The Path of a Packet Through the Linux Kernel

Alexander Stephan, Lars Wüstrich*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: alexander.stephan@tum.de, wuestrich@net.in.tum.de

***Abstract*—Networking stacks are the backbone of communication and information exchange. This paper investigates the TCP/IPv4 and UDP/IPv4 network stack of Linux, the most common server OS. We describe a trace of the most critical networking functions of the Linux kernel 5.10.8. Although Linux networking code documentation exists, it is often outdated or only covers specific aspects like the IP or TCP layer. We address this holistically, covering a packet's egress and ingress path through the Linux networking stack. Moreover, we highlight intricacies of the implementation and present how the Linux kernel realizes networking protocols. Our paper can serve as a basis for performance optimizations, security analysis, network observability, or debugging.**

***Index Terms*—linux kernel, network stack, packet processing**

1. Introduction

Nowadays, almost everything is networked, from a personal computer to a fridge [1]. Although networking is essential for modern computing, few know the complexity of getting a packet to and from a wire. Given the prevalence of Linux-based servers [2], [3], it is common for packets to traverse through the Linux network stack. However, understanding the intricacies of the complex packet processing within Linux takes time and effort. Nevertheless, this knowledge is often critical as it aids performance optimizations, security analysis, debugging, and network observability.

We base our investigation of the ingress and egress packet path on version 5.10.8 of the Linux kernel[1]. It is well-documented, stable, and contains modern features such as a Just-in-time (JIT) compiler for Berkely Packet Filters [4]. Primarily, we make observations on the kernel source code, which we link to referenced kernel symbols.

Although Linux kernel networking is becoming more diverse, e. g., with the addition of Multipath TCP [5], most traffic utilizes the standard TCP and UDP protocol stack. Moreover, despite the acceleration in IPv6 adoption, most devices still communicate over IPv4 [6]. Hence, we limit this analysis to TCP/IPv4 and UDP/IPv4.

The remainder of the paper has the following structure: Firstly, we compare this paper with existing literature in Section 2. Then, in Section 3, we explain the design of the general Linux networking stack and the `sk_buff` data structure. In Section 4, we inspect the intricacies of both the ingress and egress packet paths. Finally, in Section 5, we briefly summarize the most important findings.

1. https://elixir.bootlin.com/linux/v5.10.8/source

2. Related Work

We evaluated literature on the Linux network stack to the best of our knowledge. While doing so, we made the following observations.

Outdated Linux kernel versions. More elaborate papers emerged in the 2000s, using Linux kernel version 2 or 3 [7]–[9]. Although the implementation of older protocols in the network stack is stable, much time has passed. Therefore, we investigate possible deviations.

Fragmented Information. Many papers focus on specific layers, most commonly the TCP and IP implementation [10]–[13]. Others determine the causes of network overhead [14], [15]. A holistic view is lacking in those cases. In particular, even when authors describe the path of a packet throughout multiple layers [7]–[10], they omit UDP—in contrast to this paper.

Although there is a talk covering the whole ingress and egress path for Linux version 5 [16], it is high-level, mainly giving an intuition. Hence, we aim for a middle ground between detailed layer-specific information and high-level network stack tracing.

3. Background

We assume a basic familiarity with Linux and networking. However, we briefly describe essential Linux networking concepts relevant throughout the packet path.

3.1. Linux Networking Stack

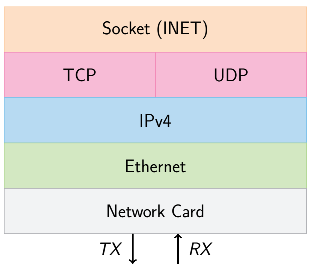

Figure 1: Depiction of the technologies used in the standard TCP/IP and UDP/IP stack in Linux, from user space to the wire.

As shown in Figure 1, a socket either passes a packet to the user space application or receives a packet from the implementation of the transport layer protocol, i. e., TCP or UDP. The IP layer then routes the packets to the network layer. Below this layer, Linux allows filtering

 doi: 10.2313/NET-2024-04-1_16

traffic via firewall rules. The network interface card (NIC) forwards the packets that it receives from the receive (RX) buffer to the kernel and transmits packets read from the transmit (TX) buffer.

3.2. Socket Buffers (`sk_buff`)

The kernel saves packets in C structures called `sk_buff`. Almost all functions along the packet path interact with it. `sk_buff` tracks packet metadata and maintains a start and end pointer to packet data in memory [17]. Using references to packet data allows for efficient packet modification by adjusting the pointers, e. g., when stripping a header away. Furthermore, `sk_buff` structures can be shared efficiently between different processes using memory references [17]. Consequently, cloning a packet is also efficient since only the metadata has to be copied [17], assuming a read-only workload. We show this in Figure 2. These properties of `sk_buff` form the basis of efficient packet processing on Linux.

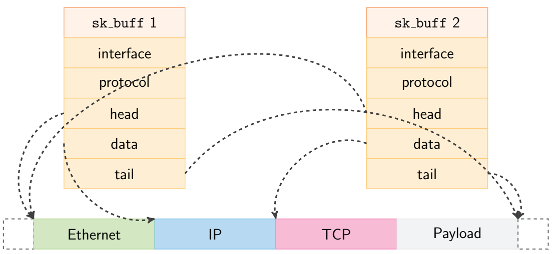

Figure 2: Two simplified `sk_buff` structures point to different locations within the same packet buffer. `head` marks the padded start of the buffer while `tail` points to the end of the actual packet data. `data` points to the currently processed header.

4. Packet Flow

Here, we are interested in both the ingress and egress paths. Both paths operate independently.

4.1. Egress Path

Firstly, we analyze the egress path, i. e., how Linux sends packets—from a user space application to the NIC as shown in Figure 3. Essentially, the egress side constructs the protocol headers, pushing them to `sk_buff` structures, which it sends out.

4.1.1. Socket Layer. All starts with a socket that has an associated domain, e. g., `AF_UNIX`, `AF_XDP`, or, in our case, `AF_INET` for IPv4. A system call wrapper function like `write()` or `sendto()` enables us to send data over the socket, e. g., as provided by the GNU C library [18]. In the context of this paper, we choose `write(filedescriptor, buffer, length)` to avoid unnecessary complexity. Writing to a file descriptor is a prime example of the UNIX philosophy *Everything is a file* since a file descriptor abstracts the socket [19].

For sockets, `write()` invokes the `sock_sendmsg()` function. It obtains the socket `struct sock` from the

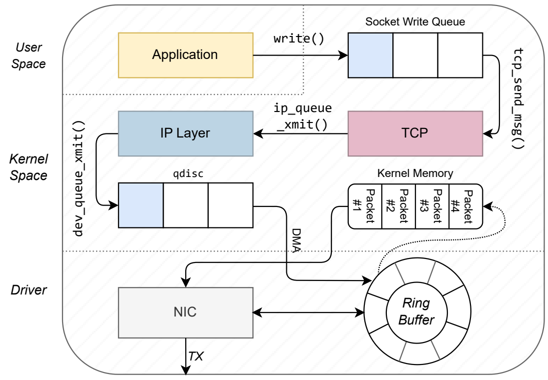

Figure 3: Egress path of a packet in case of TCP as described in Section 4.1 (adopted from [10]).

file descriptor provided by the user space application. Generally, sockets operate on socket control messages containing the process's Process ID (PID), User ID (UID), and Group ID (GID) [19]. `sock_sendmsg()` retrieves this control message from the `task_struct`, a Linux data structure that contains this information for the calling process. With this information, `sock_sendmsg()` typically passes the packet through Linux Security Modules (e. g., SELinux) to filter traffic.

Finally, it calls the corresponding transport layer handler, in our case TCP or UDP, via the macro `INDIRECT_CALL_INET`. The macro autonomously chooses the corresponding IPv4 or IPv6 variant of the transport protocol entry function, depending on the protocol specified in `sk_prot`, a field of the `sk_buff`.

4.1.2. Transport Layer. Here, we arrive at the IPv4 related entry functions, `tcp_sendmsg()` for TCP and `udp_sendmsg()` for UDP.

TCP. `tcp_sendmsg()` first waits for TCP connection establishment. Then, it allocates `sk_buff` structures for the segments and enqueues them to the socket write queue, as shown in Figure 3. `tcp_sendmsg()` also guarantees adherence to the Maximum Segment Size (MSS). After processing the queue, the kernel invokes `tcp_write_queue_tail()`. It also builds the TCP header and pushes the data from the user space into the `sk_buff`. If the data fits into the existing buffer, `skb_add_data_nocache()` is used. Otherwise, it creates new buffers, which is more expensive. It then sets the `transport_header` pointer to the beginning of this header. Next, it builds the network layer protocol header as specified in the socket options, e. g., IPv4 for `AF_INET`. `tcp_write_xmit()` guarantees that the kernel holds back data in case of congestion control restrictions. It also sets retransmission timers, i. e., resends the packet if it does not receive an `ACK` in time. Finally, `tcp_transmit_skb()` reads the write queue containing previously constructed segments and passes them to the network layer via the `queue_xmit()` function specified in the socket.

UDP. Similarly, there is `udp_sendmsg()`. Again, the function writes to the socket write queue. Next, the function waits until there are no pending frames for the UDP datagram. As before, the function builds the header, setting the destination port and the other fields.

 doi: 10.2313/NET-2024-04-1_16

There are corking and non-corking cases: corking describes waiting for frames to batch multiple UDP datagrams. Non-corking implies building `sk_buff` directly. After constructing the datagram, `ip_route_output_flow()` routes the packet and builds the network layer protocol header. Lastly, `ip_append_data()` creates an IP packet that combines multiple UDP datagrams. Overall, simplicity and absence of locking endorse that the UDP implementation is more performant than its TCP counterpart.

4.1.3. IP Layer.

IP processing starts with the function `__ip_queue_xmit()`. Firstly, the function determines the route to the destination. If a route is already in `sk_buff->_skb_refdst`, it skips the routing process. In this case, the function builds the header immediately. However, if there is no destination, the routing process continues. It determines the destination from the socket field of the `sk_buff`, that, e. g., is set if the socket previously received an IP packet. If this is not possible, it queries the routing cache, called Forwarding Information Base (FIB)—a table that is generated from the IP routing table. Eventually, if there is still no route, it returns *host unreachable* and stops the processing. Otherwise, the kernel builds the IP header if it finds a route.

Now, `ip_options_build()` is called to set IP options. It marks the beginning of the header with the `network_header` field of the `sk_buff`. Next, it triggers the `LOCAL_OUT` stage of the Linux firewall mechanism `netfilter`. Afterward, `dst_output()` calls the actual routing function via a function pointer.

Then, the kernel calls the `ip_output()` routing function for the most common unicast packet. As the routing is complete, this stage is called `POST_ROUTING`. It updates the packet metadata and calls the `NF_INET_POST_ROUTING` hook. It sets `sk_buff` metadata and invokes netfilter once again. Furthermore, it fragments the packet if it exceeds the maximum length (Maximum Transmission Unit).

Then, after passing the packet through the `NF_INET_LOCAL_OUT` hook, `ip_output()` calls `ip_finish_output()`. It increments the counters for multicast and broadcast packets. It also checks that the `sk_buff` has enough space for the MAC header. The destination MAC address is either cached or determined by the neighbor output function `neigh_resolve_output()`. The latter utilizes the Address Resolution Protocol (ARP) [20]. In case there is no ARP reply, it queues the packet again. After obtaining the MAC address, the kernel constructs the Ethernet header, adding it to the `sk_buff`.

4.1.4. Ethernet Layer.

Firstly, `dev_queue_xmit()` sets the `mac_header` field in the `sk_buff`, which is then passed to `tc_egress()`. It queues the packet in the queueing discipline (`qdisc`) [21]. As long as the NIC buffer is filled, `__qdisc_run()` dequeues the packets from the buffer. After some post-processing in `validate_xmit_skb()`, e. g., calculating the Ethernet checksum or adding VLAN tags, the kernel calls `ndo_start_xmit`, and consequently, adds the packet to the TX ring of the NIC. Eventually, the NIC's queue may be full. In this case, the kernel stops the `qdisc` [21] and queues `sk_buff`. Finally, it maps the packet to a fixed location in memory for Direct Memory Access (DMA) after adding more `sk_buff` metadata. `dev_direct_xmit` allows circumventing the `qdisc` [21], directly writing the packet to the TX ring of the NIC. eXpress Data Path (XDP) [22] is a use case of this. Eventually, the function notifies the NIC via an interrupt to end the processing and frees the `sk_buff`.

4.2. Ingress Path

Now, we trace the path of a packet that arrives at the NIC until a user space application reads it through a socket, see Figure 4. Most notably, it analyzes the headers to determine the following function call and strips them.

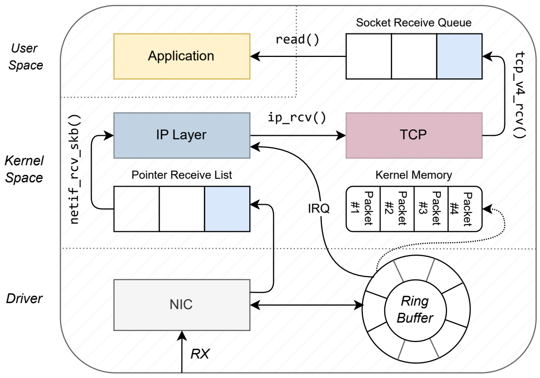

Figure 4: Ingress path of a packet in case of TCP as described in Section 4.2 (adopted from [10]).

4.2.1. Ethernet Layer.

After verifying and optionally zeroing the Ethernet checksum and applying a MAC address filter, the NIC copies the packet to the system's memory via DMA. Then, it notifies the operating system via an interrupt and indicates the location of the packet data. With this, the operating system can allocate an `sk_buff`. Now, the kernel inserts metadata into the `sk_buff`, like the `protocol` field (Ethernet), the receiving `interface`, and the `packet` type, in our case, IP.

At this stage, the kernel knows the start of the Ethernet header, so it sets the `mac_header` field to the beginning of the `sk_buff`. Finally, it removes the Ethernet header from the `sk_buff` before it passes it further up the network stack. Next, the packet arrives in `netif_receive_skb()`. The function clones `sk_buff` and forwards it to the virtual TAP interface. The TAP interface enables communication between Virtual Machines (VMs) and the host within the same network. Another important case here is forwarding VLAN-tagged packets to the VLAN interface. Furthermore, when the interface has a physical master, i. e., it is a virtual interface or part of a network bridge, `rx_handler()` steals the packet. `rx_handler()` also sets the `network_header` field of the `sk_buff`. Finally, it calls the IPv4 protocol handler function `ip_rcv()`.

4.2.2. IP Layer.

The Ethernet layer passes the packet to the IP layer via the function `ip_rcv()`. Again, `ip_rcv()` inspects the MAC address and drops foreign ones. Then, the version, length, and checksum fields are verified. Next, the function sets the `transport_header` field of the `sk_buff`. It also applies `netfilter`'s `PRE_ROUTING` rule. It implements the filter by forwarding the packet to the `NF_INET_PRE_ROUTING` hook. The hook gets a pointer to the `ip_rcv_finish()` function that it calls after completion. If a network layer master device is registered, it passes the `sk_buff` to its handler. It calls `ip_route_input_noref()`, which reads the IP header from the `sk_buff`. Next, the kernel processes IP options via `ip_rcv_options()`. Afterward, it calls the previously selected routing function via `dst_input()`. There are three options for routing a packet:

1) `ip_forward`: This function activates for packets not addressed to the current machine. It proceeds by forwarding the packet without additional processing.
2) `ip_local_deliver()`: If we are the final receiver of the packet (*localhost*), the kernel does not forward the packet but passes it up the networking stack.
3) `ip_mr_input()`: This function is for multicast packets, i. e., addressed to a multicast address.

As we are mainly interested in how a packet is handled at the final receiver, taking all layers into account, we continue with `ip_local_deliver()`. Most importantly, this function takes care of IP fragmentation by calling `ip_defrag()`, queueing packets until receiving all fragments. Afterward, the event `NF_INET_LOCAL_IN` triggers, which in return calls `ip_local_deliver_finish()`, stripping the IP header from the `sk_buff`. Finally, it passes the packet from the IP to the TCP/UDP layer via the `dst_input()` function to the `tcp_v4_rcv()` or function. It determines the corresponding protocol handler by inspecting the header pointing to the `sk_buff`.

4.2.3. Transport Layer.

Now, we inspect the counterpart of the egress TCP and UDP functions.

TCP. First, the segment arrives at the transport layer function `tcp_ipv4_recv()` with the `sk_buff` header pointer moved to the start of the TCP or UDP header. Then, it validates the transport header via `pskb_may_pull()`, validating the TCP checksum. As before, it removes the TCP header from the `sk_buff`. To pass the packet further, it locates the corresponding TCP socket via `__inet_lookup_skb()`. It writes the packet to the socket receive queue (see Figure 4) and signals that new data is available, e. g., via `SIGIO` or `SIGURG`. This notification mechanism allows for efficient polling of sockets. As for the egress, the kernel maintains the TCP state machine during packet processing. e. g., it processes no new packets for TCP connections terminated via a `TCP_CLOSING`.

We briefly highlight two important cases during processing: `TCP_NEW_SYN_RECV` and `TCP_TIME_WAIT`. `TCP_NEW_SYN_RECV` means that there is a new connection. In this case, the kernel refuses the connection at TCP level via `tcp_filter()`. During `TCP_TIME_WAIT`, the kernel discards any further TCP segments.

Furthermore, there is a slow and a fast path. The slow path contains more error checks and lookups. In contrast, the fast path is optimized for speed, not allowing introspection and traffic analysis. With the slow path, we wait until the state machine is at `TCP_ESTABLISHED` in `tcp_v4_do_rcv()`. Once updated, `tcp_v4_do_rcv()` calls `tcp_rcv_established()`, which processes packets both in the fast and slow paths. It also validates that sequence numbers are ascending. The fast path copies the packet directly to the user space. The kernel always tries to use the fast path, if possible. But when, e. g., establishing a TCP connection, this not possible since the kernel has to track the new connection.

After handling the TCP state machine and choosing the path, the kernel enqueues the packet into the socket queue so the user program can read it (see Figure 4).

Since TCP is very complex, covering further aspects is beyond this paper's scope. However, [7], [10], [11] describe it in more detail.

UDP. Compared to TCP, the implementation of UDP is less complex. It starts with `udp_rcvmsg()` called via `dst_input()` in the IP layer. First, the function calls `__skb_recv_udp()` to read the datagram from the socket with a previously calculated offset. In particular, it continuously tries to read a `sk_buff` from the socket, eventually stopping when a new UDP datagram arrives. The checksum of the datagram is then validated. Then, the function copies the destination IP and UDP port to map the datagram to the correct socket. Consequently, it consumes the UDP datagram via `skb_consume_udp()`. Finally, it adjusts the peek offset, handles reference counters, and frees the `sk_buff` via `__consume_stateless_skb()`.

4.2.4. Socket Layer.

Here, the kernel collects the new data written to a TCP or UDP socket via the `read()` function from a socket, dequeuing the packet from the socket receive queue (see Figure 4). To match the use of IPv4 in the egress, we use an `AF_INET` receiving socket. The function `sys_recv()` enables this, first calling `sys_recvfrom()` to look up the socket. Then, it calls `sock_recvmsg()` to read from the socket and passes the received message through Linux Security Modules, similar to the egress. For IPv4, `inet_recvmsg()` calls either `tcp_recvmsg()` or `udp_recvmsg()`. They dequeue the packet's content and write it to a userspace buffer, e. g., an array on the heap. Finally, they free the `sk_buff`.

5. Conclusion

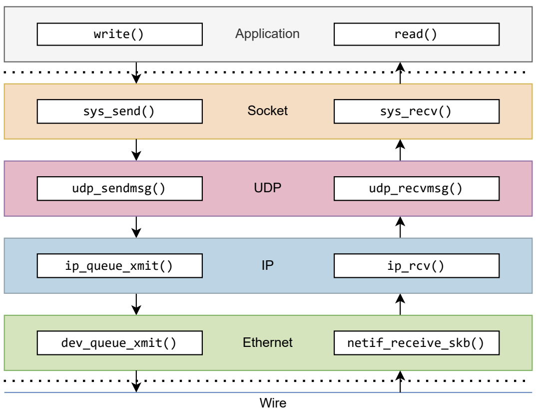

Figure 5: An overview of the most important functions in both egress and ingress for UDP, as described in Section 4.

This paper presented how a packet traverses the Linux kernel for TCP/IPv4 and UDP/IPv4. Figure 5 illustrates a recap of the packet egress and ingress path, highlighting the most important functions of each layer. Moreover, we described the intricacies of packet processing, including routing, filtering, and queuing mechanisms employed by the Linux kernel. Furthermore, we have seen how the different layers in the kernels communicate. By leveraging this knowledge, network administrators and developers can make informed decisions when optimizing network performance, designing security measures, or troubleshooting networking issues.

Overall, the observed changes to the existing literature are primarily enhancements rather than rewrites, e. g., refactorings or security improvements. A prime example is the choice of initial sequence numbers for TCP. For security reasons, the kernel authors revised the underlying hash algorithm multiple times [23]. The conservative changes make sense, as the protocols remain mostly untouched while the impact of errors is high. Performing a similar analysis for Multipath TCP or QUIC is future work.

References

[1] T.-H. Lee, S.-W. Kang, T. Kim, J.-S. Kim, and H.-J. Lee, "Smart Refrigerator Inventory Management Using Convolutional Neural Networks," in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2021, pp. 1–4.

[2] W3Techs. (2019) Usage Statistics of Operating Systems for Websites. https://w3techs.com/technologies/overview/operating_system. [Online; accessed 02-December-2023].

[3] ——. (2019) Usage Statistics of Unix for Websites. https://w3techs.com/technologies/details/os-unix. [Online; accessed 02-December-2023].

[4] J. Corbet. (2011) A JIT for Packet Filters. https://lwn.net/Articles/437981/. [Online; accessed 02-December-2023].

[5] C. Paasch and O. Bonaventure, "Multipath TCP," *Commun. ACM*, vol. 57, no. 4, p. 51–57, apr 2014. [Online]. Available: https://doi.org/10.1145/2578901

[6] M. T. Hossain, "A Review on IPv4 and IPv6: A Comprehensive Survey," 01 2022, International Interconnect Technology Conference (IITC). [Online]. Available: https://doi.org/10.13140/RG.2.2.18673.61284

[7] J. Crowcroft and I. Phillips, *TCP/IP and Linux Protocol Implementation: Systems Code for the Linux Internet*. USA: John Wiley & Sons, Inc., 2001.

[8] A. Chimata, "Path of a Packet in the Linux Kernel Stack," 01 2005. [Online]. Available: https://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Network_stack.pdf

[9] C. Guo and Z. Shaoren, "Analysis and Evaluation of the TCP/IP Protocol Stack of Linux," vol. 1, 02 2000, pp. 444–453 vol.1.

[10] Helali Bhuiyan and Mark E. McGinley and Tao Li and Malathi Veeraraghavan, "TCP Implementation in Linux : A Brief Tutorial," 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:14676835

[11] Antti Jaakkola, "Implementation of Transmission Control Protocol in Linux," 2012. [Online]. Available: https://wiki.aalto.fi/download/attachments/70789052/linux-tcp-review.pdf

[12] F. U. Khattak. (2012) IP Layer Implementation of Linux Kernel Stack. [Online]. Available: https://wiki.aalto.fi/download/attachments/70789059/linux-kernel-ip.pdf

[13] M. C. Wenji Wu, "The Performance Analysis of Linux Networking - Packet Receiving," 2006, 15th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2006).

[14] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding Host Network Stack Overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 65–77. [Online]. Available: https://doi.org/10.1145/3452296.3472888

[15] M. Abranches, O. Michel, and E. Keller, "Getting Back What Was Lost in the Era of High-Speed Software Packet Processing," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, ser. HotNets '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 228–234. [Online]. Available: https://doi.org/10.1145/3563766.3564114

[16] J. Benc, "The Network Packet's Diary: A Kernel Journey," 2018, devcon.cz 2018.

[17] The kernel development community. struct sk_buff. https://docs.kernel.org/networking/skbuff.html. [Online; accessed 02-December-2023].

[18] G. Foundation. (2023) The GNU C Library Reference Manual. https://sourceware.org/glibc/manual/2.38/html_mono/libc.html#Transferring-Data. [Online; accessed 05-December-2023].

[19] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed. USA: No Starch Press, 2010.

[20] "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," RFC 826, Nov. 1982. [Online]. Available: https://www.rfc-editor.org/info/rfc826

[21] *tc - traffic control utility*, Linux Documentation Project, December 2023, https://man7.org/linux/man-pages/man8/tc.8.html.

[22] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66. [Online]. Available: https://doi.org/10.1145/3281411.3281443

[23] F. Gont and S. Bellovin, "Defending Against Sequence Number Attacks," RFC 6528, Feb. 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6528