach that isolates non-priority and high-priority packet processing completely from each other by processing the packets on different CPU cores. We describe this approach in Section 5.4.2.2.

However, up to this point, we assumed we only have a single ingress queue per port. But we cannot implement the above approach with only one ingress queue because it is not possible to efficiently poll the same queue in parallel on two different CPU cores with DPDK. Therefore, high-priority and non-priority packets must arrive on two different ingress queues. There is an elegant solution to this problem that also avoids the overhead of classifying packets on the CPU: offloading packet classification to the NIC. This is possible with DPDK as we described in Section 3.3.2.2. In short, DPDK configures the NIC to classify packets and to place them, based on the classification result, in a different queue. We can make the configuration of the classification rules accessible to the administrator. In our case, we configure the classification rules on the NIC based on the rules in the priority list that we already described in Section 5.2.1. Even though we use the priority list here, we can still use it for the configurable time limit. The advantage of using the priority list for both solutions together is that we can use it to first separate the high-priority packets from non-priority packets and then to additionally apply a time limit to the high-priority packets.

To be able to configure in which queue the NIC should place the high-priority packets, we extend the priority list by an additional field. This field must store multiple *port-queue pairs*. For each port on which we expect high-priority packets to arrive that match the priority list rule, we must configure such a port-queue pair. A port-queue pair tells the NIC that is responsible for a port, in which queue it should place matching packets. If a packet does not match any rule, the NIC places it in the first queue (the NIC usually identifies the queues with numbers). Hence, a port-queue pair should always specify the second queue or higher.

A disadvantage of classification on the NIC is the increased dependency from the NIC model because care must be taken to coose a model that supports the required classification features. However, many NICs support the required features as long as they are supported by DPDK, especially if they are destined for data center or embedded usage.

All in all, offloading packet classification to the NIC is a helpful solution because it reduces the firewall load. The classification result is easy to process because the NIC places the packets in different queues as configured by the administrator.

5.4.2.2 High-priority packet processing

The packet classification on the NIC is ideal to process non-priority and high-priority packets on different CPU cores. VPP polls different queues that belong to the same interface on different worker threads (as long as enough CPU cores are available and multiple worker threads are configured). This is the key of the multi-core approach because as a result, VPP processes high-priority and non-priority packets completely independent of each other on different CPU cores. From the arrival at the NIC to the end of packet processing, there are no more places where packet processing of high-priority and non-priority packets can interfere with each other. For example, the CPU core processing non-priority packets can be completely overloaded without affecting high-priority packet processing. However, it is important to prevent overload on the CPU core processing high-priority packets. But usually, the rate of high-priority packets is very low, in the range of kilobits per second. In case the rate is too high, the high-priority packets can be distributed on multiple CPU cores as long enough cores and queues are available (NICs only support a limited number of queues, e.g., four ingress queues per port in our case).

The only remaining location where a high-priority packet has to wait for non-priority packets in some cases is the transmission of packets. For example, the firewall transmits a 1200 bytes large non-priority packet (this takes 9.6 µs on a 1 Gbit/s link). While transmitting the non-priority packet, the firewall receives a high-priority packet. The firewall processes the high-priority packet and transmits it via the same port. But that port is still in use to transmit the non-priority packet. Consequently, the high-priority packet must wait for the end of transmission of the non-priority packet. But this is not in scope of this thesis and a problem to be solved with, for example, TSN.

We can also extend the priority idea further: we could split an ACL into multiple per-core ACLs. For example, one CPU core only processes high-priority packets but the ACL on that core contains rules matching both, high-priority and non-priority packets. As a result, the ACL check on that core checks all rules even if some rules can never match because the CPU core never processes corresponding packets. Thus, we can split the ACL into a high-priority ACL and a non-priority ACL. This improves the efficiency of ACL checks because less rules must be checked.

5.4.2.3 Summary

Figure 5.3 shows a summary of the final desgin of the priority idea with VPP. We assume that the NIC has two queues and matches arriving packets to place high-priority packets in queue 2 and the remaining packets in queue 1 (the default queue). To configure the rules on the NIC, VPP uses the rules from the priority list (not shown on the figure). Besides the main thread, VPP runs two worker threads on different CPU cores. Worker thread 1 is responsible for the non-priority packets while worker thread 2 is responsible for the high-priority packets. They permanently poll the corresponding queue, fetch and process the arriving packets. Since both worker threads run in parallel, they do not block each other and thus, the high-priority packets do not have to wait.

If enough queues and CPU cores are available, this approach can be scaled by increasing the queue and worker thread count as desired to process more packets. The number of default queues can also be increased if RSS is used. RSS distributes non-priority packets on configurable queues without the need to add priority list rules (RSS can be configured in the VPP configuration file).

5.5 Testbed

This section is about the design of the testbed that we used to measure the performance (e.g., latency and packet loss) of the unmodified and the modified firewall. With the performance measurements, we can evaluate whether our modifications yield the expected results.

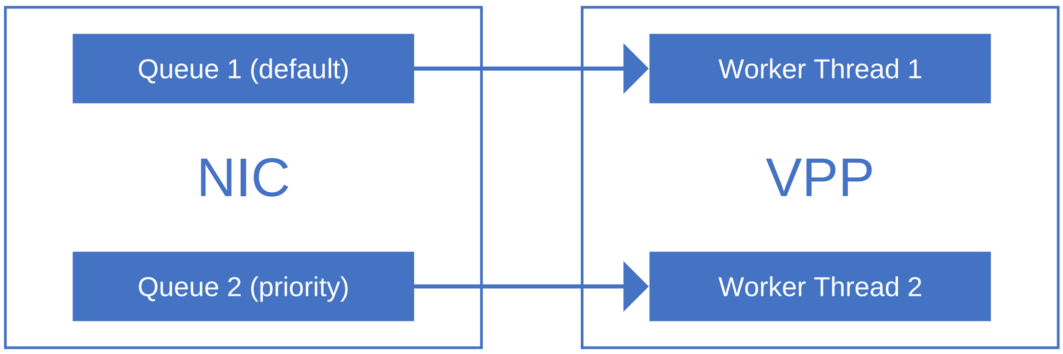


Figure 5.3: Overview of the priority design with VPP

The testbed consists of multiple components and devices. The network diagram in Figure 5.4 gives an overview of the devices. On the left-hand side is the packet generator that is responsible for packet generation and latency calculation. It is connected with two wires to the timestamping switch that writes timestamps into the packets flowing through it. The timestamping switch is in turn connected with two wires to the Device Under Test (DUT), the device running our (modified) VPP firewall. All testbed devices are also connected to a management switch that we can use to connect another device to configure the testbed devices.

The arrows on the network diagram indicate the flow of packets during a measurement. They start at the packet generator and flow through the timestamping switch to the DUT. After packet processing by the DUT, the packets flow through the timestamping switch on different wires back to the packet generator.

Not shown on the network diagram are additional components we used for the measurements. The first important component is called *internal performance measurements*. We implemented this component into VPP. It enables us to measure the time required exclusively for ACL matching of a packet without additional packet processing. In addition, internal performance measurements also give us other information about packet processing, such as the number of rules that were checked by the ACL. The other component is the automation of the measurements with scripts so that various load scenarios can be tested without manual intervention.

In the following, we discuss the packet generator, the timestamping switch, the DUT, and the mentioned components in detail.

5.5.1 Device Under Test

The DUT is our modified software firewall and can be separated into a hardware part and a software part. For the hardware part, we use an EAGLE40 industrial firewall by Hirschmann Automation and Control GmbH. It runs on an Intel Atom x7-E3950 CPU with 4 cores running at a base clock of 1.6 GHz and has 8 GB of RAM. The Gigabit Ethernet network interfaces use Intel I210 Ethernet controllers that are supported by DPDK.

In its original version, the EAGLE40 comes with its own software (HiSecOS). But

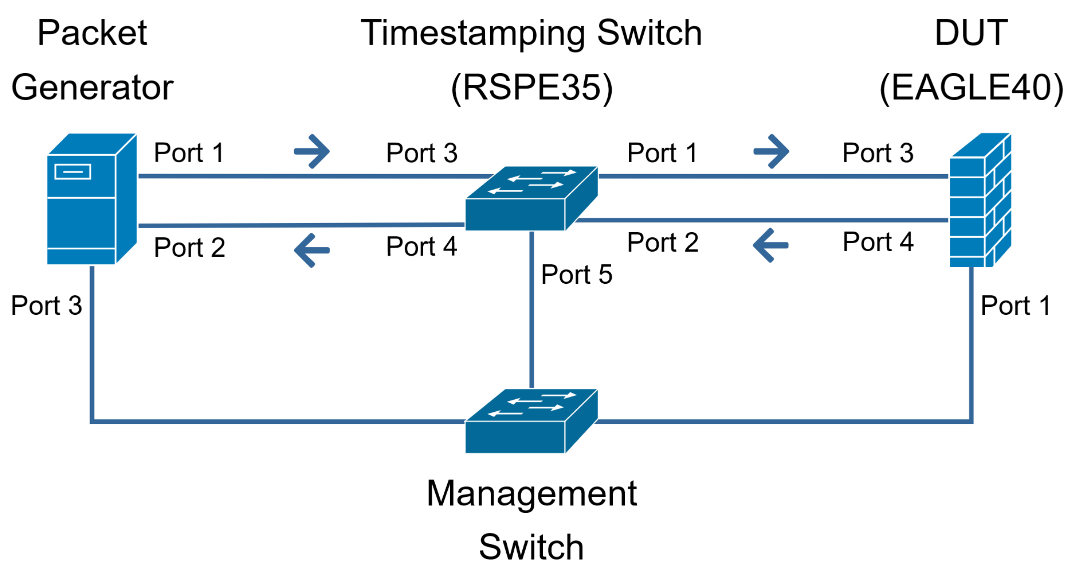


Figure 5.4: Network diagram of the testbed (the arrows indicate the flow direction of the packets during measurement)

to install VPP, we replaced the stock software with Ubuntu 20.04 LTS. We replaced the kernel of the Ubuntu LTS with an Ubuntu kernel compiled by us with version 5.4.151 (we explain why we compiled the kernel in Section 6.6.2). On top of Ubuntu, we install and configure (our modified) VPP.

5.5.2 Timestamping switch

The timestamping switch is located between packet generator and DUT. It measures the latency of the packets (i.e., how much time it takes for them to flow through the firewall). This is done by writing timestamps into a packet before and after it traverses the firewall, respectively.

5.5.2.1 Hardware

For timestamping, we use an RSPE35 industrial switch by Hirschmann Automation and Control GmbH. Its ports support a line speed of 1 Gbit/s. By default, the switch does not support timestamping. Hirschmann provided us a slightly modified stock software for the switch that supports timestamping.

5.5.2.2 Timestamping

The timestamps written to the packets have an accuracy in the range of nanoseconds and have a size of 4 bytes, respectively. The maximum period that can be stored with nanosecond precision is $2^{4 \cdot 8} = 4\,294\,967\,296\,\text{ns}$, that is, just about four seconds. The timestamps omit the full seconds and only contain the decimal portion of the seconds. In other words, the timestamps are always in the range of 0 ns to 999 999 999 ns.

This can lead to overflow: The switch writes the first timestamp to a packet just before the beginning of a new second (e.g., 0.999 998 s). On its way back, the switch writes the second timestamp that is part of the following second (e.g., 0.000 015 s). Normally, we calculate the latency by subtracting the *ingress timestamp* (packet towards DUT) from the *egress timestamp* (packet coming from DUT). But this is not possible here, the latency would be negative. As a solution, we use Equation (5.6) to calculate the latency t if the ingress timestamp t_i is larger than the egress timestamp t_e.

$$t = t_e + 1\,000\,000\,000 - t_i \tag{5.6}$$

Note that the measured latency of the packets does not only include the latency caused by the firewall itself. It also includes the latency caused by the timestamping mechanism. We call the latency caused by the timestamping mechanism *timestamping latency*. It consists of the latency caused by the transmission of the packets from the timestamping switch to the firewall and back. The timestamping mechanism itself that writes the timestamps to the packet does not cause any additional latency because it is done in hardware. In our results, we do not subtract the timestamping latency from the total latency because it is constant and small compared to the total latency. Therefore, it can be neglected in our results.

Nevertheless, we can calculate the additional timestamping latency t_{ts} with Equation (5.7). The equation calculates the duration to transmit a frame of size s_p to the firewall and back. r_l is the line rate in bits per second which is $1 \cdot 10^9$ bit/s in our case. We add 20 bytes to the packet size s_p in the equation because we must also consider the additional data and gaps required by OSI layer 1 to transmit an Ethernet frame (preamble, start frame delimiter, and interpacket gap). Assuming a packet size of 64 bytes and a gigabit link, we thus calculate a timestamping latency of 1.34 µs.

$$\begin{aligned} t_{ts} &= \frac{2 \cdot (s_p + 20)}{r_l/8} \\ t_{ts} &= \frac{16 s_p + 320}{r_l} \text{ with } r_l = 1 \cdot 10^9 \,\text{bit/s} \\ t_{ts} &= \frac{16 s_p + 320}{1 \cdot 10^9} \end{aligned} \tag{5.7}$$

5.5.3 Packet generator

The packet generator puts load on the firewall by generating and sending packets to the firewall. The firewall forwards the packets back to the packet generator so that it can process the timestamps in the packets and calculate the latency. This means, the packet generator is the sender and receiver of the packets.

5.5.3.1 Hardware

We use a PC as packet generator that we equipped with two additional NICs. It uses an Intel Core i5 3450 CPU with 4 cores running at a base clock of 3.1 GHz and has 8 GB of RAM.

We need two additional NICs because the integrated NIC with a controller by Realtek is not supported by DPDK. The first NIC uses an Intel 82576 controller and supports a data rate of up to 1 Gbit/s. Although the NIC has two ports, we need an additional NIC because there were degradations of the throughput if we load both ports at the same time. The second NIC also supports a data rate of 1 Gbit/s and uses an Intel 82574L controller.

5.5.3.2 Software

On the packet generator, we run Debian 11 with kernel 5.10.70. For the generation of the packets, we use MoonGen [45] which is available open source on GitHub [46]. MoonGen is a scriptable packet generator that can generate a high rate of packets thanks to the use of DPDK. It can be scripted with the scripting language Lua to manipulate and process packets as desired.

We wrote a script for MoonGen that meets all our needs. It generates packets, sends them to the firewall, and receives them again for evaluation. The configuration is performed mostly using command line arguments (except for the packet pattern). All notable features of our packet generator script are explained in the following sections.

5.5.3.3 Packet generation

To simulate different load scenarios, we can configure the packet size and data rate using command line arguments. During packet generation, we cannot change the packet size and data rate, but we can adjust the header fields. For example, we can send packets alternatingly with UDP source port 1234 and 1235. For this, we wrote a small and basic library that allows setting the header fields for each generated packet individually.

During our automatic measurements, we want to ensure that we always measure the latency of the same number of packets. This ensures that our measurements are comparable. To stop the packet generator at the right time, we added a command line argument to configure the number of packets to measure. Alternatively, we can also configure a measurement duration in seconds.

When we put a constant and high load on the firewall, the first packets usually have a lower latency than the later packets and are not affected by packet loss. The reason for this is that the ingress queue of the firewall is initially empty and fills up with packets until the number of packets in the queue settles. Thus, high latency and packet loss will not affect the initial packets but they will affect all later packets. The phase at the beginning where the behavior of the measured system is not equal to the long-term behavior is called *transient phase*. We added a command line argument to our MoonGen script to prevent that the transient phase affects our measurements. It accepts a duration in seconds that tells our script to start packet generation but waits for the given amount of time until it measures the latency.

5.5.3.4 Latency calculation

A huge advantage of our script is the on-the-fly extraction of the timestamps out of the packets. On-the-fly extraction means, the packet generator reads the timestamps from the packets and then calculates the latency. After calculating the latency, our script writes the result (start timestamp, end timestamp, and latency) to a CSV file. The alternative to the on-the-fly extraction would be to save each packet on the disk and extract the timestamps when the packet generation has finished using another tool. This has two disadvantages.

The first disadvantage is that the amount of data that is stored on disk is much larger than if just timestamps and the calculated latency are saved. Our measurement of a packet has only a size of 30 B while a packet is at least twice as large. Therefore, we save much disk space if we extract the timestamps on the fly and save the result instead of the whole packets. Additionally, saving whole packets requires that the disk can keep up writing the packets at the rate of the generated packets.

The second disadvantage is the additional time that we need to spend on the measurements because an additional step is required. First, the packet generator performs the measurement and after that, another tool must extract the timestamps.

On the other hand, on-the-fly timestamp extraction requires more processing during the measurement. But timestamp extraction runs on a different CPU core than the packet generation. Thus, timestamp extraction only slightly reduces the maximum rate of packets that can be generated and received again by the packet generator. The reason for the reduction is that packet generation can generate more packets per second than the timestamp extraction can process. Hoswver, since the difference is small, the timestamp extraction reduces the maximum data rate that our packet generator can process only by a few Megabits per second.

5.5.3.5 Packet loss measurement

In addition to the latency calculation, our packet generator also measures the packet loss and saves the result in a CSV file. For this, it uses the number of sent and received packets. But there is a small error in the calculated packet loss in the range of a few packets, depending on the packet rate. This is because we could not perfectly synchronize the counting of sent and received packets. There are two reasons for this:

The first reason applies if the packet generator is configured to skip the transient phase. In that case, some packets were already sent before the beginning of the measurement and they are therefore not included in the number of sent packets. But the same previously sent packets are included in the number of received packets when they arrive after the measurement started.

The second reason is that some packets were sent right before the measurement was stopped and are therefore included in the number of sent packets. But they are not included in the number of received packets because they arrive when the measurement was already stopped.

5.5.4 Internal performance measurements

We already explained the latency measurements using the timestamping switch (*external performance measurement*). But we also added some code to VPP that allows us to measure the duration of the ACL checks (internal performance measurement). The accuracy of the measurements lies in the range of nanoseconds and they can be enabled and disabled via CLI and API.

For the internal performance measurements, we had to find a balance between flexibility and impact on the packet latency. The result is that we can only measure a predefined number of packets per measurement. We must configure the number of packets that VPP should measure when we enable the measurements. As a result, VPP overwrites old measurements if more packets arrive than measurements can be saved. When we disable the measurements, VPP saves the result as a CSV file on the disk.

The internal measurements measure not only the latency but also save other helpful information about each packet, mostly about ACL processing. In the following, we explain all measured information except the latency.

Thread index The thread index shows which worker thread processed a packet. We can use it to see whether the packet classification of the NIC works and to track the packet latency on the specific worker thread (i.e., packets applying to specific priority list rules).

Adaptive time limit The adaptive time limit measurement shows the value of the adaptive time limit that VPP calculated for a packet. It helps us to tune the adaptive time limit control because we can analyze how the controller behaves if the packet rate (i.e., firewall load) changes.

Effective time limit The effective time limit measurement shows whether VPP used the adaptive time limit or the configurable time limit to limit ACL processing of a packet. If the effective time limit and adaptive time limit are equal, the adaptive time limit was used. If the effective time limit and adaptive time limit differ, the configurable time limit was used.

Number of processed rules The number of processed rules shows how many ACL rules VPP was able to process within the effective time limit. It equals the position of the matching rule if VPP found a matching rule within the time limit. Otherwise, it equals the number of rules that VPP was able to check before the time limit exceeded. Knowing the number of processed rules helps us to find a balance between the length of the time limit and security. So we can estimate how many rules are most likely matched by VPP if we configure a certain time limit. But this value is largely dependent on the processing power of the firewall and only applicable to the specific firewall model.

Index of the matched priority list rule The index of the matched priority list rule measurement shows which rule in the priority list matched the packet. It helps us to identify whether a packet was a priority packet or a non-priority packet. As a result, for example, we can analyze the latency of non-priority packets and priority packets independently of each other.

Vector size The vector size can be used to estimate the load on the firewall. This helps us to analyze the change of latency and adaptive time limit based on the firewall load. Note that multiple packets in a row have the same vector size if it is larger than 1 because these packets were in the same vector. For example, if the vector size is 3, three packet measurements in a row have the same vector size. This must be considered when calculating, for example, the average vector size. To calculate the correct statistics, the duplicated vector sizes must be removed first.

To sum it up, internal performance measurements yield a lot of helpful information. They help to analyze how the firewall behaves in different scenarios and to improve its behavior based on the results. But care must be taken to keep the performance impact of the internal measurements on the packet latency low. We analyze the performance impact of our internal performance measurements on the firewall in Section 7.4.

5.5.5 Automation

Starting various measurements with different packet sizes, data rates and a different number of ACL rules to simulate different load scenarios is tedious. But it is important to evaluate the success of the implementation after each implementation step (e.g., after the implementation of an idea). Because of this, we wrote a bunch of scripts using the scripting language Python that automatically performs measurements with configurable parameters. In addition, the scripts also create a report, so we can get an overview of the latency at a glance. The report consists of statistics (e.g., mean and maximum) and various plots of the measurements. It is also possible to create a report that compares two measurements with each other using statistics and plots. In the following, we present those scripts to give a brief overview of them. We also show what ACL and priority list rules we configure on the DUT.

5.5.5.1 Main script

The main script orchestrates all other scripts and the devices. It is also the central point for configuration. We can configure the script at each start with the needed configuration.

Based on the configuration, the main script starts the other scripts with different configuration options or it does not start them at all. Another task of the main script is to download the measurement results from the packet generator (external performance measurements) and the DUT (internal performance measurements). To download the results, the main script uses SFTP.

The main script can be started from any device as long as it is connected to the management switch. This means, the main script must configure the packet generator, the timestamping switch, and the DUT remotely using SSH. The main script expects that some helper scripts are placed on the remote devices and on the local device. Their location and the IP addresses as well as the credentials of the devices are currently hardcoded in the scripts. Our main priority was to make them work for our specific setup, not to keep them flexible. But extending the scripts later to make them more flexible is possible.

5.5.5.2 ACL installer script

We use the ACL installer script to add and delete ACL rules, priority list rules, and to enable or disable internal performance measurements (including the number of internal packet measurements to store). Thus, it is not only responsible to configure the ACL as the name suggests. The script is placed on the DUT and connects to VPP's binary API to configure it.

5.5.5.3 Report generator script and diff report generator script

We developed two different report generator scripts. We call the first script report generator because it generates a report about the just performed measurements. We call the second script diff report generator because it generates a report that compares two bunches of measurements. Usually, we compare the just performed bunch of measurements with another bunch of measurements. The main script uses both report generator scripts twice, for external and for internal performance measurements.

The reports consist of statistics about each measurement (packet loss, mean latency, median latency, min latency, max latency) and various plots that are in parts similar to the plots shown in this thesis. The diff reports use fewer plots and instead focus on comparing the statistics of two different bunches of measurements (e.g., change of mean latency).

To create nice-looking reports as easily as possible, we use Jupyter notebooks that we wrote in Python. Jupyter notebooks support all features we need like determining the presentation of the results and exporting them in a portable format. But instead of manually running Jupyter, the main script invokes it automatically to render the notebooks as HTML files so that the results can be viewed in the browser.

6 Implementation

In the last chapter, we discussed the design of the ideas that we initially presented in Section 2.4. In this chapter, we present how we implemented the ideas in VPP. We also discuss how we make sure that our implementation increases the latency and jitter as little as possible.

6.1 Measuring and limiting the ACL processing time

In order to implement the timebound idea to stop ACL processing after a given time limit, we must know how long the ACL processing is already ongoing. For this, we first get the current time that we then use to check whether the time limit exceeded.

6.1.1 Getting the current time

VPP offers two functions that we can use to get the current time. First, we can use the `clib_time_now` function[1]. It returns the number of seconds since VPP started with a precision of 1 µs [47].

An accuracy in the range of microseconds is not exact enough for us because packets must be forwarded in less than a microsecond if the packet rate is high (see our discussion about interarrival times in Section 2.3). Therefore, we use the second function that VPP offers, the `clib_cpu_time_now` function. It returns the number of CPU clock cycles since the CPU was started. To get the number of clock cycles, `clib_cpu_time_now` uses a CPU instruction that is called `rdtsc` on x86 CPUs. Its accuracy is dependent on the CPU frequency. Assuming a frequency of 1.6 GHz, we get an accuracy of around 0.625 ns (this is the period of one clock cycle). Since current CPU models count the clock cycles based on the CPU's base frequency, we can convert clock cycles to seconds or vice versa. VPP reads the base frequency and makes it available in the code.

6.1.2 Limiting the ACL processing time

Figure 6.1 shows how we extended the ACL matching logic explained in Section 3.3.3.8 to limit the ACL processing time (the two extensions are highlighted in green). In

[1]Bonus info (optional): In VPP, we do not get the current time from the kernel. Instead, VPP gets the time from the kernel approximately every 16 seconds and interpolates it in between using the (less accurate) CPU clock [47]. VPP does this to avoid expensive system calls. In addition, VPP ensures that the clock is monotonic and therefore resistant to changes of the system time (i.e., the time does not go backward).

the first extension (the upper green box), we added some code before the actual ACL matching loop. We use this code to calculate, out of the given time limit, the point of time in the future after which the matching should be stopped. We do this by converting the time limit from seconds to the number of clock cycles by multiplying it with the number of clock cycles per second. Then we get the current time using `clib_cpu_time_now` and add the converted time limit to it. The result is the number of clock cycles after which matching should stop. We store the point of time in clock cycles, not in seconds because we compare it later with the output of `clib_cpu_time_now` and thus do not need to convert it. Listing 6.1 shows the above calculation as pseudo code.

```
time_limit_clocks =
  clib_cpu_time_now() + time_limit * clocks_per_second;
```

Listing 6.1: Calculation of the point of time at which matching should stop

Our second extension of the code is inside the matching loop and is executed before the matching of each rule (the lower green box in Figure 6.1). It is the place where we check whether the time limit exceeded. The check is very short so we can show it in Listing 6.2. We only compare whether the current number of clock cycles is larger than the point of time in clock cycles that we calculated earlier using the time limit. If the latter is true, the time limit exceeded, we store the next rule that must be matched (for analyze later), and stop matching. In addition, we check whether the variable `check_time_limit` is true. It allows us to check the ACL without time limit in case it is needed. Because the matching function is an inline function, the check whether to enable or disable the time limit produces no overhead. Instead, the compiler “hardcodes” it (as long as we assign a constant value to `check_time_limit`).

```
if (check_time_limit
     && (clib_cpu_time_now() > time_limit_clocks))
{
  // Time limit exceeded
  // Store next rule that must be matched and stop matching
}
```

Listing 6.2: Code that checks before each rule check whether the time limit exceeded

We tried to keep the additional processing, required to check the time limit, as low as possible. Therefore, we calculate the point of time and convert it to clock cycles beforehand. So we can reduce the time limit check during matching to a comparison of the current time with a future point of time.

In summary, by checking the time limit before the matching of each rule, we can detect the exceeding of the time limit very soon and stop further matching. But we only limit the execution of stateless filtering – stateful filtering is not included. However, since stateful filtering causes only a small and constant delay, this is no drawback.

Even though we implemented the time limit checks as lightweight as possible,

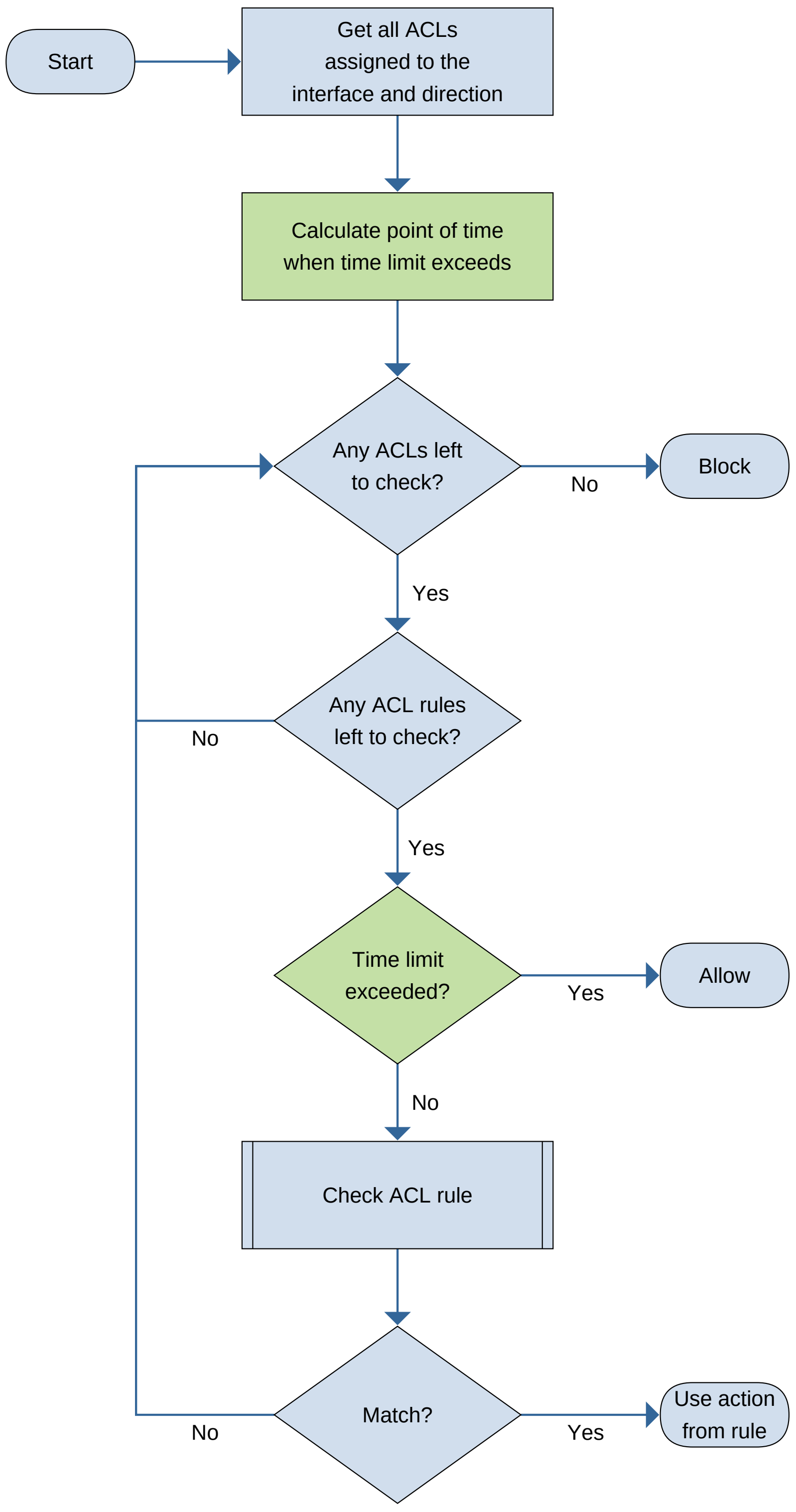

Start
Get all ACLs assigned to the interface and direction
Calculate point of time when time limit exceeds
Any ACLs left to check?
No
Block
Yes
Any ACL rules left to check?
No
Yes
Time limit exceeded?
Yes
Allow
No
Check ACL rule
Match?
No
Yes
Use action from rule

Figure 6.1: Flowchart showing the ACL plugin matching mechanism with additional matching time limitation (green)

they still require some processing that is not neglectable in comparison to ACL matching. For example, if the matching of a rule (see the flowchart in Figure 3.11 on page Page 41) already ends after the IP version or the destination IP check, the fraction of instructions for the time limit check is significant. On the other hand, if the matching of a rule traverses the whole matching code, the fraction of instructions for the time limit check is small. In Section 7.5.1, we measure the impact of the time limit check on latency to get an overview of the performance impact.

6.2 Timebound

In this section, we build upon the capability of limiting the ACL processing time that we presented in the last section – this means, we focus on the implementation of how we determine the actual time limit. More precisely, we discuss the implementation of the configurable time limit and the adaptive time limit whose design we presented in Section 5.2. Moreover, we discuss the implementation of analyze later that continues ACL processing in the background (the design of analyze later can be found in Section 5.2.3).

6.2.1 Configurable time limit

The configurable time limit limits the ACL processing time of packets based on the flow to which they belong. To configure which flows should be limited to which duration, we introduced the priority list.

We decided to design and implement the priority list similar to the ACL plugin's ACL because it offers flexible rules. Nevertheless, it is still possible to extend the priority list with more fields like the VLAN ID. The difference between the ACL and the priority list is that we replaced the action field with a time limit field. Moreover, the ACLs can be assigned to interfaces, while this is not possible with the priority list. Instead, the priority list is only one single list that contains all rules and does not differentiate between interfaces.

To allow the configuration of the priority list, we added two API endpoints. One to add rules to the priority list (this deletes all previous rules) and one to delete all rules from the priority list. These two endpoints are the minimum required endpoints. While they are not flexible, they are sufficient for our measurements in Chapter 7. It is possible to extend the API endpoints in the future and to add CLI commands.

We added the priority list check just before the ACL check as Figure 6.2 illustrates (the priority list check is highlighted in green). If a rule in the priority list matches, it returns the time limit. Otherwise, it returns infinity as time limit (this is possible because we use floating point values).

We can then use the time limit either in conjunction with the adaptive time limit or we use it directly to limit the stateless ACL check. In the latter case, we must turn the time limit for the ACL check off instead of passing infinity (using the `check_time_limit` variable as described in Section 6.1.2).

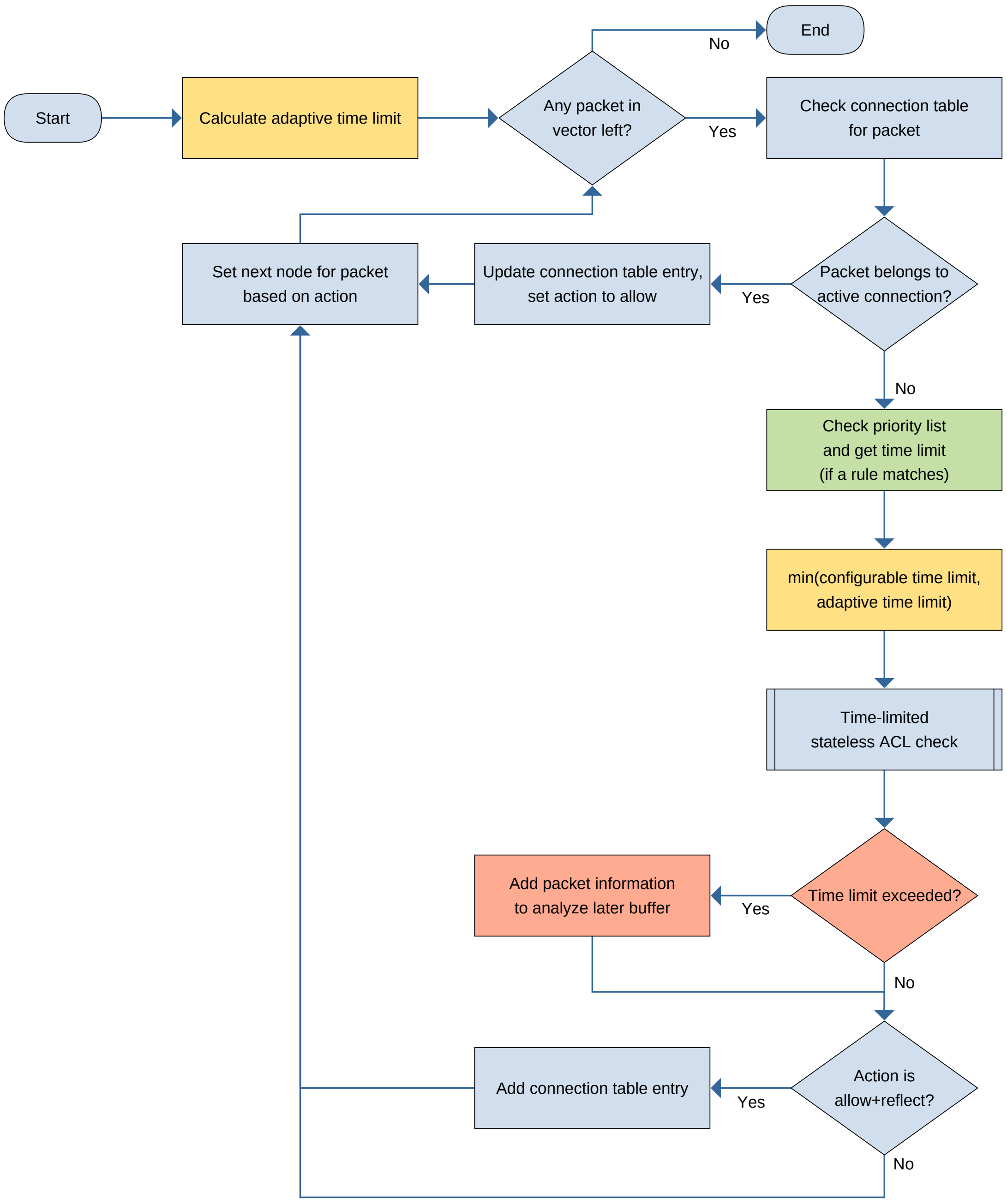

Start
Calculate adaptive time limit
Any packet in vector left?
No
End
Yes
Check connection table for packet
Packet belongs to active connection?
Yes
Update connection table entry, set action to allow
Set next node for packet based on action
No
Check priority list and get time limit (if a rule matches)
min(configurable time limit, adaptive time limit)
Time-limited stateless ACL check
Time limit exceeded?
Yes
Add packet information to analyze later buffer
No
Action is allow+reflect?
Yes
Add connection table entry
No

Figure 6.2: Flowchart showing the ACL plugin structure with additional configurable time limit (green), adaptive time limit (yellow) and analyze later (red)

6.2.2 Adaptive time limit

The time limits determined by the configurable time limit can be too long if the packet rate dynamically changes. Therefore, we designed the adaptive time limit in Section 5.2.2. It adapts the time limit to the current firewall load so that packets do not delay following packets for too long.

6.2.2.1 Measuring the firewall load

Among the different possibilities to measure the firewall load that we introduced in Section 5.2.2.1, we decided to use the vector size. However, we use a slightly different implementation that is simpler to implement. We measure the vector size inside the ACL processing node instead of directly at the ingress queue. As a result, we only take vectors into account that traverse the ACL. As explained in Section 3.3.3.3, vectors only traverse the nodes they need to. If they originate from a port that has no ACLs assigned, they do not traverse the ACL processing node. This means, that we do not measure the real firewall load in some cases. One case would be, for example, if the packet rate at a port is high but it has no ACLs assigned so that the large vectors do not traverse the ACL. In the ACL process node we would only see small vectors from ports with a low packet rate and thus wrongly assume a low firewall load.

In addition, with our implementation we ignore the fact that the vectors can originate from different ports. If the vectors originate from different ports, they can have a different size depending on the packet rate at that port (as explained in Section 5.2.2.1). This means that we get large and small vectors mixed together so we derive a varying firewall load from them. But in reality, the firewall load was constantly high, we only processed a vector from a port with a small packet rate.

The mentioned problems of our implementation are problematic in production environments but during our latency measurements we use only one port and all packets traverse the ACL. Therefore, we always derive the correct firewall load. The correct implementation would be to measure the vector size at the input nodes and then choose the largest of them as explained in Section 5.2.2.1.

6.2.2.2 Calculating the adaptive time limit

Figure 6.2 shows the locations of the ACL plugin where we implemented the adaptive time limit in yellow. Most processing happens before we iterate over the packets in the vector, that is, only one time for all packets in the vector (see the top left yellow process in Figure 6.2). There we measure the vector size and calculate the adaptive time limit because the vector size is the same for all packets in the vector. Thus, the adaptive time limit is also the same for all packets in the vector. The calculation of the adaptive time limit is exactly the same as we already described in Section 5.2.2.2: we use a P controller to calculate a time limit between 0.2 µs and 2 µs.

There is one small step that we still perform for each packet, but only in case we use the adaptive time limit in combination with the configurable time limit (see the yellow process on the right in Figure 6.2). To get the configurable time limit, we must check the priority list for each individual packet. This means, we have a different configurable time limit for each packet. According to Equation (5.5) on Page 52, we must choose the smaller of the two time limits to limit ACL processing. Therefore, we must determine the minimum of them for each packet individually.

6.2.3 Analyze later

If the time limit for the ACL check exceeds, analyze later continues the ACL check in the background so that the firewall can forward the packet without complete check. In Section 5.2.3, we pointed out that analyze later should run on a different CPU core, or specific to VPP, in a process node running on the main thread. This ensures that packet processing stays unaffected from analyze later. But we must take care that the communication between the ACL processing node and the analyze later node is as efficient as possible to delay the packets as little as possible.

6.2.3.1 Analyze later buffer size

The ACL processing node passes information about packets to check to the analyze later node using the analyze later buffer. While it would be favorable if the analyze later buffer had an unlimited amount of space, we have to limit it to a static number of entries to keep the performance impact minimal. In our implementation, the analyze later buffer can store up to 1 000 000 entries. We could increase its size further, but for our measurements this value is sufficient. It is easily possible to adjust the analyze later buffer size during compile time.

There are two reasons why we decided to make the analyze later buffer size static. The first reason is the performance cost of resizing the buffer dynamically during runtime. If we make the size of the analyze later buffer dynamic and we add an element to the buffer but the static array in the background is full, VPP must copy all data to a new location in memory. Because the analyze later buffer size is in the range of tens of Megabytes, this takes some time and blocks packet processing in the meantime. The result is an increased latency and an increased jitter. With a static buffer size, the memory is preallocated and VPP never needs to copy the buffer entries to another location.

The second reason why we decided to make the analyze later buffer size static is to prevent out-of-memory situations. If we make the analyze later buffer size dynamic, VPP can allocate an arbitrary amount of memory for the buffer if many entries must be stored. In worst case, the analyze later buffer allocates so much memory that almost no more memory is left (or there is no contiguous space left to store such a large array). This can hinder other VPP functions from allocating new memory for data structures so that errors occur. To prevent VPP from running out of memory, we should limit the buffer size so the buffer allocates a reasonable amount of memory. Making the buffer size static is the easiest way to limit the buffer size and ensures, the requried memory is always reserved and available.

In Section 5.2.3.2, we listed what information about a packet must be passed to analyze later. Since our buffer has a capacity of 1 000 000 packet information entries, the size of an individual entry has a significant impact on the memory consumption

Data	Size
5-tuple, TCP flags	60 B
ACLs that must be checked (pointer)	4 B
Next rule and corresponding ACL that must be checked (indexes)	8 B
Total	72 B

Table 6.1: Amount of memory required to store the packet information of one packet for analyze later

of the buffer. Table 6.1 lists the memory requirement in bytes for each type of data of which a packet information entry consists. In total, one entry in the buffer requires 72 bytes of memory which makes our buffer to use 72 MB of memory.

6.2.3.2 Accessing the analyze later buffer

Our analyze later buffer implementation uses the ring buffer structure offered by VPP. A ring buffer is like a queue, it has a beginning and an end. The regular ring buffer operations are enqueue (append entry at the end), peek (read first, i.e., oldest entry), and dequeue (delete first entry). This is ideal for us because we can enqueue the packet information at the end of the buffer and then read and delete it in the order we enqueued it. If it is really needed, we could also access any entry in the middle of the buffer (but we do not make use of this).

Another advantage of the ring buffer, besides the processing in the order in which we enqueued the entries, is the constant and short access time. All regular ring buffer operations above have a time complexity of $O(1)$. This means, we introduce no jitter through the buffer access. And because the operations are very simple, the access is very fast.

However, the buffer access time is not the only delay when accessing the analyze later buffer in our case. Both, the ACL processing node and the analyze later node access the analyze later buffer from two different threads. But parallel access by both nodes could leave the buffer in an inconsistent state or could return invalid data if the timing is unfavorable. Therefore, we must prevent that both nodes can access the buffer at the same time. To do this, both nodes lock the buffer before they access it and unlock the buffer after accessing it (we use the spinlocks that are already available in VPP). Listing 6.3 shows an excerpt from the VPP source code how we lock the analyze later buffer before accessing it and unlock it after we finished accessing it.

If a node tries to lock the buffer if it is already locked, it must wait until the buffer is unlocked again. This is an active wait, meaning, the node does nothing else but waiting for the buffer being unlocked. As a result, packet processing cannot continue if the ACL processing node tries to lock the analyze later buffer while the analyze later node already locked the buffer. The consequence is an increased packet latency and an increased jitter. But because the ring buffer operations are very fast, the nodes must only lock the buffer for a short time. Therefore, the impact on latency and jitter is small. We evaluate the impact on the latency and jitter in Section 7.6.

	Single-core	Multi-core
Suspend time	11 µs	15 µs
Runtime limit	0.4 µs	400 µs

Table 6.2: Amount of time that the analyze later node suspends/operates in single-core and multi-core mode

```
// Lock buffer (and actively wait if buffer is already locked
// until it is unlocked again)
clib_spinlock_lock_if_init(&analyze_later_buffer_lock);

// Delete oldest analyze later buffer entry
clib_ring_deq(analyze_later_buffer);

// Unlock buffer
clib_spinlock_unlock_if_init(&analyze_later_buffer_lock);
```

Listing 6.3: Example of accessing the analyze later buffer and locking it before (from the VPP source code)

6.2.3.3 Analyze later node

The analyze later node is responsible for background ACL checks. It is a process node and thus runs on the main thread. Process nodes share their resources with other nodes, so we must make sure not to block other nodes for too long (see Section 3.3.3.6 for more information about how VPP schedules nodes).

Analyze later node scheduling To control the analyze later node run time, we define a suspend time and a runtime limit. The suspend time determines how long the analyze later node suspends until the VPP scheduler wakes it up again. The runtime limit determines how long the analyze later node is allowed to run, or more precisely, how long the ACL checks should maximally run in total.

We set the suspend time and runtime limit differently based on whether VPP runs on a single core or on multiple cores. Table 6.2 shows the exact values. If VPP runs on a single core, the analyze later node must share the processing resources with the packet processing nodes. To prevent high latencies, we set the runtime limit much lower than in multi-core mode to block the packet processing as little as possible. We also reduce the suspend time to compensate for the reduced runtime limit. But we have to set the suspend time to at least 10 µs because VPP interprets all values below 10 µs as zero and does not suspend. As a result, the analyze later node runtime is too short for background ACL processing in single-core mode if we expect a high rate of packets. Additionally, we already stated the contradiction of running analyze later on the same core as packet processing in Section 5.2.3.1. Thus, we do not recommend running VPP on a single core.

There are two drawbacks due to the analyze later node scheduling. Since the analyze later node always suspends for a configured amount of time, it cannot instantly continue processing if the ACL processing node enqueued new packet information.

Additionally, the analyze later node wakes up unnecessarily even if no packet information is available to process. As a solution, VPP also offers a feature to wake up process nodes using events. We first wanted to use this feature in combination with the suspend feature. This means, the ACL processing node uses the event feature to wake up the analyze later node if there are packets to process. Additionally, the analyze later node can suspend and wake up by itself if it could not finish background ACL processing within the runtime limit. But we cannot use the event feature because it can only be used in single-core mode if the ACL processing node and the analyze later node run in the same thread. Thus, we have to use the polling-based approach and analyze later starts delayed with background processing instead of getting notified by the ACL processing node.

Analyze later node operation Figure 6.3 shows how the analyze later node operates. After waking up, it enters the background ACL check loop (this is not the process node's main loop). The background ACL check loop runs as long as the runtime limit did not exceed and as long as packet information is available in the analyze later buffer. To keep track of the remaining node runtime, the analyze later node has a remaining time variable that it sets to the runtime limit after waking up. If packet information is available and the time limit is not exceeded, the analyze later node gets the oldest packet information entry from the analyze later buffer. Using the packet information, it starts the ACL check with the time limit set to the remaining time that the node is allowed to run. Analyze later runs the same code for the ACL check as the ACL processing node. If the check is finished, the analyze later node updates the remaining time by subtracting the duration of the ACL check (the ACL check returns how long it ran). If the analyze later node could complete the ACL check within the time limit, it deletes the corresponding packet information from the buffer. Based on the ACL action, the analyze later node should, according to our design, log wrong decisions. But we did not implement the logging of wrong decisions and instead increment counters. We explain more about this in Section 6.2.3.4. At this point, the analyze later node finished the background ACL check of a packet and returns to the beginning of the background ACL check loop. Depending on the remaining time and available entries in the analyze later buffer, it checks another packet or suspends.

6.2.3.4 Reaction to wrong decisions

According to the timebound idea (see Section 2.4.1), we should log wrong forwarding decisions. But we did not implement logging because we focused on implementing the ideas in a way so we can measure their impact on latency and jitter. Logging can be implemented in future and does not affect the latency and jitter of packet processing as long as analyze later runs on a different CPU core. When analyze later and packet processing run on the same CPU core, the latency and jitter increase depends on the logging mechanism.

To be able to at least track the number of correct and wrong decisions, we use counters that are already available in VPP. We have two counters, one to count correct decisions (ACL action was allow) and another one to count incorrect decisions

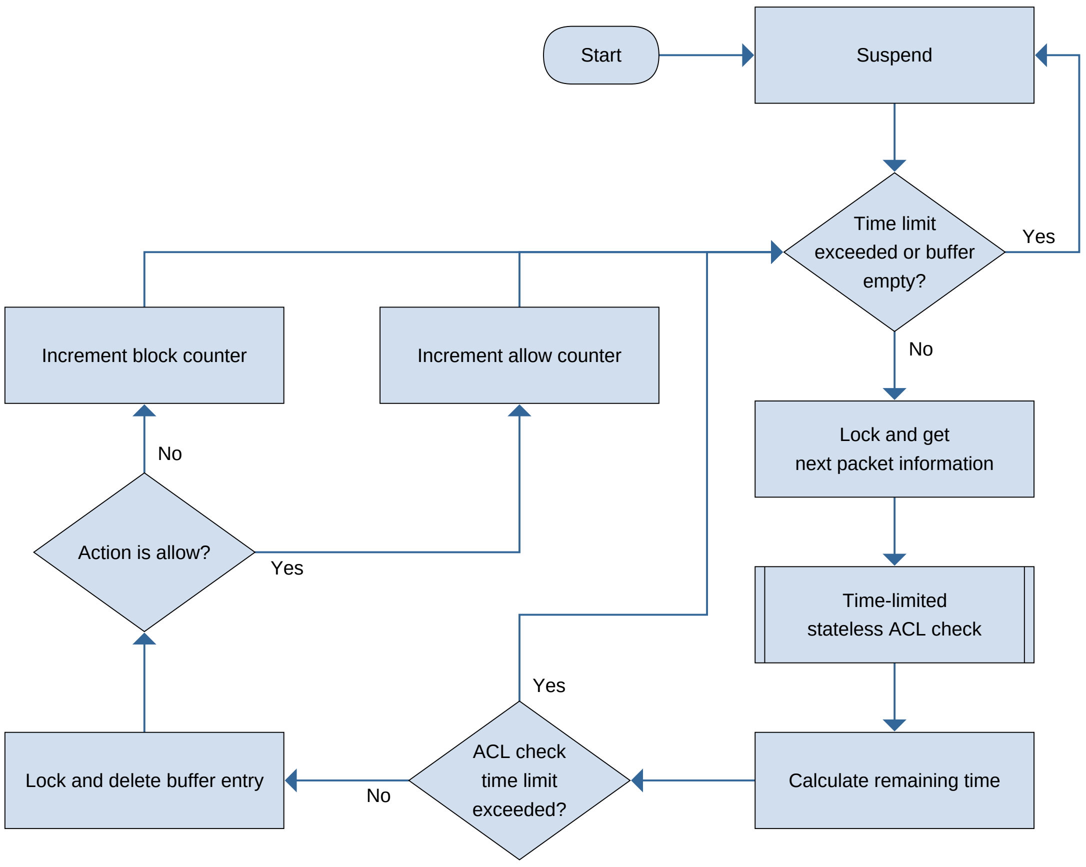

Start
Suspend
Time limit exceeded or buffer empty?
Yes
No
Lock and get next packet information
Time-limited stateless ACL check
Calculate remaining time
ACL check time limit exceeded?
Yes
No
Lock and delete buffer entry
Action is allow?
Yes
No
Increment block counter
Increment allow counter

Figure 6.3: Flowchart showing the analyze later node structure

(ACL action was block). For each correct or incorrect decision, the analyze later node increments the corresponding counter by one (see Figure 6.3). The counter values can be checked using the CLI (with the command `show errors`).

6.2.3.5 ACL processing node

Figure 6.2 (see Page 74) shows how we extended the ACL processing node to support analyze later (the analyze later parts are highlighted in red). After the ACL check, the ACL processing node checks whether the time limit exceeded. If that is the case, it checks whether the analyze later buffer has space left (we omitted this in Figure 6.2). If the analyze later buffer has space left, the ACL processing node adds the packet information to the analyze later buffer.

If the analyze later buffer has no space left, the ACL processing node should drop the packet according to our recommendation in Section 5.2.3.2. But we decided not to drop the packets in our implementation because it would interfere with our latency measurements. Since our measurements put a constant and, depending on the measurement, high load on the firewall, the buffer would fill quickly and packet loss would occur. But with packet loss, we would not be able to perform our latency measurements because we need the packets to calculate the latency and other statistics. To at least keep track of how many packets cannot be processed by analyze later, we increment a counter for each packet that cannot be added to the buffer because it is full.

6.3 Passive

We did not implement any additional features in order to support the passive idea. Instead, we use the timebound idea to configure a time limit of zero using the configurable time limit. This has the disadvantage that VPP still executes the ACL check code up to the point where the time limit is checked. As a result, there is a small overhead because the adaptive time limit calculation (if it is implemented) and the initial code to start the ACL check are executed. But since this is a proof of concept and we already save time through the omitted ACL check, this implementation is sufficient.

6.4 Priority

We already explained in Section 5.4 that the priority idea works best if we separate high-priority and non-priority packet processing to run on different CPU cores. The firewall does not classify and separate the packets by itself in this case. Instead, we configure the NIC to classify the packets and place them in different queues. In this section, we explain how we modified and configured VPP to achieve this.

6.4.1 Configuration of VPP

We must extend the VPP configuration to tell VPP on how many CPU cores it should run and how many ingress queues it should create for each port. Listing 6.4 shows the relevant parts from the configuration file at the example of our testbed.

In the `cpu` section, we configure VPP to create two worker threads using the `workers` parameter. As a consequence, VPP requires three CPU cores to be available on the system (the main thread runs on one of the three cores).

Even though we changed the number of worker threads, VPP creates only one ingress queue per port. We must configure the queues for each port individually in the `dpdk` section. In our case, we configure one port to have two ingress queues using the `num-rx-queues` parameter. When we run VPP with this configuration, VPP configures the NIC to enable RSS. With RSS enabled, the NIC distributes the packets among both queues. However, we do not want such behavior because our goal is to place all packets in the default queue unless we configure VPP otherwise. Because of this, we “disable” RSS by configuring VPP to only use the first queue (queue 0) for RSS. We do this with the parameter `rss-queues`.

```
cpu {
  # Configure VPP to create two worker threads
  workers 2
}

dpdk {
  # This port transmits packets only
  # So we do not need multiple ingress queues
  dev 0000:06:00.0

  # This port receives the packets from the packet generator
  # Ports are addressed using the PCI adress in the config
  # (not with the port id used in VPP)
  dev 0000:05:00.0 {
    # Configure two ingress queues
    num-rx-queues 2
    # "Disable" RSS
    # Ensures the NIC places all packets in queue 0 by default
    rss-queues 0
  }
}
```

Listing 6.4: Relevant parts of the VPP configuration to be able to separate high-priority packets (at the example of our testbed)

6.4.2 Priority list

We already explained in Section 5.4.2.1 that we use the priority list to configure which packets the NIC places in which queue. For this, we extend the priority list with a field for port-queue pairs. With port-queue pairs, we can specify for each port individually in which queue the NIC should place matching packets.

Src IP	Src port	Dst IP	Dst Port	Proto	TCP flags	Time limit	port	queue
10.0.1.2	-	10.0.2.2	8080	UDP	-	0.6 µs	0	1
10.0.2.1	-	10.0.3.2	8098	UDP	-	1.0 µs	1	1
10.0.3.2	-	10.0.2.5	8099	TCP	-	1.5 µs	0	1
10.0.2.5	8099	10.0.3.2		TCP	-	1.5 µs	1	1

Table 6.3: Example of a priority list in our implementation that supports a single port-queue pair per rule

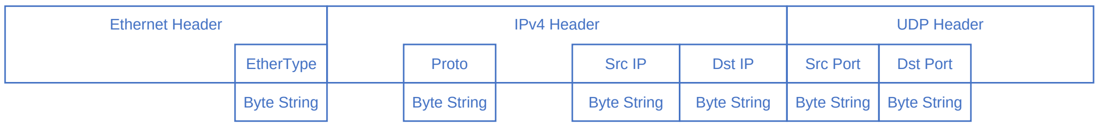


Figure 6.4: Visualization how the `RAW` pattern items match the 5-tuple fields of a packet. Each byte string belongs to a `RAW` pattern item.

However, we simplified the configuration of port-queue pairs in our implementation. Instead of multiple port-queue pairs, we only support one port-queue pair per priority list rule. As a consequence, we can only configure one port per priority list rule that supports the prioritization of matching packets. Since the firewall in our testbed only receives high-priority packets on one port, this is no limitation for our latency measurements. Table 6.3 shows the fields of the extended priority list, including a few example rules.

6.4.3 Representation of the priority list on the NIC

We use DPDK's generic flow API (see Section 3.3.2.2) to configure the priority list rules on the NIC. As we already explained in Section 3.3.2.3, the generic flow API is very limited on the I210 Ethernet controller that our firewall NIC uses. To support the priority list as much as possible, we used the flex filter to recreate an ACL.

This means, when VPP configures a rule on the NIC, it creates a flow rule whose matching pattern consists of multiple `RAW` pattern items. We use a separate `RAW` pattern item for each header field that we match:

- EtherType (to check whether the packet is an IPv4/IPv6 packet)
- Source IP address
- Destination IP address
- IPv4 protocol/IPv6 Next Header
- UDP/TCP source port
- UDP/TCP destination port

This way, we can dynamically include or omit a matching pattern in the flow role based on whether the ACL rule defines a value for the ACL field. Figure 6.4

also visualizes how the `RAW` pattern items match the header fields. Each byte string in the figure belongs to one `RAW` pattern item and is positioned in a way that it overlaps the corresponding header field in the packet. For all flow rules, we set the ingress attribute to signify that the NIC should apply the rule in ingress direction. As action, we set the `QUEUE` action and specify the queue that is given in the priority list rule.

With this recreation of an ACL, we support the matching of IP addresses, the protocol, and the ports. Due to the limitations of the flex filter, we cannot support the following ACL features:

- IP address ranges are not supported (only a single source and destination address, respectively)
- Port ranges are not supported (only a single source and destination port, respectively)
- The position of fields in the packet must not be shifted (shifts are caused by, e.g., a VLAN tag or IPv4 header options that are inserted between other headers)
- TCP flags are not supported because we could only match the whole byte, not single bits

If an ACL rule defines these unsupported features, VPP ignores them when configuring the NIC. For example, if a rule defines the IP network 10.1.1.0/24, VPP configures the IP address 10.1.1.0/32. Moreover, we did not handle the case if too many priority list rules are configured that cannot be stored on the NIC due to the limit of eight flow rules. Except for a log message, we silently ignore such a case. While these limitations are too restrictive on a production firewall, it is sufficient for our proof of concept. For production firewalls, we recommend using another NIC with an Ethernet controller that supports 5-tuple matching and that can store more flow rules. Or as an alternative, we could switch to VLAN tag matching (e.g., the VLAN priority) which is supported by the I210 controller according to its data sheet [23].

6.4.4 Automatic NIC configuration

Every time an administrator adds or deletes priority list rules, VPP must reconfigure the corresponding NIC to match the updated priority list rule(s). In other words, VPP must configure or delete the flow rules on the NIC. VPP already includes functions to manage flow rules. These functions abstract the generic flow API and keep track of the rules that VPP configured on the NIC. Keeping track means that these functions store or delete references to the flow rule objects in the memory so that VPP remembers which rules it configured on the NIC. This ensures that we can configure flow rules on the NIC and delete them again using the reference to the flow rule object.

VPP also includes functions to create flow rules. However, VPP uses many pattern item types that the NIC model of our firewall does not support (or does only support partially). Therefore, we added another function that accepts a priority list rule and creates a flow rule according to the flex filter (as explained in the last section). We can then use the other functions already available in VPP to configure the created flow rule on the NIC or to delete it.

6.5 Internal performance measurements

As we already described in Section 5.5.4, we measure the duration of the ACL check and collect other information about each packet using internal performance measurements. In this section, we show how we measure the duration of the ACL check and how we buffer the measurement results.

Figure 6.5 shows where we start and stop measuring the duration in green. We start before the priority list check and stop after adding the packet information to the analyze later buffer. As a result, we measure the duration of the configurable time limit (priority list check and choosing the minimum time limit), the ACL check, and analyze later (the part that runs in the ACL processing node).

To measure the duration, we use the `clib_cpu_time_now` function that we also use to measure and limit the ACL processing time (see Section 6.1). This means, we get an accuracy in the range of nanoseconds. We call the `clib_cpu_time_now` one time when we start measuring the duration to get the start time and one time when we end measuring the duration to get the end time. After getting the end time, we calculate the duration by subtracting the start time from the end time. At this point, we also get other information about the packet like the thread index and the adaptive time limit. Finally, we enqueue the measurement result to a ring buffer.

We try to keep the performance overhead of the internal performance measurements as low as possible. Because of this, we allocate a static ring buffer. We can configure its size when we enable the measurements using the CLI or API. A buffer with a static size prevents sporadic delays caused by increasing the buffer size if no more space is available in the buffer. This is the same effect as with the analyze later buffer in Section 6.2.3.1. The disadvantage is that if the buffer is full, VPP begins overwriting the oldest measurements in the buffer. We compensated for this drawback, as written above, by allowing to configure the desired buffer size when enabling the measurements. We do not set any limits regarding the buffer size but the amount of available memory on the firewall should be kept in mind. One entry (i.e., measurement result) in the buffer occupies 64 bytes of memory, this means, a buffer for 1 000 000 entries occupies 64 MB of memory. To compare, our firewall has 8 GB of memory available. Only when we disable the internal performance measurements, VPP saves the measurement results to a CSV file. This prevents any delays due to file access during the measurement.

If VPP runs multiple threads, it creates multiple buffers for the internal performance measurements (one for each worker thread and also one for the main thread). This way, we prevent that multiple threads access the same buffer. If multiple threads shared the same buffer, we would have to use locks to prevent parallel buffer access by the threads. This is also the reason why we do not write the measurement results to a file with the main thread at the same time as the worker thread enqueues the measurement results to the buffer. However, one buffer per thread also means that we multiply the memory consumption of the buffers because all buffers have the same size, independent of the number of threads.

6.6 Kernel configuration tweaks to improve VPP performance

If we run VPP on a Linux-based distribution with default settings, we leave some unused potential regarding performance. With some tweaks to the Linux kernel configuration, we could further reduce the jitter when running VPP. In this section, we discuss the configuration changes we made to the Linux kernel. We present the results of the optimization in Section 7.3. Our inspiration for the configuration was a paper by Stylianopoulos *et al.* [35] and the FD.io Wiki [48]. We examined their suggestions and evaluated which of them reduce the jitter.

All following configuration changes can be made by configuring the kernel parameters that are applied at system startup. They can be usually configured in `/etc/default/grub` with the `GRUB_CMDLINE_LINUX_DEFAULT` parameter (at least in Ubuntu and Debian). For example, we use the configuration shown in Listing 6.5 on our firewall.

```
GRUB_CMDLINE_LINUX_DEFAULT = isolcpus=1-3 nohz_full=1-3 \
  rcu_nocbs=1-3 intel_iommu=enable iommu=pt
```

Listing 6.5: Linux kernel parameters that we configured on our firewall in the testbed (excerpt from the `/etc/default/grub` file)

6.6.1 CPU core isolation

By default, Linux schedules tasks on all available CPU cores. This means, Linux periodically suspends running tasks to run other tasks. This also affects VPP which means that packet processing interrupts periodically. As a consequence, the latency of some packets increases which in turn increases jitter.

As a solution, we can tell Linux not to schedule tasks on certain CPU cores. We do this with the `isolcpus` kernel parameter by assigning it the list of CPU cores that should be excluded from scheduling [49]. With the parameter configured, Linux no longer runs any tasks on the given CPU cores. Instead, we must manually assign tasks to these cores (which VPP does automatically).

Note that the `isolcpus` parameter is deprecated and instead cpusets should be used [49]. However, we still used the `isolcpus` parameter because the cpusets configuration is more complex and it would have taken considerably more time to get it running[2].

[2]We find that cpusets are more complex because they are configured during runtime while the `isolcpus` parameter takes effect at boot time. This has the effect that we have to move already

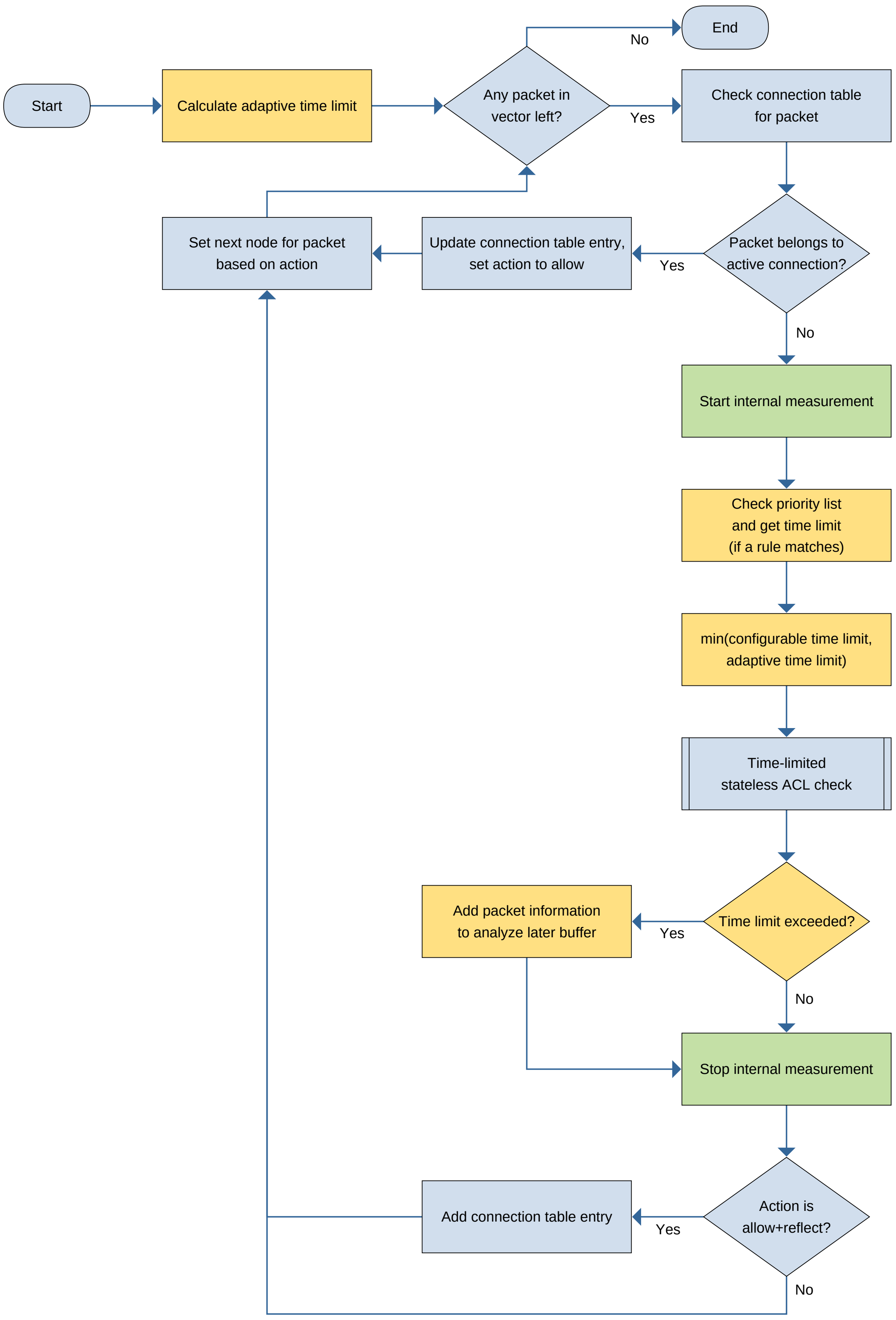

Start
Calculate adaptive time limit
Any packet in vector left?
No
End
Yes
Check connection table for packet
Set next node for packet based on action
Update connection table entry, set action to allow
Yes
Packet belongs to active connection?
No
Start internal measurement
Check priority list and get time limit (if a rule matches)
min(configurable time limit, adaptive time limit)
Time-limited stateless ACL check
Add packet information to analyze later buffer
Yes
Time limit exceeded?
No
Stop internal measurement
Add connection table entry
Yes
Action is allow+reflect?
No

Figure 6.5: Flowchart showing the ACL plugin structure with internal performance measurements (green) and all other timebound modifications (yellow)

6.6.2 Reducing the number of interrupts

Although we disabled scheduling on the CPU cores on which VPP runs, Linux still suspends the running task periodically because of interrupts that occur on the CPU core. We observed between 10 and 100 interrupts per second on a single CPU core with the default configuration on our firewall[3].

With the parameter `nohz_full`, we can prevent most of these interrupts on the given CPU cores. Additionally, we set the `rcu_nocbs` parameter, but this is not required because `nohz_full` sets it automatically [49].

Note that these parameters are not supported by many Linux distributions out of the box[4]. We had to compile the Linux kernel ourselves because Ubuntu does not set the required compile flag (`CONFIG_NO_HZ_FULL=y`).

6.6.3 Other configuration tweaks without effect

IOMMU We set the `intel_iommu` parameter to `on` and the `iommu` parameter to `pt` (passthrough) to bypass the I/O Memory Management Unit (IOMMU). The IOMMU degrades the performance and additionally, the DPDK documentation recommends turning the IOMMU off when using the `uio_pci_generic` kernel driver [51] which VPP uses by default to run the NIC. However, this change has no effect in our case because the IOMMU was already turned off.

Turbo Boost Stylianopoulos *et al.* [35] and the FD.io Wiki [48] also recommend turning off Turbo Boost. The Turbo Boost technology increases the CPU frequency above the base frequency if a CPU core is under high load and if the thermal budget and the power budget allow it [52]. However, disabling the turbo boost had no effect on our firewall because the CPU frequency was already constant at the base frequency. We did not further investigate the reason for this.

We did also look at the other configuration changes described in the FD.io Wiki [48] and in the documentation about the performance test environment of the VPP developers [53]. But none of them seem to bring any effect on our firewall.

running tasks away from the CPU cores that we isolated. Among these tasks are kernel threads which we could not move to other CPU cores. In hindsight, maybe it would work in combination with the `rcu_nocbs` parameter but we did not try it again.

[3]We roughly counted the number of interrupts using the command `watch -n1 "cat /proc/interrupts"`.

[4]Debian just recently (October 2021) set the required flag [50].

7 Results and discussion

In this chapter, we evaluate whether our ideas yield the expected results. We first introduced our ideas in Section 2.4, designed them in Chapter 5, and implemented them in Chapter 6. We expect to see improvements in latency and jitter regardless of the data rate, packet size, and rule count of the arriving packets. Moreover, we analyze our testbed on its limitations and investigate the overhead caused by our measurements to verify that our measurements are correct. Lastly, we discuss the security tradeoffs introduced by our ideas.

7.1 Testbed setup

We already introduced our testbed in Section 5.5 and gave an overview of the testbed architecture. In this section, we document the settings we used for the measurements in this chapter.

7.1.1 Packet generation

We designed two types of load that we put on the firewall:

1. UDP packets that are all the same.
2. UDP packets with two different source ports. We first send nine packets of one type (non-priority) followed by one packet of the other type (high-priority). The packet generator permanently repeats this pattern.

When generating packets of load type 2), the data rate of high-priority packets makes up 10 % of the total data rate. This way, we consider the fact that real-time industrial applications in general only exchange a small number of packets.

If we put a constant load on the firewall, the first packets do not reflect the long-term behavior of the firewall (transient phase, see Section 5.5.3.3). Therefore our packet generator waits five seconds after the packet generation starts before recording timestamps. After the five seconds elapsed, the packet generator records 10 000 timestamps for data rates of 10 Mbit/s and below, and 500 000 timestamps for data rates higher than 10 Mbit/s. Since we record a static number of timestamps, the duration of the measurement depends on the packet rate. The internal performance measurement records the same number of timestamps.

7.1.2 ACL and priority list rules

Table 7.1 shows the ACL rules we configure on the firewall. They belong to one ACL assigned in ingress direction to the port receiving the packets from the packet

Src IP	Src port	Dst IP	Dst Port	Protocol	TCP flags	Action
3.0.0.0.0/32	-	-	-	UDP	-	Allow
3.0.0.0.1/32	-	-	-	UDP	-	Allow
3.0.0.0.2/32	-	-	-	UDP	-	Allow
...	...	...	...	...	...	...
-	-	2.2.2.0/24	-	UDP	-	Allow

Table 7.1: ACL rules that are configured on the firewall (the first rules are placeholders of configurable number, the last one is the matching rule)

Src IP	Src port	Dst IP	Dst Port	Proto	TCP flags	Time limit	Port	Queue
-	1236	-	-	UDP	-	0.6 µs	3	1
-	-	-	-	-	-	1 µs	3	0

Table 7.2: Priority list rules for the configurable time limit and priority measurements (queue 1 is the high-priority queue). The catch-all rule is not configured for priority measurements.

Src IP	Src port	Dst IP	Dst Port	Proto	TCP flags	Time limit	Port	Queue
-	-	-	-	-	-	0.0 µs	-	-

Table 7.3: Priority list rules for passive idea measurements

generator. The first rules are placeholder rules that do not match the generated packets. Our ACL installer script generates these rules by incrementing the source IP address. The last rule is the matching rule. In this chapter, we use the terms "rule count" and "position of matching rule". They are equal to the sum of placeholder rules and the matching rule.

Table 7.2 shows the priority list rules we configure on the firewall for configurable time limit measurements and for priority measurements. The first rule matches the high-priority packets we generate. The second rule is a catch-all rule to limit the ACL processing time of all packets. We do not configure the catch-all rule for priority measurements because it is not supported with the priority idea enabled (due to the limitations of the NIC, see Section 6.4.3). For the measurements of the passive idea, we configure a single catch-all priority list rule with a time limit of zero (see Table 7.3).

7.2 Measurement limitations

Packets arriving at a firewall are diverse. They have a different size, content, and arrive in different intervals. We cannot cover all of these variations in our measurements in this chapter. Instead, we cover only a small subset of the possible patterns.

7.2.1 General limitations

In our measurements, we put a constant load on the firewall with a repeating pattern of packets. We assume that a constant load at a high data rate simulates the worst-case scenario of the firewall. However, we cannot simulate the sporadic arrival of

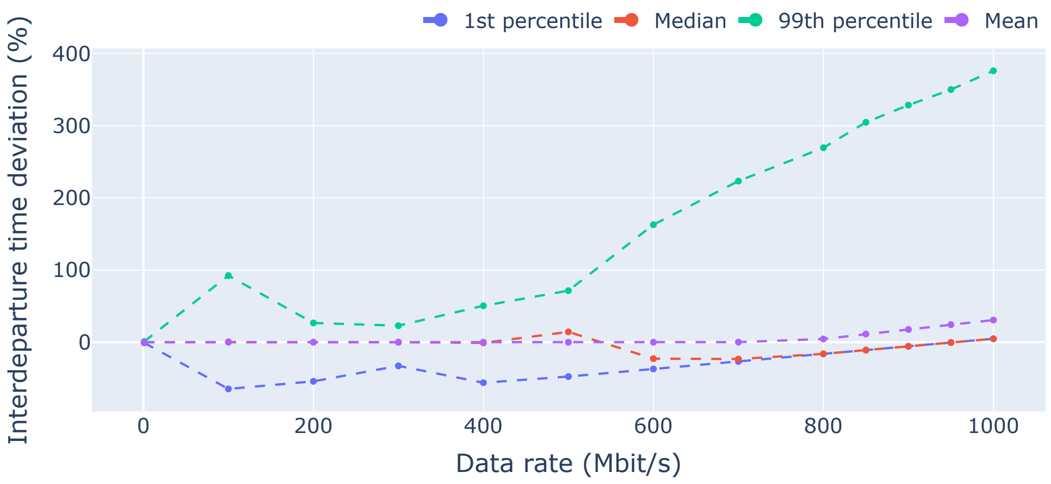


Figure 7.1: Relative deviation of the interdeparture time at the packet generator from the ideal interdeparture time (packet size: 64 B)

other packets that do not fit the pattern. These packets could change the result of the measurements, for example, if they match at a different position in the ACL. Moreover, sporadic changes in the packet arrival rate affect the adaptive time limit. In our measurements, the adaptive time limit is rather static than varying due to the constant packet arrival rate. To further analyze and improve the adaptive time limit control, we would have to vary the packet arrival rate during the measurement.

7.2.2 Reliability of the packet generator

Before evaluating our measurement results, we verify that our packet generator generates packets according to its configuration. Besides the actual size and content of the packets, the most important aspect is the interdeparture time of the packets. The interdeparture time is the time interval between the transmission of two packets. Since we generate packets at a constant data rate, the interdeparture time should be constant. In Section 2.3, we explained how the interdeparture time is calculated based on the packet size and data rate. We analyze the interdeparture time of our packet generator in the following.

Figure 7.1 shows the relative deviation of the measured interdeparture time from the calculated interdeparture time at different data rates. To show the outliers, we not only included the mean and median in the plot but also show the 1st and 99th percentile. We conducted the measurements with a packet size of 64 bytes. This way, we generate the highest possible rate of packets at a data rate of 1000 Mbit/s. The results are similar when using the same packet rate at a different packet size. We identify significant outliers in Figure 7.1 at all data rates. However, the median stays close to the calculated (ideal) interdeparture time until 600 Mbit/s. Then, the median decreases because the packet generator compensates for the increasing outliers. The compensation of the outliers works, as the average stays constant until

Data rate (Mbit/s)	Ideal (µs)	Mean (µs)	Median (µs)
1	672.000	672.004	672.000
100	6.720	6.720	6.752
200	3.360	3.360	3.360
500	1.344	1.343	1.536
600	1.120	1.119	0.864
700	0.960	0.961	0.736
800	0.840	0.878	0.704
850	0.791	0.880	0.704
900	0.747	0.878	0.704
950	0.707	0.879	0.704
1000	0.672	0.878	0.704

Table 7.4: Comparison of the calculated interdeparture time and the measured interdeparture time of the packet generator at different data rates (packet size: 64 B)

800 Mbit/s. At this point, the packet generator reaches its limits, also leading to an increasing average. Table 7.4 illustrates this (as well as the 1st percentile in Figure 7.1). Starting at 800 Mbit/s, the mean and median of the interdeparture stop decreasing. This means that the packet generator is not able to generate more packets and the data rate does not increase anymore. Thus, we consider 800 Mbit/s with 64-bytes-packets as the limit in our measurements. This corresponds to approximately 1.265 Mpps. Consequently, the packet generator can still generate, for example, packets of 100 bytes size at 1 Gbit/s ($1.265\,\text{Mpps} \cdot (100\,\text{B} + 20\,\text{B}) \approx 1.21\,\text{Gbit/s}$).

7.3 Performance optimization

In Section 6.6, we explained how we tweaked the Linux configuration to improve the performance. Figure 7.2a shows the latency during one measurement on an unoptimized system. Each marker in the plot depicts the latency of one packet (due to the large number of packets, the markers overlap and create lines). The measurement shows many outliers up to 140 µs. When we repeat the measurement on an optimized system, we get the latencies shown in Figure 7.2b. In the optimized measurement, the outliers are smaller and less frequent compared to the unoptimized measurement. On the other hand, the average latency did not change considerably. Table 7.5 shows the change in greater detail. There is almost no change in minimum, median, and latency. In contrast, the 99th percentile, 99.9th percentile, and maximum decreased by up to 60 µs.

The reason for the decrease of the outliers while the average latency stays constant is the reduction of interrupts by the kernel. Kernel interrupts lead to a short-term increase of packets in the ingress queue. When the firewall runs again, it gradually empties the queue, and normal operation continues. Outliers are a cause of jitter because the difference between the minimum and maximum latency increases. Therefore, the configuration tweaks to the Linux kernel reduce the jitter while the

Parameter	Unoptimized (µs)	Optimized (µs)	Absolute change (µs)
Mean	27.411	25.894	-1.516
Median	25.408	25.008	-0.400
Minimum	13.216	13.136	-0.080
75th percentile	27.392	26.624	-0.768
90th percentile	33.296	30.976	-2.320
99th percentile	71.840	38.928	-32.912
99.9th percentile	90.176	58.608	-31.568
Maximum	138.080	77.520	-60.560

Table 7.5: Comparison of the latency between an unoptimized and an optimized Linux system (data rate: 500 Mbit/s; packet size: 64 B; position of matching rule: 100)

average latency stays the same.

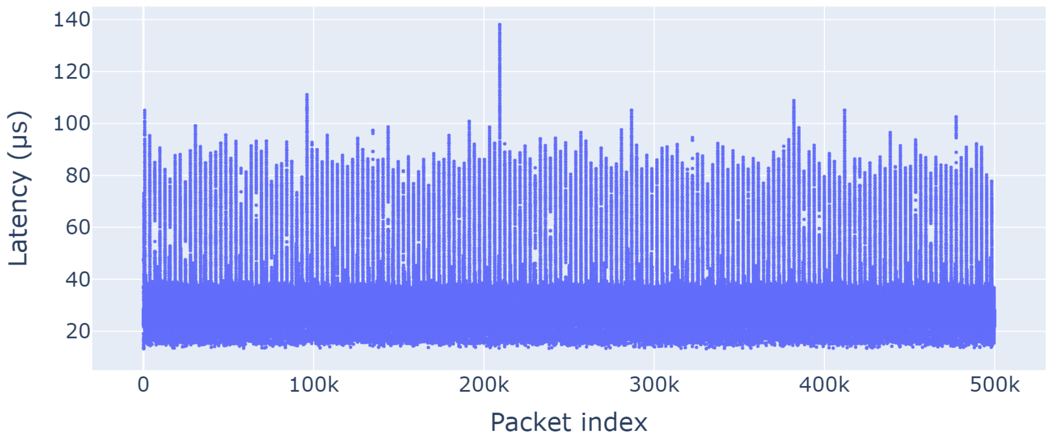
7.4 Performance impact of internal performance measurements

We implemented the internal performance measurement to receive more information about ACL processing. Many figures in this chapter rely on the internal performance measurement. In Section 5.5.4, we listed the information that we measure internally. Furthermore, we explained the implementation of the internal performance measurement in Section 6.5. While we tried to keep the impact on latency and jitter as small as possible, we cannot prevent a latency increase. In this section, we analyze the latency increase caused by the internal performance measurement. Knowing the latency increase helps us estimating the impact of the internal performance measurement on the measurements in this chapter.

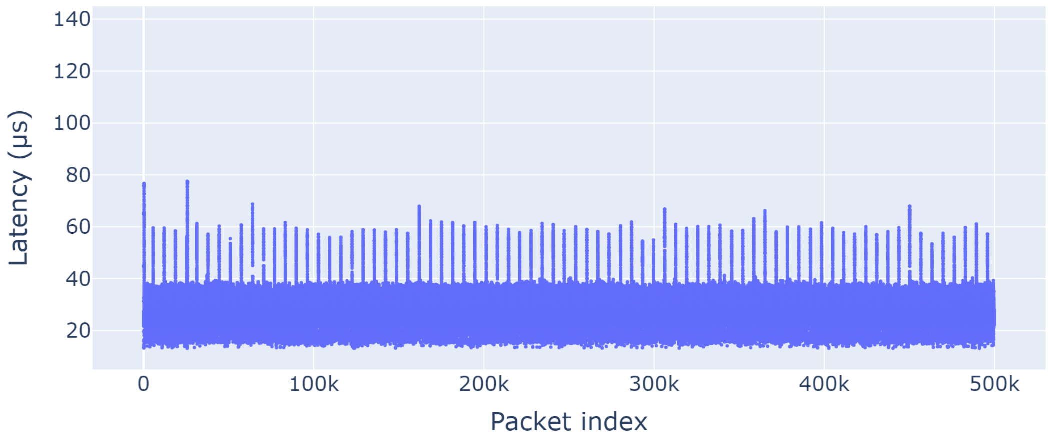
However, the impact on the latency is small. Therefore, it is challenging to separate the delay caused by internal measurements from other latency variations. In order to measure the impact of the internal performance measurement using only the external performance measurement, we repeated the measurements 100 times (one measurement encompasses 500 000 timestamps, as explained in Section 7.1.1). After that, we calculated the median latency of each measurement.

The box plots in Figure 7.3 show the distribution of these median latencies. We get a median latency of 13.760 µs for disabled measurements, and 13.904 µs for enabled measurements. Hence, the approximate delay caused by internal performance measurements is 0.144 µs. We did not notice a negative effect on the jitter of our measurements.

We performed all measurements in this chapter with internal performance measurements enabled (if not told otherwise). If the vector size is larger than one, the internal performance measurement delay multiplies by the vector size. However, measurements without ACL rules do not include the delay caused by the internal performance measurement. The reason for this is that the internal performance measurement is implemented in the ACL plugin. Without any ACL rules configured,



(a) Latency on an unoptimized Linux system



(b) Latency on an optimized Linux system

Figure 7.2: Latency on an unoptimized and on an optimized Linux system (data rate: 500 Mbit/s; packet size: 64 B; position of matching rule: 100)

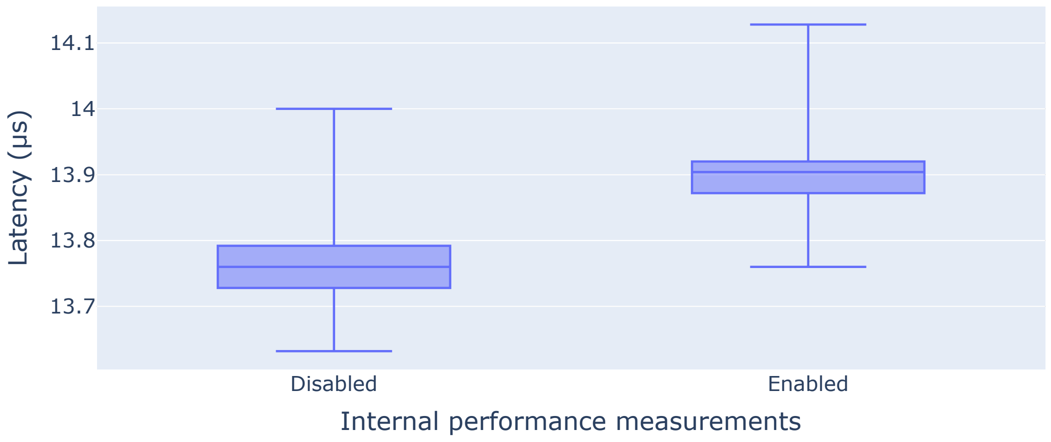


Figure 7.3: Comparison of latency when internal performance measurements are disabled or enabled (data rate: 100 Mbit/s; packet size: 100 B; position of matching rule: 50)

the ACL plugin is inactive.

7.5 Timebound

In this section, we evaluate the timebound implementation (see Section 6.2 for timebound implementation). First, we measure the overhead of ACL timekeeping. ACL timekeeping is necessary to limit the ACL processing time (see Section 6.1). After that, we evaluate whether the timebound components, the configurable time limit, and the adaptive time limit, reduce the latency and jitter.

7.5.1 Performance impact of ACL timekeeping

ACL timekeeping encompasses the tasks necessary to interrupt ACL processing when the time limit exceeds. In general, ACL timekeeping consists of two tasks. First, calculating the time point of time when the time limit exceeds. Second, checking whether the time limit exceeded before each ACL rule matching. We already explained the details about timekeeping in Section 6.1.

Due to the additional processing, ACL timekeeping increases the latency of the packets. With an increasing number of ACL rules, the timekeeping overhead further increases because each rule check involves additional timekeeping overhead. Therefore, we measured the overhead for a different number of ACL rules. We performed two measurements for each rule count. One measurement with timebound disabled, and another measurement with a static time limit (hardcoded) that never exceeds. To calculate the delay caused by ACL timekeeping, we subtracted the first measurement's mean latency from the second measurement's mean latency. We calculated

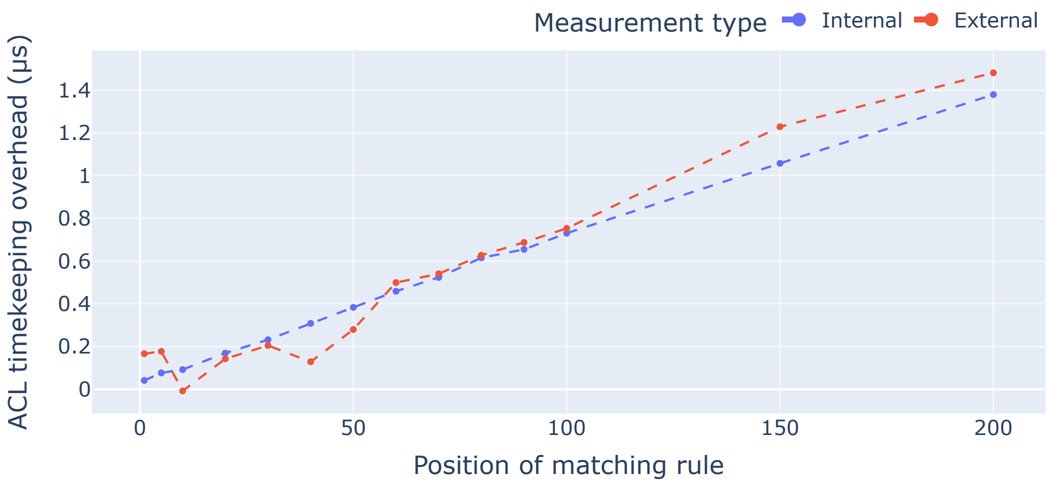


Figure 7.4: Additional delay caused by ACL timekeeping (data rate: 100 Mbit/s; packet size: 100 B)

the delay twice: using the latencies from the internal performance measurement, and using the latencies from the external performance measurement.

Figure 7.4 shows the delays we calculated. Given the delay fluctuations in the external performance measurement, we conclude that the internal measurement is more exact. There are fewer components in the internal measurement that can introduce jitter. However, given that the timekeeping delay is only in the range of a microsecond, the external measurement is pretty exact.

As written above, the timekeeping delay increases with the number of ACL rules. On average, the delay increases by around 7 ns per ACL rule. The beginning of the ACL checks introduces an additional delay because VPP must calculate the point of time when the time limit exceeds. This calculation and other organizational tasks introduce an additional delay in the range of 41.3 ns. Consequently, the overhead of ACL timekeeping is larger if VPP must only check a few rules (0 to 5).

7.5.2 Configurable time limit

The configurable time limit limits the ACL processing time so that the ingress queue does not fill up with packets. We first introduced it in Section 2.4.1, discussed its design in Section 5.2.1, and explained its implementation in Section 6.2.1.

For the measurements in this section, we generated packets of two different flows where 10 % of the packets are high-priority packets (see Section 7.1.1 load type 2). As defined in Section 7.1.2, we configured two priority list rules. One of these rules is a catch-all rule limiting ACL checks of all packets to 1 µs. We used a constant data rate of 400 Mbit/s, a packet size of 64 bytes, and varied the number of rules.

Figure 7.5a shows the initial situation without configurable time limit. The latencies are in a normal range until a rule count of 200. More rules lead to an increasing latency and to increasing packet loss. Due to the ACL checks, the packet processing

Parameter	Median
Vector size	14
Number of processed rules (non-priority)	79
Number of processed rules (high-priority)	47

Table 7.6: Configurable time limit statistics (data rate: 400 Mbit/s; packet size: 64 B; position of matching rule: 500)

takes longer than the packet interarrival time.

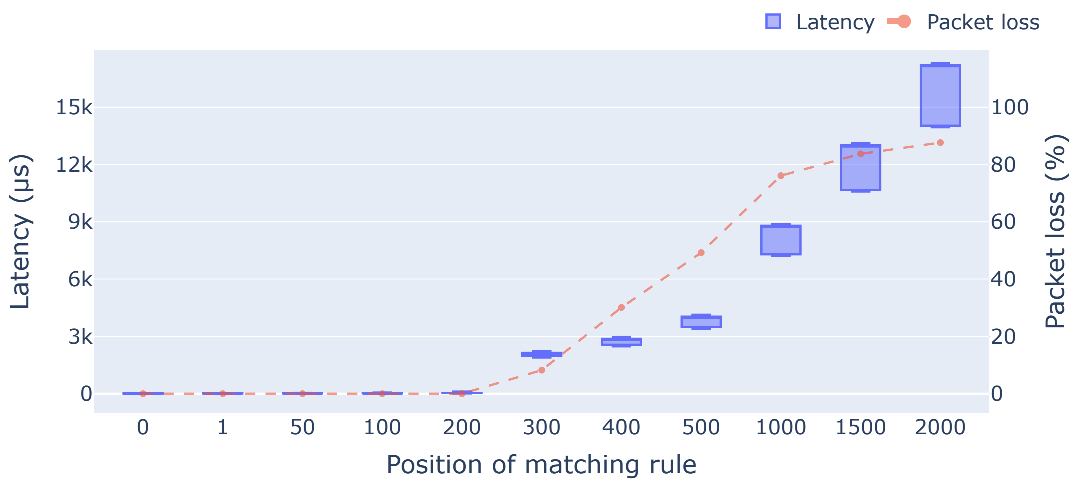
Figure 7.5b shows the same measurement with configurable time limit enabled. The catch-all rule in the priority list limits the ACL processing time, resulting in a latency decrease. As soon as the catch-all rule limits the ACL processing time, the latency and jitter stay the same regardless of the rule count. In our example, the catch-all rule starts limiting the ACL processing time at around 100 rules. Below 100 rules, the firewall does not limit the ACL processing time. However, ACL timekeeping (see Figure 7.4), priority list processing, and enqueuing the packets to analyze later increase the overhead. The size of the overhead mainly depends on the number of priority list rules.

With the configurable time limit, the median vector size settles at a value of 14 in our example (see Table 7.6). The vector size does not increase further when increasing the number of rules. It only decreases below approximately 100 rules. However, the vector size increases when increasing the data rate, as we show later. Table 7.6 also shows the number of processed rules before the firewall interrupts ACL processing. In contrast to the vector size, the number of processed rules stays the same when increasing the data rate. We see that the firewall interrupts high-priority packets earlier (47 rules median) than non-priority packets (79 rules median). This behavior corresponds to the time limit configured using the priority list. Figure 7.6 shows that the firewall interrupts the ACL check after the configured time. The measured ACL processing durations are slightly longer due to additional processing before and after the ACL check. Additionally, the processing in software causes outliers that we do not further investigate.

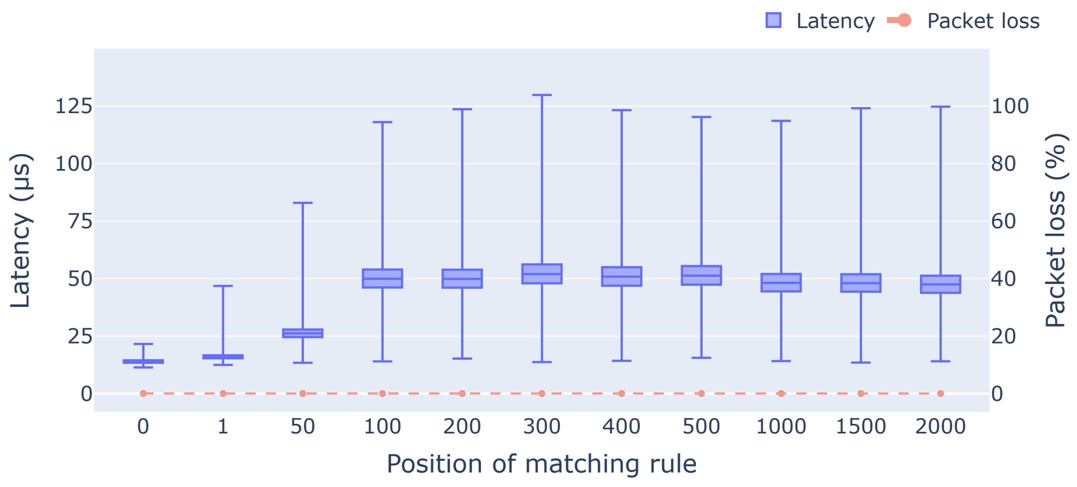
The disadvantage of the configurable time limit is the static time limit. It does not adapt to the current packet rate. For example, a time limit of 1 µs works at a data rate of 400 Mbit/s. This changes at higher data rates. For example, Figure 7.7a shows the situation at a data rate of 600 Mbit/s. Starting at around 50 ACL rules, latency and jitter increase too much, resulting in packet loss. The reason for this is that the interarrival time of the packets decreases below the packet processing time. At a constant data rate, the latency and packet loss settle. In our example, the latency settles around 7 ms, and the packet loss settles around 20 %. More rules do not further increase the latency because the configurable time limit limits the processing time.

7.5.3 Adaptive time limit

The adaptive time limit overcomes high latency, jitter, and packet loss, caused by high packet rates. It overrides the configurable time limit under high load to further



(a) Without configurable time limit



(b) With configurable time limit

Figure 7.5: Latency and packet loss for different matching positions without and with configurable time limit. Note that the y-axes have a different scaling to improve the readability of (b) (data rate: 400 Mbit/s; packet size: 64 B)

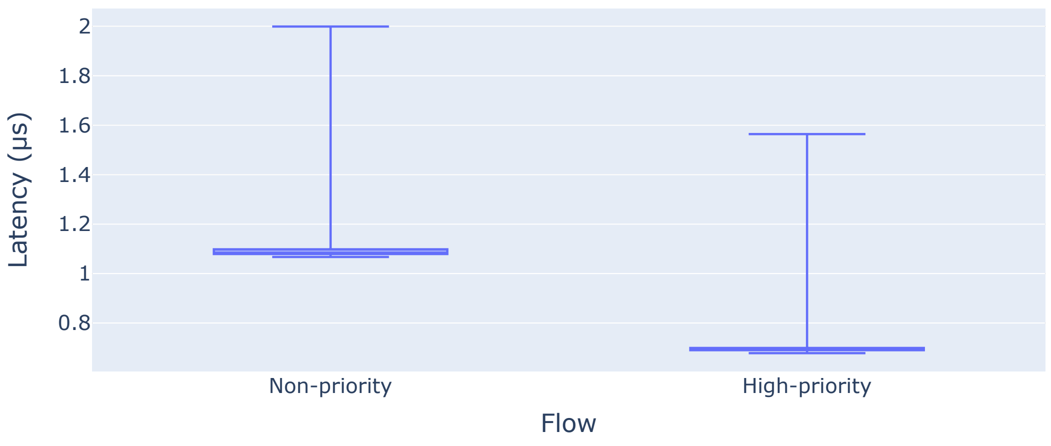


Figure 7.6: ACL processing duration split into non-priority and high-priority packets (internal performance measurement; data rate: 400 Mbit/s; packet size: 64 B; position of matching rule: 500)

decrease the ACL processing time. We first introduced the configurable time limit in Section 2.4.1, discussed its design in Section 5.2.2, and explained the implementation in Section 6.2.2.

In the last section, we discussed the limitations of the configurable time limit at the example in Figure 7.7a. The figure shows that the configurable time limit cannot prevent overload if the interarrival time of the packets is shorter than the configured time limit. Figure 7.7b shows the same scenario but with adaptive time limit enabled. We leave the configurable time limit enabled, but its effect is not visible in the measurement. With the adaptive time limit enabled, the latency and jitter are low again. Similar to the configurable time limit, the rule count does not change the latency and jitter.

Latency and jitter stay low, even when further increasing the data rate. Figure 7.8a shows the latency behavior at different data rates[1]. Figure 7.8b shows the effective time limit. The effective time limit is the minimum of the configurable time limit (usually used at low load) and the adaptive time limit (usually used at high load). In Section 5.2.2.2, we explained how the firewall calculates the adaptive time limit. We defined a range between 0.2 µs and 2 µs for the adaptive time limit. In this range, the firewall adapts the time limit in steps of 0.2 µs. We can identify these steps in Figure 7.8b. With increasing data rate, the adaptive time limit decreases down to 0.2 µs. On the other hand, we see the configurable time limit dominating at a data rate of 300 Mbit/s. In this case, the firewall interrupts most packets after 1 µs of ACL processing. After 0.6 µs, the firewall interrupts the fewer high-priority packets.

[1]Some of the following figures contain violin plots. They depict the probability density at different values (e.g., latencies). The wider the plot, the more samples were measured at a given value. If the violin plots offer enough space, we included box plots inside of them.

Figure 7.8c shows the vector size at different data rates. When increasing the data rate, the vector size increases. The reason for this is that the adaptive time limit cannot decrease below 0.2 µs. To compensate for the higher load, the firewall processes packets in larger batches. However, we notice an increasing jitter of latency and vector size at 700 Mbit/s. Eventually, the adaptive time limit control causes the jitter because the controller's parameters are not fully tuned. Since the jitter is still within an acceptable range, we did not investigate its cause further.

7.6 Analyze later

Analyze later continues ACL processing in the background. We first introduced analyze later in Section 2.4.1, discussed its design in Section 5.2.3, and explained the implementation in Section 6.2.3.

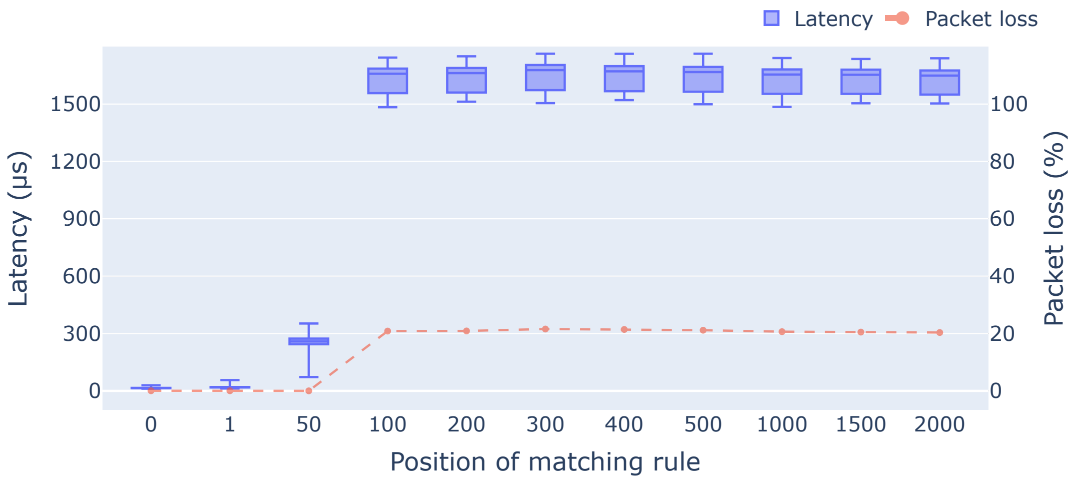
Analyze later runs on a different CPU core and locks the analyze later buffer to access it (see Section 6.2.3.2). While the analyze later buffer is locked, the ACL processing node cannot access it. To verify that the locking does not heavily increase the latency and jitter, we measured the latency with and without analyze later.

Figure 7.9 shows the result at different data rates. At all data rates shown in the figure, the 1 000 000 entries in the analyze later buffer are occupied after a few seconds under constant load. Consequently, analyze later is always busy and accesses the analyze later buffer at a high rate to fetch and delete packet information. By increasing the data rate, the ACL processing node accesses the analyze later buffer more frequently. This potentially leads to more frequent situations where the ACL processing node has to wait for the analyze later node. However, we only measure a small latency increase at higher data rates. In Figure 7.9, we can hardly see a difference. The number of outliers only increases at high data rates. The median latency increases between 1 µs and 1.5 µs.

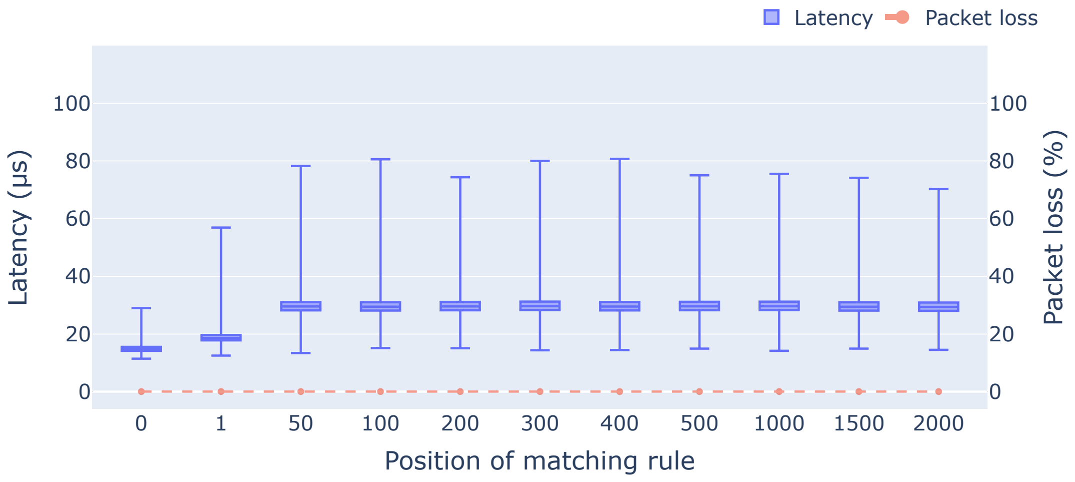
7.7 Passive

With the passive idea, the firewall forwards all packets without a prior ACL check. To detect wrong forwarding decisions at least afterward, the passive idea enqueues all packets to analyze later. We first introduced the passive idea in Section 2.4.2, discussed its design in Section 5.3, and explained the implementation in Section 6.3.

Figure 7.10a compares the latency of passive with the latency of the adaptive time limit at different data rates. We omitted a comparison at a different number of ACL rules because passive does not check any rules at all. In the last section, we concluded that the latency of the adaptive time limit is almost constant at different data rates except for some outliers. The passive idea behaves even better. Increasing the data rate leads to a smaller increase of latency and jitter compared to the adaptive time limit. In addition, the median latency is lower at all data rates in Figure 7.10a. The comparison of the vector size in Figure 7.10b also reflects the lower latency. At all data rates, the passive vector sizes are smaller than the adaptive time limit vector sizes.

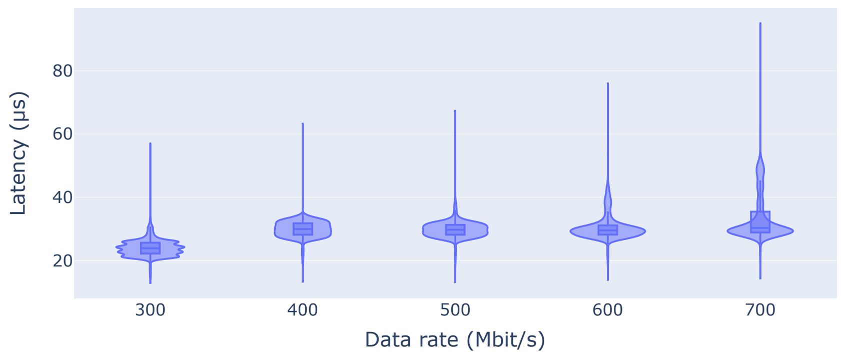


(a) Only configurable time limit

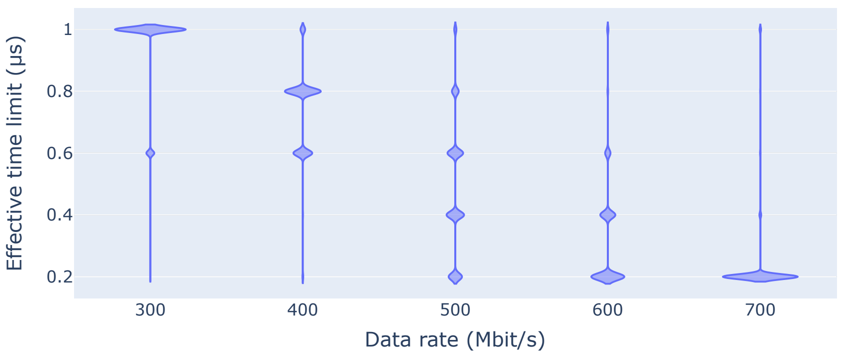


(b) Configurable time limit and adaptive time limit

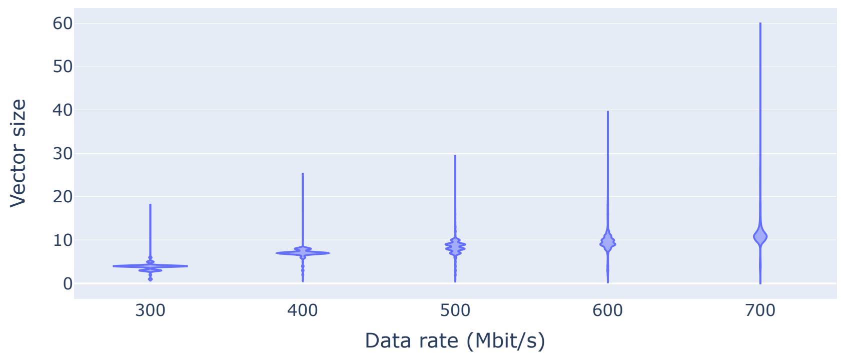
Figure 7.7: Latency and packet loss for different matching positions with configurable time limit only, and configurable time limit and adaptive time limit. Note that the y-axes have a different scaling to improve the readability of (b) (data rate: 600 Mbit/s; packet size: 64 B)



(a) Latency



(b) Effective time limit



(c) Vector size

Figure 7.8: Effective time limit, vector size, and latency for adaptive time limit in combination with configurable time limit at different data rates. (packet size: 64 B; position of matching rule: 500)

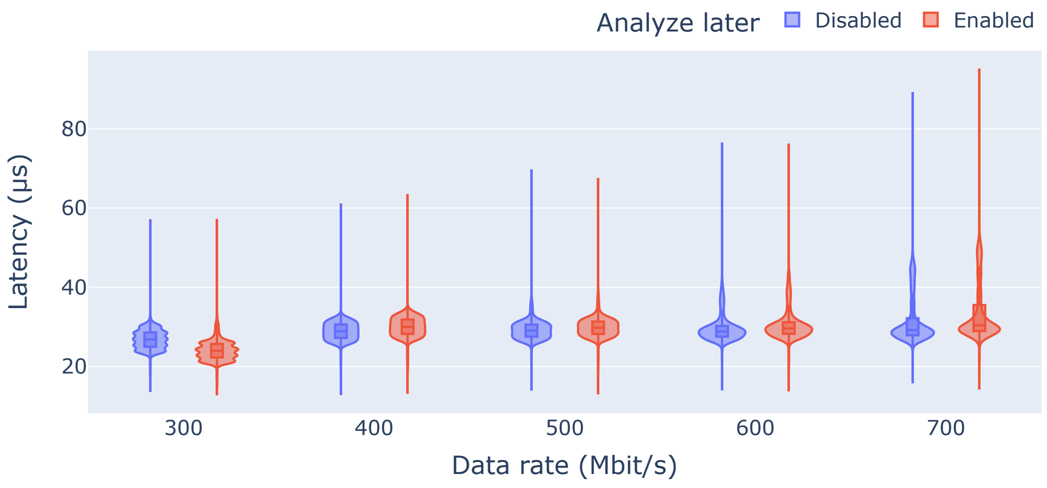


Figure 7.9: Latency for different data rates with analyze later disabled and enabled (packet size: 64 B; position of matching rule: 500)

Eventually, we can reduce the latency and jitter of the adaptive time limit by improving the time limit calculation (as we explained in the last section). In that case, we would have to check again whether the passive idea performs better. However, we think that the latency difference of the passive idea compared to the adaptive time limit and configurable time limit is small. Both are equally suitable (or unsuitable) for real-time applications. Hence, we recommend using the configurable time limit or adaptive time limit for security reasons (or a combination of them).

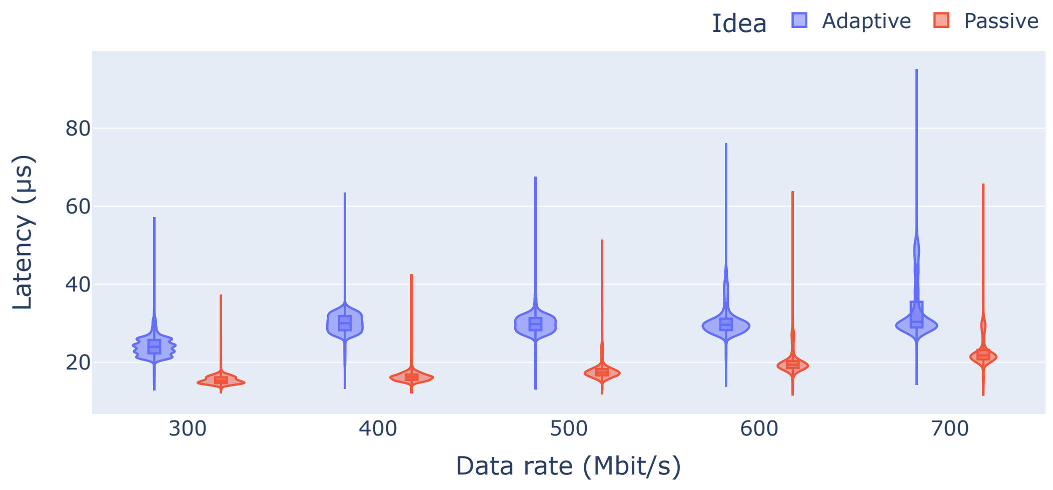
7.8 Priority

With the priority idea, the firewall processes the high-priority and non-priority packets on different CPU cores (worker threads). Hence, non-priority packet processing does not influence high-priority packet processing. We first introduced the priority idea in Section 2.4.3, discussed its design in Section 5.4, and explained the implementation in Section 6.4.

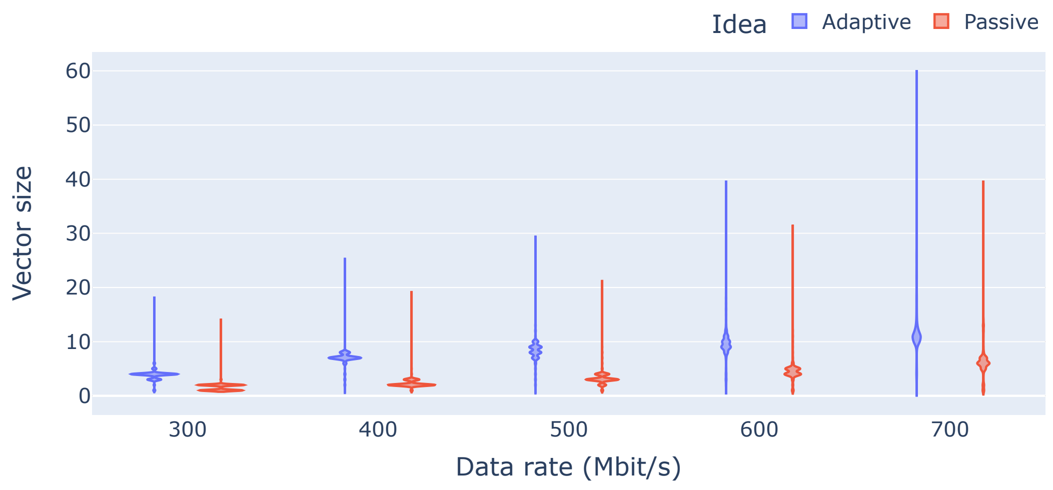
7.8.1 Latency behavior

To verify that the firewall processes high-priority packets independently of the non-priority packets, we look at the latency. Only 10 % of the packets that our packet generator generates are high-priority packets. Therefore, the load on the high-priority CPU core should be lower than on the non-priority CPU core. A lower load results in a lower latency.

Figure 7.11a shows the latency at different data rates. At all data rates, the violin plots show two horizontal peaks. The upper peak is larger than the lower peak. We claim that the lower peak (low latency) represents the high priority packets and that the upper peak (higher latency) represents the non-priority packets.



(a) Latency



(b) Vector size

Figure 7.10: Comparison of adaptive time limit in combination with configurable time limit and passive at different data rates (packet size: 64 B; position of matching rule: 500)

To further verify that the firewall processes high-priority packets independently of the non-priority packets, we look at the vector size. Figure 7.11b shows that the vector size on the high-priority CPU core is always one. This proves that the load on the high-priority CPU core is low. Furthermore, the vector size on the non-priority CPU core is comparable to the vector size of the adaptive time limit in Figure 7.8c. It is just slightly lower because only 90 % of the packets arrive on the non-priority CPU core. Thus, we claim that the non-priority CPU core processes all non-priority packets.

7.8.2 Combination with timebound

The priority idea can be used in combination with the configurable time limit and/or adaptive time limit (as we did in the above measurements). Combining both ideas has the advantage that the firewall mitigates a high load on the high-priority CPU core. However, if only a low rate of high-priority packets is expected, the priority idea can be used on its own. Many real-time applications in industrial environments only require a low data rate. Therefore, they do not require a combination of all timebound and priority.

7.9 Security considerations

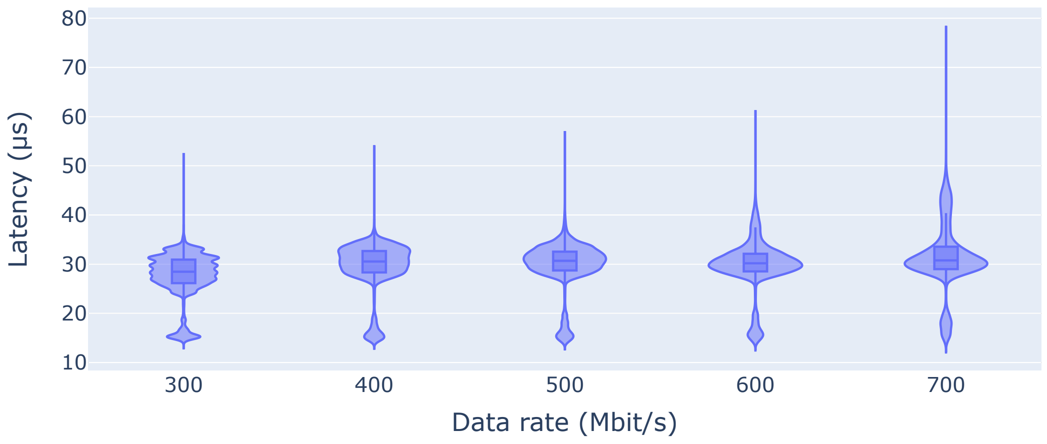
The timebound idea and the passive idea that we proposed in Section 2.4 worsen the security because they can forward packets before ACL processing finished. In contrast, the priority idea does not influence security. In this section, we discuss noteworthy security tradeoffs, attacks, and one mitigation to guarantee that the firewall checks at least a few rules.

7.9.1 Timebound and passive

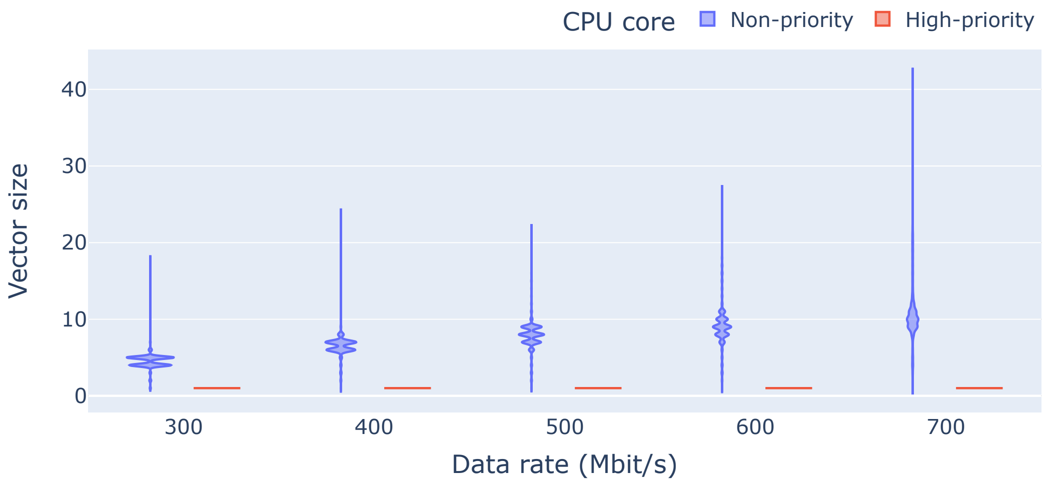
Timebound and passive worsen the security because of the time-limited ACL processing. To provide at least some security, we introduced analyze later. Analyze later logs wrong forwarding decisions for further handling by administrators.

7.9.1.1 Early interruption of ACL processing

Timebound and passive interrupt ACL processing if the time limit exceeded. While timebound checks at least some rules, passive does not check any rules at all. With the configurable and adaptive time limit, it is possible to place the most important rules at the beginning of the ACL, for example, to disallow access to certain networks. However, the number of checked rules is dependent on the applied time limit. If we configure a short time limit, the firewall checks fewer rules than with a long time limit if the firewall is under high load. Furthermore, the firewall does not guarantee how many rules it checks, even at a constant time limit. Due to software processing, the number of checked rules varies on different hardware as well as during the runtime.



(a) Latency



(b) Comparison of the vector size of the non-priority and high-priority worker threads

Figure 7.11: Latency and vector size of the priority idea at different data rates (packet size: 64 B; position of matching rule: 500)

As an example, we look at the number of checked rules at the lowest possible adaptive time limit, which is 0.2 µs. If the firewall interrupts the ACL check of all packets, the median number of checked rules is 15. The whole range of checked rules is larger. It ranges from 1 to 16 checked rules As a consequence, we cannot even be sure that the firewall checks two rules. The firewall interrupts 0.6 % of the packets after the first rule. To ensure that the firewall checks at least a certain number of rules, we can introduce a rule limit in addition to the time limit. The rule limit interrupts ACL processing after a certain number of checked rules instead of time. This improves the security but likely increases the jitter.

7.9.1.2 Analyze later buffer

In Section 5.2.3.2, we recommended to drop packets if the firewall cannot enqueue packet information into the analyze later buffer because it is full. If the firewall does not drop the packets if the buffer is full, it leaves room for attacks. For example, an attacker can send a large number of packets that are all enqueued into the analyze later buffer. If the analyze later is not sufficiently dimensioned for such a load, the analyze later buffer gets full. As a consequence, VPP forwards packets without enqueuing them into the analyze later buffer. The attacker can exploit this and send forbidden packets without the administrators noticing because analyze later does not check the packet. However, the administrators probably notice the attack due to the high firewall load or due to the full analyze later buffer.

On the other hand, dropping packets if the analyze later buffer is full also enables an attack. The attack starts the same way. An attacker sends packets that are all enqueued into the analyze later buffer. If the analyze later buffer is full, the firewall drops all packets that it cannot enqueue into the analyze later buffer. This way, the attacker can interrupt the packet transmission.

7.9.2 Priority

The priority idea does not worsen the security. It does not limit the ACL processing time but only processes packets on different CPU cores. However, if the priority idea is used in conjunction with timebound or passive, the security issues are the same as in Section 7.9.1.

8 Conclusion

State of the art software firewalls are unsuitable for use in real-time industrial environments. Due to their high jitter, long time slots must be configured on TSN network devices to ensure high-priority packets arrive within the time slot. However, long time slots decrease the available time to process other packets.

To enable real-time packet processing on software firewalls, we proposed three ideas: timebound, passive, and priority (see Section 2.4). We introduced and discussed the design of these ideas in Chapter 5. Our goal was to keep the design as generic as possible, allowing the implementation in different software firewalls. Nevertheless, parts of the design are specific to VPP, the software firewall we chose to implement the ideas. In Chapter 6, we explained the implementation of the ideas in VPP, based on our design.

During the whole design and implementation phase, we had efficiency in mind. The ideas should degrade the firewall performance only as little as possible. In Chapter 7, we evaluated whether our implementation works as expected and is suitable for real-time packet processing in industrial environments. Additionally, we analyzed how much our modifications increase latency and jitter due to the processing overhead.

8.1 Summary

We chose VPP as a software firewall to implement the three ideas. In the following, we briefly summarize the functioning of each idea.

Timebound Timebound consists of two mechanisms to limit the ACL processing time: the configurable time limit and the adaptive time limit. Both interrupt ACL processing if the time limit exceeded to forward the packet.

The configurable time limit enables the administrator to configure the time limit by itself. In contrast, the adaptive time limit automatically determines a time limit to check as many ACL rules as possible while staying below the packet interarrival time. Both, the configurable time limit and the adaptive time limit, fulfill our expectations regarding latency and jitter. The adaptive time limit is especially useful in dynamic load scenarios where the interarrival time of the packets moves below the configured time limit. In this case, the adaptive time limit still prevents an overload situation.

Additionally, we introduced analyze later that checks packets in the background if ACL processing was interrupted. This is an additional security measure to notify the administrator if the firewall forwarded a packet although it should have dropped it.

Passive With the passive idea, the firewall does not perform ACL checks at all before forwarding the packet. Instead, it just forwards the packets and performs the checks using analyze later. We implemented the passive idea by setting the configurable time limit to zero.

Priority The priority idea is different from the timebound and passive ideas. It does not interrupt ACL processing. Instead, the firewall instructs the NIC to place high-priority and non-priority packets in different queues. Then, the firewall processes the packets on different CPU cores. Processing the packets on different CPU cores prevents that non-priority packet processing slows down high-priority packet processing. The priority idea can be used in combination with timebound or passive, but it is not mandatory.

8.2 Suitability for real-time packet processing in industrial environments

Our modified firewall reaches our goals we defined in Section 2.2. It prevents overload scenarios caused by ACL processing. On the unmodified firewall, we measured a latency of up to 8 ms with a jitter of up to 1 ms at a data rate of 400 Mbit/s, a packet size of 64 B, and 1000 ACL rules. Such a high latency also involves almost 80 % packet loss. On the modified firewall, the latency settles below 40 µs with a jitter below 60 µs in situations that are worse than the above (700 Mbit/s, 1000 rules). When lowering the data rate, the jitter of the modified firewall decreases. For example, at 1 Mbit/s, the jitter is less than 3 µs at a latency of below 18 µs.

The jitter of the modified firewall is much lower than the cycle time of cyclic traffic, 2 ms to 20 ms (see Section 2.1). Thus, the firewall is suitable to process cyclic traffic. However, the jitter of 60 µs at high data rates is still high. As a result, the time slot on the network devices must be configured relatively long to ensure that the high-priority packets arrive within the time slot. Long time slots block other traffic for a long time and reduce the firewall throughput.

As already mentioned, the jitter is lower when lowering the data rate. Therefore, if the load in the network does not cross a certain data rate, a lower jitter can be assumed. Consequently, a shorter time slot can be chosen, increasing the firewall throughput for other packets. The priority idea is ideal in this case. Typically, high-priority packets make up only a small portion of the traffic. Since the firewall processes high-priority packets on a different CPU core, the load on this core stays low, as well as the jitter.

We did not measure any packet loss during our measurements of the modified firewall. Therefore, the firewall is also suitable for acyclic traffic that does not tolerate packet loss. However, we cannot guarantee that the firewall will never drop any packet because software firewalls are not fully deterministic. The same applies to the jitter. Despite we never measured any significant outliers of the jitter above, they can still occur very rarely.

In summary, we think that firewalls implementing one of the proposed ideas are suitable for real-time industrial environments regarding latency and jitter. At least, as long as the requirements on the cycle time are not too demanding and very rare outliers or packet loss are tolerable. Especially the priority idea is interesting because it does not involve security tradeoffs compared to the timebound idea and the passive idea. However, when using the priority idea without timebound or passive, the ACL should not contain too many rules (depending on the data rate).

8.3 Future work

Our ideas turned out to yield the expected improvements in latency and jitter. Therefore, we can improve and extend them to, for example, further improve latency and jitter. We identified several points of our ideas and their implementation that we can further improve and investigate.

8.3.1 Timebound and Passive

Many points for improvement apply to the timebound idea and the passive idea, including analyze later. Most of the following points apply to timebound only but some apply to both.

Allow configurable rule limit instead of time limit As a consequence of the configurable or adaptive time limit, we cannot tell anymore how many rules the firewall checks before interrupting ACL processing. Alternatively, we can introduce a rule limit that interrupts ACL processing after a configurable number of rules. This ensures that the firewall checks a minimum number of rules but worsens the latency and jitter behavior.

Extend priority list by more fields In our implementation, the priority list only matches 5-tuples. However, priorities are often configured using VLAN priorities in industrial environments. Therefore, the firewall should support matching VLAN priorities. The usage of VLAN priorities has another positive effect: the number of priority list rules decreases (there are only eight priorities). Only a few rules are needed to match high-priority packets because they can be identified by their VLAN priority. As a result, the priority list matching is faster.

Improve adaptive time limit control In its current implementation, the adaptive control of the time limit is very basic. We only use a P controller in our implementation (see Section 5.2.2.2). In future, we can further extend it to a PID controller to improve the control in case of fluctuating packet rates. With an improved controller, we can eventually further reduce the jitter when using the adaptive time limit, as we suppose in Section 7.5.3.

Scale analyze later Some firewalls do not offer enough processing power to run analyze later on the same device. To support such firewalls, we can offload analyze later to another device. We already discussed this idea in Section 5.2.3.1.

Alternatively, we can scale analyze later on the same device if the firewall has enough CPU cores (this is probably not the case on low-budget firewalls). In the current implementation, analyze later shares its resources with other organizational tasks on the main thread (e.g., the CLI and API). As a first step, we can assign analyze later to a dedicated CPU core. In another step, we can run analyze later on multiple CPU cores. However, we think that scaling on multiple CPU cores is a rare use case. It is only useful if the firewall processes a large number of packets with a large number of ACL rules at the same time.

8.3.2 Priority

In the following, we discuss possible enhancements of the priority idea to make it more flexible and efficient.

Split ACLs The firewall does not differentiate between ACL rules that match high-priority packets and ACL rules that match non-priority packets. Consequently, the high-priority CPU core also matches non-priority ACL rules, even if it never processes non-priority packets. We can reduce the number of rules that need to be matched by splitting an ACL into a high-priority ACL and a non-priority ACL. Both ACLs only contain one type of rules: rules that match high-priority packets or rules that match non-priority packets, respectively.

Extend hardware classification We only implemented a basic version of hardware classification to place high-priority packets in different ingress queues. Thus, the NIC can only partly match a 5-tuple (see Section 6.4.3). Most NICs supported by DPDK can match at least VLAN tags. Thus, we can improve the suitability for industrial environments by implementing VLAN priority classification on the NIC.

8.3.3 Other improvements

Future work is not limited to improving our ideas and their implementation. There are other interesting areas that improve the latency and the overall functionality.

Further reduction of the latency We already achieve low latency and jitter at low data rates. However, the latency is still quite high. For example, we measure a median latency of around 16 µs at a data rate of 1 Mbit/s. On the other hand, Stylianopoulos *et al.* [35] achieved a latency of around 5 µs using DPDK. We can investigate investigate the reason for the latency difference.

Stateful matching In this thesis, we did not consider stateful connections. As soon as the firewall stores a connection of a packet in the connection table, following packets of the same connection bypass the ACL check. The firewall only matches the first packet that belongs to a connection using the ACL. If the firewall interrupts ACL processing of the first packet to forward it after the time limit exceeded, the firewall also forwards all further packets without ACL check due to the connection table entry. This is a security issue and we should also consider stateful packets in our design.

Acronyms

ACL	Access Control List
ASIC	Application-Specific Integrated Circuit
DDoS	Distributed Denial of Service
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DPI	Deep Packet Inspection
DUT	Device Under Test
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IOMMU	I/O Memory Management Unit
IPsec	Internet Protocol Security
NAT	Network Address Translation
NIC	Network Interface Controller
OS	Operating System
RSS	Receive Side Scaling
SIMD	Single Instruction, Multiple Data
tc	traffic control
TSN	Time-Sensitive Networking
URL	Uniform Resource Locator
XDP	eXpress Data Path

Bibliography

[1] L. Wüsteney, M. Menth, R. Hummen, and T. Heer, "Impact of Packet Filtering on Time-Sensitive Networking Traffic," in *2021 17th IEEE International Conference on Factory Communication Systems (WFCS)*, 2021, pp. 59–66. DOI: `10.1109/WFCS46889.2021.9483611`.

[2] R. Belliardi *et al.* "Use Cases IEC/IEEE 60802," IEEE. (Sep. 13, 2018), [Online]. Available: `https://www.ieee802.org/1/files/public/docs2018/60802-industrial-use-cases-0918-v13.pdf` (visited on 03/14/2022).

[3] W. Noonan and I. Dubrawsky, *Firewall Fundamentals*. Cisco Press, 2006, ISBN: 1-58705-221-0.

[4] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 6th. Pearson, 2012, ISBN: 0-13-285620-4.

[5] J. M. Kizza, *Guide to Computer Network Security*, 5th. Springer, 2020, ISBN: 978-3-030-38141-7. DOI: `10.1007/978-3-030-38141-7`.

[6] J. Wang and Z. A. Kissel, *Introduction to Network Security: Theory and Practice*, 2nd. Wiley, 2015, ISBN: 978-1-118-93948-2.

[7] "Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model," ISO/IEC, Genève, CH, Standard ISO/IEC 7498-1:1994, Nov. 1994. [Online]. Available: `https://www.iso.org/standard/20269.html`.

[8] R. T. Braden, *Requirements for Internet Hosts - Communication Layers*, RFC 1122, Oct. 1989. DOI: `10.17487/RFC1122`. [Online]. Available: `https://rfc-editor.org/rfc/rfc1122.txt`.

[9] M. D. Leech, *SOCKS Protocol Version 5*, RFC 1928, Mar. 1996. DOI: `10.17487/RFC1928`. [Online]. Available: `https://rfc-editor.org/rfc/rfc1928.txt`.

[10] R. Rosen, *Linux Kernel Networking: Implementation and Theory*, 1st. Apress, 2014, ISBN: 978-1-4302-6197-1. DOI: `10.1007/978-1-4302-6197-1`.

[11] J. García-Dorado, F. Mata, J. Ramos, P. Santiago del Río, V. Moreno, and J. Aracil, "High-Performance Network Traffic Processing Systems Using Commodity Hardware," in *Data Traffic Monitoring and Analysis: From Measurement, Classification, and Anomaly Detection to Quality of Experience*. Springer Berlin Heidelberg, Jan. 2013, pp. 3–27, ISBN: 978-3-642-36784-7. DOI: `10.1007/978-3-642-36784-7_1`.

[12] "netfilter/iptables project homepage - The netfilter.org "iptables" project," Netfilter. (2021), [Online]. Available: `https://www.netfilter.org/projects/iptables/index.html` (visited on 03/09/2022).

[13] "netfilter/iptables project homepage - The netfilter.org "nftables" project," Netfilter. (2021), [Online]. Available: `https://www.netfilter.org/projects/nftables/index.html` (visited on 03/09/2022).

[14] "eBPF," Linux Foundation. (2021), [Online]. Available: `https://ebpf.io/` (visited on 03/09/2022).

[15] "eBPF Documentation - What is eBPF?" Linux Foundation. (2021), [Online]. Available: `https://ebpf.io/what-is-ebpf` (visited on 03/09/2022).

[16] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance Implications of Packet Filtering with Linux eBPF," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, 2018, pp. 209–217. DOI: `10.1109/ITC30.2018.00039`.

[17] S. Miano, M. Bertrone, F. Risso, M. Vásquez Bernal, Y. Lu, and J. Pi, "Securing Linux with a Faster and Scalable Iptables," *ACM SIGCOMM Computer Communication Review*, vol. 49, Jul. 2019. DOI: `10.1145/3371927.3371929`.

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

[19] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015, pp. 5–16. DOI: `10.1109/ANCS.2015.7110116`.

[20] "Ssupported hardware," DPDK Project. (2022), [Online]. Available: `https://core.dpdk.org/supported/` (visited on 02/23/2022).

[21] "Generic flow API (rte_flow)," GitHub. (2022), [Online]. Available: `https://github.com/DPDK/dpdk/blob/ee05a93e1e6633d0fdec409faf09f12a2e05b991/doc/guides/prog_guide/rte_flow.rst` (visited on 02/23/2022).

[22] "igb_flow.c," GitHub. (2021), [Online]. Available: `https://github.com/DPDK/dpdk/blob/v21.05/drivers/net/e1000/igb_flow.c` (visited on 02/23/2022).

[23] *Intel® Ethernet Controller I210 – Datasheet*, Revision Number: 3.7, Intel Corporation, Jan. 2021.

[24] "VPP source code Version 21.10," FD.io. (2021), [Online]. Available: `https://gerrit.fd.io/r/gitweb?p=vpp.git;a=tag;h=refs%2Ftags%2Fv21.10` (visited on 02/08/2022).

[25] "VPP source code Version 21.10 (GitHub mirror)," GitHub. (2021), [Online]. Available: `https://github.com/FDio/vpp/tree/v21.10` (visited on 02/08/2022).

[26] "FDio - The Universal Dataplane," FD.io. (2022), [Online]. Available: `https://fd.io/` (visited on 02/08/2022).

[27] The linux foundation, "FD.io - Vector Packet Processing - One Terabit Software Router," White Paper, Jul. 1, 2017. [Online]. Available: `https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf` (visited on 02/08/2022).

[28] "Use Cases For FD.io," FD.io. (2022), [Online]. Available: `https://fd.io/overview/usecases/` (visited on 02/08/2022).

[29] A. Orel. "Network Function Virtualization (NFV) using IOS-XR," Cisco. (Jan. 2020), [Online]. Available: `https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2020/pdf/BRKSPG-2724.pdf` (visited on 02/08/2022).

[30] "VPP Technology," FD.io. (2022), [Online]. Available: `https://fd.io/gettingstarted/technology/` (visited on 02/11/2022).

[31] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-Speed Software Data Plane via Vectorized Packet Processing," *IEEE Communications Magazine*, vol. 56, no. 12, pp. 97–103, 2018. DOI: `10.1109/MCOM.2018.1800069`.

[32] "Get Involved With FD.io," FD.io. (2022), [Online]. Available: `https://fd.io/getinvolved/community/` (visited on 02/08/2022).

[33] "Access Control Lists with VPP," GitHub. (2021), [Online]. Available: `https://github.com/FDio/vpp/blob/cdaf0d8c884ae0f337ef94b0ceb7449c991a3e6c/docs/usecases/acls.rst` (visited on 02/15/2022).

[34] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015, pp. 29–38. DOI: `10.1109/ANCS.2015.7110118`.

[35] C. Stylianopoulos *et al.*, "On the Performance of Commodity Hardware for Low Latency and Low Jitter Packet Processing," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20, Montreal, Quebec, Canada: Association for Computing Machinery, 2020, pp. 177–182, ISBN: 9781450380287. DOI: `10.1145/3401025.3403591`. [Online]. Available: `https://doi.org/10.1145/3401025.3403591`.

[36] P. Emmerich *et al.*, "Optimizing latency and cpu load in packet processing systems," in *2015 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2015, pp. 1–8. DOI: `10.1109/SPECTS.2015.7285275`.

[37] M. Cereia, I. C. Bertolotti, L. Durante, and A. Valenzano, "Latency evaluation of a firewall for industrial networks based on the Tofino Industrial Security Solution," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–8. DOI: `10.1109/ETFA.2014.7005177`.

[38] M. Cheminod, L. Durante, A. Valenzano, and C. Zunino, "Performance impact of commercial industrial firewalls on networked control systems," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–8. DOI: `10.1109/ETFA.2016.7733576`.

[39] M. Cheminod, L. Durante, L. Seno, and A. Valenzano, "Performance Evaluation and Modeling of an Industrial Application-Layer Firewall," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 5, pp. 2159–2170, 2018. DOI: `10.1109/TII.2018.2802903`.

[40] D. Zvabva, P. Zavarsky, S. Butakov, and J. Luswata, "Evaluation of Industrial Firewall Performance Issues in Automation and Control Networks," in *2018 29th Biennial Symposium on Communications (BSC)*, 2018, pp. 1–5. DOI: `10.1109/BSC.2018.8494696`.

[41] M. D. Pesé, K. Schmidt, and H. Zweck, "Hardware/Software Co-Design of an Automotive Embedded Firewall," in *WCX™ 17: SAE World Congress Experience*, SAE International, Mar. 2017. DOI: `https://doi.org/10.4271/2017-01-1659`. [Online]. Available: `https://doi.org/10.4271/2017-01-1659`.

[42] "bpf-iptables," GitHub. (2019), [Online]. Available: `https://github.com/mbertrone/bpf-iptables` (visited on 03/10/2022).

[43] "CSIT 2n-skx-xxv710," FD.io. (2021), [Online]. Available: `https://docs.fd.io/csit/rls2101/report/vpp_performance_tests/packet_throughput_graphs/ip4-2n-skx-xxv710.html#b-2t1c-features-ip4routing-base-avf`.

[44] D. Su, Y. Jiang, and W. Wang. "Towards Low Latency Interrupt Mode DPDK," DPDK Project, Linux Foundation. (Jun. 2017), [Online]. Available: `https://www.dpdk.org/wp-content/uploads/sites/35/2018/06/DPDK-China2017-JiangWang-Low-Latency-PMD.pdf` (visited on 02/03/2022).

[45] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[46] "MoonGen," GitHub. (2022), [Online]. Available: `https://github.com/emmericp/MoonGen/`.

[47] "VPPINFRA (Infrastructure)," GitHub. (2021), [Online]. Available: `https://github.com/FDio/vpp/blob/8ccc6b350703d3390633636d2b1c2f578f37cb21/docs/developer/corearchitecture/infrastructure.rst` (visited on 02/17/2022).

[48] "VPP/How To Optimize Performance (System Tuning)," FD.io. (2022), [Online]. Available: `https://wiki.fd.io/view/VPP/How_To_Optimize_Performance_(System_Tuning)` (visited on 02/24/2022).

[49] "kernel-parameters.txt," GitHub. (2022), [Online]. Available: `https://github.com/torvalds/linux/blob/23d04328444a8fa0ca060c5e532220dac8e8bc26/Documentation/admin-guide/kernel-parameters.txt`.

[50] "kernel/time: Enable NO_HZ_FULL," Debian. (2021), [Online]. Available: `https://salsa.debian.org/kernel-team/linux/-/commit/f6aad27f05c007d6f30b34ff77bc7ea47844f117`.

[51] "Linux Drivers," GitHub. (2021), [Online]. Available: `https://github.com/DPDK/dpdk/blob/ee05a93e1e6633d0fdec409faf09f12a2e05b991/doc/guides/linux_gsg/linux_drivers.rst` (visited on 02/24/2022).

[52] "Intel® Turbo Boost Technology 2.0: Higher Performance When You Need It Most," Intel Corporation. (2022), [Online]. Available: `https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`.

[53] "Test Environment," FD.io. (2021), [Online]. Available: `https://docs.fd.io/csit/rls2106/report/vpp_performance_tests/test_environment.html` (visited on 02/24/2022).

List of Figures

List of Tables