

Moving Down the Stack: Performance Evaluation of Packet Processing Technologies for Stateful Firewalls

Katharina Dietz, Nicholas Gray, Manuel Wolz, Claas Lorenz^{*}, Tobias Hoßfeld, Michael Seufert
University of Würzburg, `firstname.lastname@uni-wuerzburg.de`
^{*}genua GmbH, `claas_lorenz@genua.de`

Abstract—Software-based network security solutions using SDN/NFV provide high flexibility and short development cycles, but may impose a bottleneck onto the network due to their lack of ASIC-based hardware packet processing. To overcome this limitation, several frameworks have emerged to enable flexible high speed packet processing in software, e.g., NAPI, XDP, or DPDK, or on programmable data planes in hardware, e.g., P4. Despite aiming for a common goal, the design principles of these technologies diverge, which raises the question of their suitability for critical security-related network functions, such as firewalls. In this work, we implement a stateful firewall, which is capable of tracking TCP state and sequence numbers, for each of the four aforementioned high speed packet processing technologies and make the firewall modules publicly available. We integrate multithreading strategies, where applicable, and discuss the impact of each packet processing technology during the development process. Finally, we evaluate and compare their performance in terms of throughput in two scenarios following the guidelines of RFC3511 in a 100 Gbps testbed.

I. INTRODUCTION

In recent years, innovation has brought forward a myriad of new Internet services, communication protocols, and device categories, which led to new application fields like the Internet of Things (IoT), Industry 4.0, and Smart Cities. Despite the benefits of these novel technologies, they also broaden the attack surface. This is further aggravated by an ever rising demand of bandwidth, thus, increasing the overall complexity. Driven by the rising market share and importance of these systems, cyber-criminals are constantly expanding their efforts to profit from this changing environment [1].

Countermeasures, such as firewalls, have to evolve continuously to cope with current and future threats to provide network security. Firewalls are commonly deployed as middleboxes, allowing for the use of Application-specific Integrated Circuits (ASICs), which can greatly improve their performance. Yet, being situated at the network's perimeter limits their effectiveness as they are unable to mitigate insider attacks or contain the damage once their borders have been breached. Furthermore, they are rather inflexible as they are required to be physically connected at the defined security boundaries and hence do not integrate well into cloud environments. Concepts like Bring Your Own Device (BYOD) or the increased necessity to work from home due to COVID-19 make it even more difficult to define clear security boundaries [2].

To counteract these drawbacks, concepts, such as virtualized software-based firewalls, gained attention. Software-defined Networking (SDN) and Network Function Virtualization (NFV)

allow for fast development cycles and more flexibility by separating the data and control plane of legacy devices, thus, enabling the use of Commodity-Off-the-Shelf (COTS) platforms instead of proprietary systems.

A firewall implemented as Virtualized Network Function (VNF) may impose a bottleneck onto the network due to its lack of dedicated hardware acceleration. This is especially severe in core and backbone networks with high bandwidth capacities, as every packet needs to be inspected. Several techniques have emerged to optimize the packet processing stack, e.g., by interrupt mitigation/packet throttling via the overhauled New API (NAPI) [3], early hooks for extended Berkeley Packet Filter (eBPF) programs within the kernel with the eXpress Data Path (XDP) [4], or entirely bypassing the kernel via the Data Plane Development Kit (DPDK) [5]. Additionally, advances in Programming Protocol-independent Packet Processors (P4) [6] allow for the deployment of flexible custom programs, which can run directly on the switching equipment.

Although the aforementioned approaches achieve their performance optimization differently, they all move functionality down the stack, i.e., closer to the origin of the network packet, saving on operations and context switches. This comes at the cost of enforced limitations and design principles, dictated by the specific packet processing technology, thus, increasing development costs or even rendering the implementation impossible. Besides usability concerns, the question arises to what extent the performance gain is dictated by the application, packet processing technology, and other optimizations, e.g., multithreading. It remains unclear how suitable these technologies are for virtualized security applications in core and backbone networks with speeds up to 100 Gbps.

The contributions of this work are as follows:

- ① Implementation and publication [7] of a modular stateful firewall supporting four high speed packet processing technologies, i.e., Linux NAPI, XDP, DPDK, and P4.
- ② Subjective discussion of the impact of packet processing technologies on the development process.
- ③ Objective performance evaluation of the packet processing technologies as engine for a stateful firewall in a high speed 100 Gbps testbed, following the guidelines of RFC3511 [8].

The organization of this work is structured as follows. Section II summarizes background and related work, before the testbed setup and implementation of the stateful firewall is detailed in Section III. Section IV presents the evaluation results, and Section V concludes this work.

II. BACKGROUND & RELATED WORK

Stateful Firewalls. A firewall is defined as a networking component, which inspects in- and outgoing network traffic at a security boundary against a set of policies and blocks violating data packets. The firewall can be implemented in software and run as a virtual appliance, or can be deployed as a physical device within the networking infrastructure. Several types of firewalls exist [9], categorized by the amount of information taken into account during the filtering process as well as the layer of the Open Systems Interconnection (OSI) [10] architecture on which they are operating.

As we focus on the performance of the underlying packet processing techniques and not computationally intense attack detection/mitigation logic as needed for firewalls of higher generations, we implement a stateful packet filter. Commercial stateful packet filters also support state verification of Internet Control Message Protocol (ICMP) and User Datagram Protocol (UDP) messages, while our implementation focuses on the state inspection of Transport Control Protocol (TCP) packets.

As defined in RFC793 [11], TCP is connection-oriented and requires the communication partners to partake in a three-way-handshake, illustrated in Figure 1. Each packet transmission/reception triggers a transition within the TCP state machine on the client, server, and the stateful firewall. A similar procedure is applied to close a connection. To enable packet reordering and packet loss detection, the protocol employs sequence (SEQ) and acknowledgement (ACK) numbers, where the SEQ number of the first packet is randomized and following packets continuously increase this number by the bytes previously sent. The firewall evaluates a packet’s SEQ and ACK numbers on a per flow basis and limits the range of valid values, making it harder to successfully attempt a TCP hijacking attack [12], as the injected packets need to comply to the narrow range of accepted SEQ and ACK numbers. While the validation of TCP state transitions mainly concerns only initial and final packets of a connection, validating SEQ and ACK numbers requires the inspection of all packets.

Enabling quick adaption of the infrastructure to new protocols and services, NFV [13] gained increasing popularity. On-demand deployment and a seamless integration into cloud environments helps saving capital expenditures (CAPEX) and operating expenses (OPEX). Though, performance bottlenecks of VNFs remain an ongoing challenge, especially with the increase in transmission speeds of modern networks, the lack of sophisticated scheduling/filtering of modern NICs, and the amount of context switches involved in the network stacks of general purpose operating systems (OSs) [14, 15]. To address these bottlenecks, dedicated packet processing frameworks have emerged, such as NAPI, DPDK, and XDP.

The need for dynamicity not only benefited NFV, but also SDN [16]. By decoupling the control from the data plane via programmable open interfaces, a new device category emerged. These devices evolved to SmartNICs [17] and programmable switches [18] capable of running task specific applications directly on the hardware, often supporting the P4 language [19].

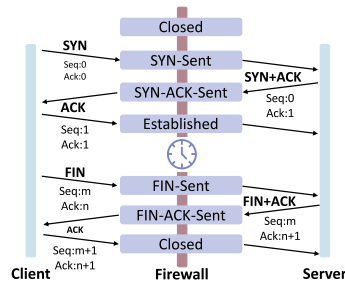


Fig. 1: Overview of firewall state transitions.

Existing Software-based Solutions. The tendency towards software-based implementations is reflected by the multitude of available systems [20]–[24]. Significant performance improvements are reported in [25]–[28] when upgrading systems with XDP. Similarly, DPDK was incorporated in the past to improve existing solutions [29]–[35]. The active use and impressive results led us to include these frameworks within our investigations. Enabled by the rise of SDN, OpenFlow-based stateful firewalls have been implemented [36, 37]. Due to devices’ limitations of tracking stateful information, the approaches require the interaction of a controller unit, creating a trade-off between security and performance. Next generation SDN-enabled devices provide more control over internal processes and can often be programmed directly via P4. [38]–[40] feature first implementations of basic stateful firewalls whereas [41] presents an implementation featuring simple stateless packet filtering. Other systems evaluate various port knocking authentication mechanisms [42], scalable DDoS mitigation [43, 44], or high performance Intrusion Detection Systems (IDS) [45, 46], often coupled with offloading specific tasks to P4-enabled switches. Encouraged by these recent advances, we aim for implementing a stateful firewall capable of tracking TCP state transitions and SEQ numbers in P4. Thriving for a full implementation of the functionality within a P4-enabled switch, we fall back to an intermediary controlling unit when necessary as detailed next.

III. TESTBED DESCRIPTION, SYSTEM DESIGN, AND DEVELOPMENT EXPERIENCE

Testbed Description. Figure 2 depicts the schematic server configuration for the software-based approaches, i.e., NAPI, DPDK, and XDP, as well as the setup for P4. In both cases a single server acts as traffic generator/sink and is connected with both ports to the device under test (DuT), running the firewall modules. The servers in use are composed of an Intel Xeon Silver 4114 @ 2.20GHz CPU with a total of 10 cores and 188GB RAM. On both systems, the installed OS is Ubuntu 18.04.5LTS and the kernel version is fixed to 5.4.0-42-generic. The servers are equipped with a two-port Mellanox ConnectX-5 NIC. Using both ports, we mimic common deployments, where each port resides in its own network, thus, creating a security boundary. Note that our setup in principle can be replicated with any server and any multi-port NICs or multiple single-port NICs. However, DPDK and XDP require using a NIC driver, which supports these packet processing frameworks. For the P4 approach, we employ the Barefoot Wedge 100BF-65X



Fig. 2: Overview of the testbed configuration.

P4-enabled Tofino switch with a total switching capacity of 6.5 Tbps, featuring 64 ports each having a maximum capacity of 100 Gbps. The chassis includes a host controller, which is composed of a 8-core Intel Xeon CPU with 32 GB of memory running Ubuntu 18.04.

Firewall Modules Each firewall implementation consists of three major building blocks, i.e., policy configuration, filter & state logic, and packet processing. Initially, all firewalls need to be configured by specifying the allowed connections, defined by a 5-tuple of source/destination IP, port, and the protocol. Currently, the Address Resolution Protocol (ARP) and TCP are supported. The default firewall behavior is to block incoming and outgoing packets, if no policy explicitly allows the transmission. The policies are loaded into a map data structure to provide $O(1)$ access during the filtering stage. The key of the hash map’s entry is derived as hash from the extracted 5-tuple and the stored data is composed of the initialized TCP state information.

In the filter & state logic stage, each packet is subject to two validation checks. At first, the policy map is queried to confirm that the packet header information is compliant, before verifying the state information. Here, the stored state of the TCP state machine is first compared to the packet’s TCP flags, which may result in a violation, pass, or pass/state transition. In case of a pass, the next verification examines if the packet’s SEQ number remains within the expected boundaries. In addition to the original equations given in RFC793 [11] to validate SEQ numbers, we also incorporate the Window Scale Factor [47], which allows sending and receiving larger block sizes to increase the performance in high bandwidth networks. If the packet passed the SEQ number check, the state information is updated and the packet is cleared for forwarding or dropped otherwise.

The packet processing component is responsible for receiving packets from the NIC, triggering the aforementioned validation processes, and – in case of a benign packet – sending it towards its destination. This implies that a received packet has to be transferred to another port, which is subject to different copy processes depending on the packet processing technology, and thus, may introduce a performance bottleneck. Figure 3 illustrates the design concepts of the three packet processing frameworks used in this work, namely Linux NAPI, DPDK, and XDP. It shows where the application is running and at which stages system resources are consumed.

NAPI. NAPI addresses high computational overhead or even live locks of standard Linux network routine triggers by introducing interrupt mitigation. The system is informed via

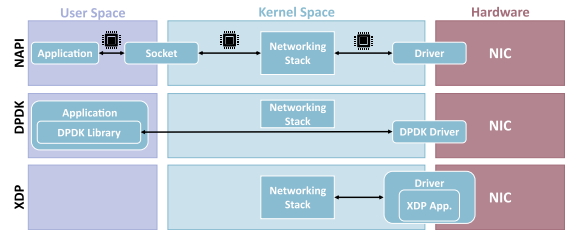


Fig. 3: Overview of utilized packet processing engines.

an interrupt of an arriving packet as previously, but deactivates future interrupts of the same NIC to transition to a polling mode. NAPI also introduces packet throttling to reduce any overhead further by simply overwriting packets within the NIC’s ring buffer if not processed in time, instead being dropped by direct kernel interaction.

However, as depicted in Figure 3, a packet is still subject to several copy operations and context switches until it is received by the application running in user space. At first, the packet is copied from the NIC by the driver to dedicated kernel structures and processed further. As the OS supports various protocols and operations on packets, these operations may not be tailored to the specific use case of the implemented network function, resulting in inefficient resource utilization. Next, the contents of the packets are requested via a system call through a socket to user space causing a context switch and triggering additional memory reallocation routines wasting time and resources as analyzed in [48].

Being the most basic implementation, the realization via NAPI serves as reference point and relies on two raw sockets of the *AF_PACKET* protocol family as it provides access to the entire packet without stripping the header fields. To prevent blocking calls the function *recv* is called with the *MSG_DONTWAIT* flag in an active polling loop. The hash functionality is provided by the Klib [49] library, which was chosen for its small footprint and good performance [50]. Multithreading is accomplished by the standard Pthreads library.

DPDK. DPDK improves the performance by providing direct access to the contents of the packets to the application through a dedicated driver, thus, circumventing the kernel of the OS entirely as displayed in Figure 3. It cuts down on possibly unnecessary operations and employs optimization techniques, e.g., zero-copy and pre-allocation of memory via Hugepages. Though, it forces the application to possibly re-implement certain network functions, e.g., routing and forwarding, which would be otherwise available by the OS, and it relays greater responsibility on the application as safeguards enforced by the system’s kernel are bypassed.

Before DPDK can be utilized within user space, a particular driver needs to be loaded, a compatible NIC has to be present in the server, and the Hugepages, which provide the memory storage for the packets, need to be mounted. We load the *uio_pci_generic* driver in combination with our Mellanox ConnectX-5 NIC and DPDK in version stable-18.05.1. One receive and one send queue is bound to each of the two ports of the interface. Klib is again employed for the hash computation and storage management of the hash map, and multithreading is

now implemented directly via DPDK’s *rte_ctrl_thread_create* function, which is optimized for the framework. As before, the packets are actively and constantly polled from the NIC.

XDP. Whereas the aforementioned approaches consolidate the main functionality of the firewall in user space, the XDP implementation runs almost completely in kernel space, as seen in Figure 3. This is achieved by executing eBPF [51] programs directly in the kernel within a VM. A user space program initiates eBPF code and links it to the interface via the *bpf_set_link_xdp_fd* function. As we use two ports of our NIC, the initiation process needs to be done for both interfaces and an eBPF map ties the individually operating programs together for forwarding purposes. Furthermore, an eBPF map replaces the prior Klib hash map, as external libraries are unavailable in this context. Yet, the key to the hash map’s entry and the stored data remain unchanged. Once the user space has finished the initialization process, it yields and merely waits for an user interrupt to properly shutdown the firewall operation.

P4. Moving further down the stack, the last firewall implementation offloads certain functionality to a P4-enabled switch, where each packet is processed by a pipeline design pattern [19]. The advantage compared to the aforementioned solutions is that we can achieve linerate on all ports, 6.5 Tbps in our scenario, which is not achievable with the software-based solutions. Here, each packet is subject to a parser, which is able to extract the metadata. This is used to perform lookups against Match+Action tables to determine the further fate of the packet, e.g., altering, dropping, or forwarding the packet. The Tofino platform implements the so-called Protocol-Independent Switch Architecture (PISA). This extends the P4 base operations by enabling re-submission or packet mirroring/cloning, and offers additional resources for basic arithmetic operations.

Determined to port the entire firewall module to P4-programmable hardware, we quickly realized that this is impossible due to the restrictions of the targeted device. The most limiting factors are the amount of arithmetic operations, which can be performed within one pipeline stage, and not being able to read and write to the same register. Therefore, a hybrid approach was chosen similar to [36], i.e, the functionality is split between the P4 switch, which handles the forwarding, and a controller, which implements the state tracking.

As our Tofino platform features an integrated management host able to run a standard Linux OS, the controller is implemented on the management host and can communicate via the PCIe bus with the programmable switching fabric. However, the standard Apache Thrift [52] API, which is used by a Python controller, may impose a bottleneck. Thus, we implemented an additional C controller, which accesses the PCIe bus directly. For both implementations, the controller program reads the configuration and installs rules to initially redirect the packets of allowed connections to the controller. The controller then extracts the relevant header information to monitor and validate the state. In case of a violation, the controller drops the packet. Otherwise, it is sent back to the switch and marked with a special ingress port identifier, whitelisted by the P4 program, so that it can be forwarded directly. As no packet can be queued in

TABLE I: Subjective experience with packet processing technologies, using the absolute category rating scale (XX:bad, X:poor, (✓):fair, ✓:good, ✓✓:excellent).

	NAPI	DPDK	XDP	P4
Documentation	✓✓	✓✓	X	X
Universality	✓✓	✓✓	(✓)	X
Functional Completeness	✓✓	✓✓	✓✓	X
External Libraries	✓✓	✓✓	X	XX
Ease of Integration	✓✓	✓✓	✓	XX
Level of Control	✓✓	✓✓	X	X
Risk	✓	X	✓✓	✓

the memory of the switch with the option to stall until further instructions are received, the entire packet has to be forwarded to the controller. This further increases the load on the PCIe bus as no compression or aggregation can be performed. Early development stages proved that forwarding every packet to the controller breaches the device’s limitations and that the validation of SEQ numbers is not possible. Thus, this validation was disregarded for the P4-based firewall module. This allowed us to alter the process, so that only packets which trigger a TCP state transition are relayed to the controller, e.g., packets during the initiation and the tear down of the connection, while packets carrying the main payload can be forwarded instantly.

Multithreading Strategies. For NAPI and DPDK, we implement a total of four multithreading strategies as depicted in Figure 4. In the first variant, a single thread is used, which handles the entire functionality. Hence, multicore architectures cannot be used to their full capacity and the probability of CPU cache misses rises due to the context switches. The two thread variant merely parallelizes the functional blocks regarding both ports of the NIC. The third approach extends this concept by incorporating one thread for each of the functional blocks, i.e., reading packets from the internal and external network, validating packets, and forwarding packets to the internal and external networks, summing up to a total of five threads. The last variant further breaks apart the validation of packets, as this is the computationally most intense part of the code. A single thread is used to confirm the connection validity by examining the packet’s header fields before the packet’s state is checked. For the latter a variable number of worker threads can be spawned. We opted for two worker threads as starting point, which results in a total of seven threads for this variant.

Subjective Developer Experience. Table I summarizes our experiences using a subjective scoring on a five-point absolute category rating scale ranging from bad to excellent. Whereas the documentation of NAPI and DPDK is excellent and features many examples, XDP and the Tofino platform lack in this category due to their novelty and public unavailability. The most development time was spent on the latter technologies. The term *universality* describes the flexibility of the technologies to be applied in different fields. We assign NAPI and DPDK a high rating, as they run completely in user space. XDP is limited regarding its instruction set, but provides a built-in opportunity to escalate the packet either to other kernel functions or to user space. A similar statement can be made for P4 in combination with the Tofino platform, yet the forwarding of packets to the controller requires additional efforts. In terms of functional

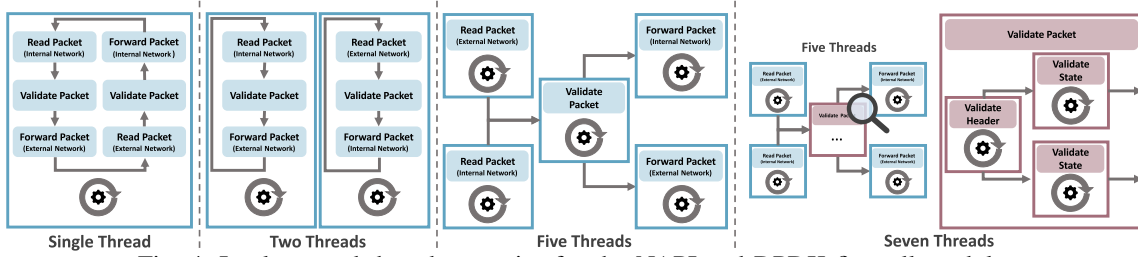


Fig. 4: Implemented thread strategies for the NAPI and DPDK firewall modules.

completeness, we were able to implement the desired firewall module with all technologies except for P4. Due to their design, XDP and P4 lack the support of external libraries. Yet, as stated before XDP provides means to easily subject the packet to kernel functions or to a user space program. Regarding the ease of integration, using the NAPI implementation as reference point, DPDK and XDP required almost no adaptations to the main firewall logic. In contrast, the realization via P4 involved a complete rewrite, an adaptation of the requirements, and requires to be deployed physically in the network. The level of control shows a similar trend. NAPI and DPDK provide various ways to access and process the packets in combination with how and where the program is run, XDP can only be hooked to certain execution stages, and P4 enforces its pipeline design pattern. At last, XDP provides the most safety due to its limited instruction set, secure execution environment, and code validation. NAPI and P4 achieve the same score, as packet processing is protected by the kernel or by dedicated firmware with limited access for change. DPDK bypasses the kernel, which may result in unrecoverable errors due to user errors.

IV. EVALUATION

Scenario Configurations. Following the guidelines of RFC3511 [8], we evaluate the firewall modules in two scenarios. In the first scenario the impact of long lasting flows with varying packet sizes is investigated. Here, *iperf2* [53] is configured on the traffic generator/sink server to send and receive TCP streams. To saturate the link to the best possible rate, a total of 8 sending processes are executed in parallel. To prevent the OS to forward the packets internally, the sending and receiving processes are started in different network namespaces [54]. All processes are bound to a CPU using CPU core pinning for optimal performance [55]. The packet size is either set to 88 B, the minimum size of a 802.1Q [56] tagged frame, or 1500 B, a common maximum MTU size, as the packet size has a direct impact on the number of generated packets, influencing the load on the firewall module under test. The available bandwidth is set to different common rates as a parameter study. All parameter combinations are repeated 5 times and executed for 30 s, of which the first 10 s are discarded as transient phase.

The second scenario evaluates the performance of the firewall modules for HTTP traffic patterns, which feature fixed web page sizes and concurrent connections. Here, single web pages for various content sizes have been created with the Linux tool *dd* [57]. The pages are statically supplied by an Apache2 [58] web server to a Varnish [59] front end cache, which is configured to hold the web pages in RAM and features

an optimized network stack. The requests are executed by the Apache HTTP server benchmarking tool (*ab*) [60], which is configured to run until 1280 requests are completed. Regarding the varying number of concurrent connections, a parameter study is performed and all configurations are repeated 5 times.

Baseline. Before diving into detailed performance analyses of the four firewall implementations, we conduct a baseline measurement without the interaction of a firewall, as the output of the traffic generation process limits the maximum achievable performance. To do so, the NIC ports of the traffic generator and sink are directly connected. For the *iperf* measurements an average throughput rate of 70 715 Mbps could be achieved when using 1500 B as packet size, and only 4217 Mbps were achieved for 88 B packets while using 16 cores. A similar behavior can be seen for a rising number of concurrent connections for the HTTP scenario with *ab*. At first, a higher number of connections benefits the throughput utilization, as the link capacity can be utilized more efficiently. In detail, the throughput with a web page size of 1 KB rises from 47 Mbps for one connection, to 267 Mbps for eight parallel connections. Though, 256 parallel connections drastically reduce the performance to 65 Mbps again. The same trend can be observed for larger web page sizes at a different scale. For a web page size of 1 MB, the throughput starts at 2223 Mbps, rises to 13 226 Mbps with eight connections, and sinks to 8687 Mbps again for 256 connections. These maximum values have to be considered when analyzing the firewall implementations as a reduction in the performance may be linked to the degradation of the traffic generator and not the firewall module.

IP Throughput. Figure 5 depicts the results for the IP throughput using *iperf* for all firewall implementations and multithreading strategies. The upper and lower boxplot represent the two packet sizes 88 B and 1500 B, the x-axis depicts the particular implementation, the y-axis the achieved throughput, and the color-coding the applied bandwidth limitation. The boxes extend from the lower to upper quartile showing a line at the median. The whiskers show the range of the data except for outliers, which are plotted as separate points. The round marker indicates the mean value.

Comparing the achieved performance of the software-based firewall implementations only, the DPDK module using a single thread clearly outperforms NAPI and XDP. This is true for both packet sizes, yet smaller packets also have a strong impact on DPDK lowering its effectiveness by a factor of almost 50. NAPI struggles for both packet sizes, while the throughput of the XDP firewall leans more towards the performance of

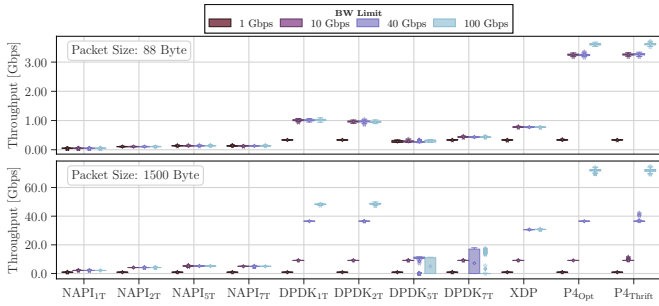


Fig. 5: IP throughput using *iperf*.

DPDK. Taking a closer look at the different multithreading strategies, NAPI is the only module which profits from this optimization approach. However, the performance gain quickly stagnates with a rising number of utilized threads. For DPDK the implementation using 7 threads is especially devastating, which is reflected by the large variation of values starting with an offered load of 40 Gbps. Looking at the relative speedup over the single thread implementation, NAPI can benefit from multithreading up to 5 threads, while the speedup of DPDK consistently follows a downward trend with more threads.

In contrast to the software-based firewall modules, the P4-enabled implementation only performs state validation and omits SEQ number checking. As shown in the figure, both controller implementations operate at line rate considering the limited capacity of the traffic generator. At last, the plot shows a smooth and reliable operation for all firewall modules as long as their limitations are not reached.

HTTP Transfer Rate. Figure 6 illustrates the measurements for the HTTP transfer rate using *ab* for all implementations as boxplots and for a multitude of parallel connections. The analysis focuses on two web page sizes, i.e., 1 KB in the upper and 1 MB in the lower boxplot, as these provide a suitable resemblance of the entire parameter study. The x-axis depicts the implementation, the y-axis the achieved throughput, and the color-coding the number of concurrent connections.

In opposition to the previous analyses with *iperf*, most of the software-based implementations perform similar regarding the page size of 1 KB, hence proving a strong dependence on the subjected traffic. For larger page sizes, DPDK and XDP are more effective than NAPI, except for the 7-thread implementation of DPDK. As we repeated all runs with this configuration, we can foreclose the possibility of a measurement error in this case, but the exact cause of its poor performance is yet to be identified. When comparing DPDK and XDP directly, XDP is able to obtain similar speeds as DPDK.

In stark contrast to the software-based solutions stand the P4-based implementations. The results clearly indicate that incorporating the controller for connection initiation and tear down imposes a significant bottleneck. This becomes even more obvious when comparing it directly to the *iperf* measurements for the IP throughput, where the flows are long lasting and the P4-based modules outperformed the software-based solutions and reached the limit of the traffic generator because the initial trip to the controller is amortized over time. In contrast, detouring the short HTTP flows via the controller has a

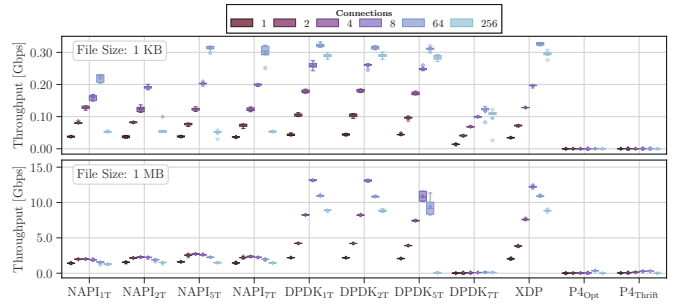


Fig. 6: HTTP transfer rate using *ab*.

comparatively large impact on the performance. As the small data volume of the HTTP flow is quickly transmitted, the connection is only of short duration, which does not allow to amortize the controller detour over time. To provide a numerical example, the throughput of the P4-based solutions is at best at 0.5 Mbps for 1 KB web pages, compared to possibly over 300 Mbps for the software-based solutions.

V. CONCLUSION

Driven by the increasing number of cyber threats, concepts like SDN and NFV are promising technologies, already integrated into security solutions today. However, the packet processing in software may impose a performance burden onto the entire network, especially if a high bandwidth is demanded.

In this work, we therefore evaluate three packet processing stacks regarding their usability and performance by the example of a stateful firewall. In addition, we try to port the functionality to a P4-enabled switch, hence moving the operation further down the stack and closer to the packet. The evaluation of the packet processing technologies showed that high throughputs of up to 50 Gbps are achievable with the right framework. However, the experiments also proved that these technologies may behave differently and unexpected if they are subjected to varying traffic patterns or if multithreading strategies are applied. Our investigation regarding the P4-based implementation revealed certain shortcomings in the capabilities of the device, which made the need for an external controller essential. Unfortunately, this design change imposed a critical bottleneck on the system, which made the implementation futile for specific scenarios. In general, we found that current software-based solutions express higher usability than the implement P4-based approach, with the DPDK-based firewall module emanating as a clear winner from our investigations, both from a subjective and objective point of view, i.e., with regards to usability as well as performance.

For future work, we plan to investigate additional packet processing technologies like Netmap [61], PF_Ring [62], or Vector Packet Processing (VPP) [63], and traffic generators such as *pktgen* [64] or *trex* [65].

ACKNOWLEDGMENT

This work has been performed in the framework of the WINTERMUTE project, which is funded by the BMBF (Project ID 16KIS1129). The authors want to thank Michael Ott, Felix Maurer, and Joshua Schüler for their programming efforts.

REFERENCES

- [1] <https://www.checkpoint.com/downloads/resources/cyber-security-report-2020.pdf>, [Online; accessed 18-January-2023].
- [2] <https://www.sophos.com/en-us/mediablibrary/PDFs/technical-papers/sophos-2021-threat-report.pdf>, [Online; accessed 18-January-2023].
- [3] J. H. Salim, "When NAPI comes to town," in *Linux 2005 Conf*, 2005.
- [4] T. Höiland-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*, 2018, pp. 54–66.
- [5] <https://www.dpdk.org/>, [Online; accessed 18-January-2023].
- [6] <http://p4.org/specs/>, [Online; accessed 18-January-2023].
- [7] <https://github.com/linfo3/noms2023-firewall-implementations>, [Online; accessed 17-January-2023].
- [8] B. Hickman, D. Newman, S. Tadjudin, and T. Martin, "RFC3511: Benchmarking methodology for firewall performance," 2003.
- [9] K. Neupane, R. Haddad, and L. Chen, "Next generation firewall for network security: A survey," in *SoutheastCon 2018*. IEEE, 2018, pp. 1–6.
- [10] H. Zimmermann, "OSI reference model-the ISO model of architecture for open systems interconnection," *IEEE Transactions on communications*, vol. 28, no. 4, pp. 425–432, 1980.
- [11] J. Postel, "RFC793: Transmission control protocol. usc," *Information Sciences Institute*, vol. 27, no. 793, pp. 123–150, 1981.
- [12] B. Harris and R. Hunt, "TCP/IP security threats and attack methods," *Computer communications*, vol. 22, no. 10, pp. 885–897, 1999.
- [13] B. Yi, X. Wang, K. Li, M. Huang *et al.*, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212–262, 2018.
- [14] J. L. Garcia-Dorado, F. Mata, J. Ramos, P. M. S. del Río, V. Moreno, and J. Aracil, "High-performance network traffic processing systems using commodity hardware," in *Data traffic monitoring and analysis*. Springer, 2013, pp. 3–27.
- [15] G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, "Profiling and accelerating commodity NFV service chains with SCC," *Journal of Systems and Software*, vol. 127, pp. 12–27, 2017.
- [16] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.
- [17] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, "Survey of performance acceleration techniques for network function virtualization," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 746–764, 2019.
- [18] <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.htm>, [Online; accessed 18-January-2023].
- [19] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
- [20] <https://www.netfilter.org/projects/iptables/index.html>, [Online; accessed 18-January-2023].
- [21] <https://www.pfsense.org/>, [Online; accessed 18-January-2023].
- [22] <https://www.cisco.com/c/en/us/products/collateral/security/adaptive-security-virtual-appliance-asav/adapt-security-virtual-appliance-ds.html>, [Online; accessed 18-January-2023].
- [23] <https://suricata.io/>, [Online; accessed 18-January-2023].
- [24] <https://www.snort.org/>, [Online; accessed 18-January-2023].
- [25] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating Linux security with eBPF iptables," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, 2018, pp. 108–110.
- [26] 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. 1. IEEE, 2018, pp. 209–217.
- [27] G. Bertin, "XDP in practice: integrating XDP into our DDoS mitigation pipeline," in *Technical Conference on Linux Networking, Netdev*, vol. 2, 2017.
- [28] N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Accelerating virtual network functions with fast-slow path architecture using eXpress data path," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1474–1486, 2020.
- [29] D. Zhang and S. Wang, "Optimization of traditional Snort intrusion detection system," in *IOP Conference Series: Materials Science and Engineering*, vol. 569, no. 4. IOP Publishing, 2019, p. 042041.
- [30] L. Shuai and S. Li, "Performance optimization of Snort based on DPDK and Hyperscan," *Procedia Computer Science*, vol. 183, pp. 837–843, 2021.
- [31] G. Pongrácz, L. Molnár, and Z. L. Kis, "Removing roadblocks from SDN: OpenFlow software switch performance on Intel DPDK," in *2013 Second European Workshop on Software Defined Networks*. IEEE, 2013, pp. 62–67.
- [32] <https://software.intel.com/content/www/us/en/develop/articles/open-vs-switch-with-dpdk-overview.html>, [Online; accessed 18-January-2023].
- [33] J. E. Varghese and B. Muniyal, "An efficient IDS framework for DDoS attacks in SDN environment," *IEEE Access*, vol. 9, pp. 69 680–69 699, 2021.
- [34] M. Trevisan, A. Finamore, M. Mellia, M. Munafò, and D. Rossi, "Traffic analysis with off-the-shelf hardware: Challenges and lessons learned," *IEEE Communications Magazine*, vol. 55, no. 3, pp. 163–169, 2017.
- [35] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi, "FlowMon-DPDK: Parsimonious per-flow software monitoring at line rate," in *2018 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, 2018, pp. 1–8.
- [36] C. Lorenz, D. Hock, J. Scherer, R. Durner, W. Kellerer, S. Gebert, N. Gray, T. Zinner, and P. Tran-Gia, "An SDN/NFV-enabled enterprise network architecture offering fine-grained security policy enforcement," *IEEE communications magazine*, vol. 55, no. 3, pp. 217–223, 2017.
- [37] F. Heimgaertner, M. Schmidt, D. Morgenstern, and M. Menth, "A software-defined firewall bypass for congestion offloading," in *2017 13th International Conference on Network and Service Management (CNSM)*. IEEE, 2017, pp. 1–9.
- [38] <https://github.com/p4lang/tutorials/tree/master/exercises/firewall>, [Online; accessed 18-January-2023].
- [39] https://github.com/open-nfsw/p4c_firewall, [Online; accessed 18-January-2023].
- [40] R. Datta, S. Choi, A. Chowdhary, and Y. Park, "P4Guard: Designing P4 based firewall," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 1–6.
- [41] P. Vörös and A. Kiss, "Security middleware programming using P4," in *International Conference on Human Aspects of Information Security, Privacy, and Trust*. Springer, 2016, pp. 277–287.
- [42] E. O. Zaballa, D. Franco, Z. Zhou, and M. S. Berger, "P4Knocking: Offloading host-based firewall functionalities to the network," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2020, pp. 7–12.
- [43] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric DDoS attacks with programmable switches," in *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.
- [44] A. Febro, H. Xiao, and J. Spring, "Distributed SIP DDoS defense with P4," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2019, pp. 1–8.
- [45] B. Lewis, M. Broadbent, and N. Race, "P4ID: P4 enhanced intrusion detection," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2019, pp. 1–4.
- [46] N. Gray, K. Dietz, M. Seufert, and T. Hossfeld, "High performance network metadata extraction using P4 for ML-based intrusion detection systems," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2021, pp. 1–7.
- [47] V. Jacobson, R. Braden, and D. Borman, "RFC1323: TCP extensions for high performance," 1992.
- [48] T. Zinner, S. Geissler, S. Lange, S. Gebert, M. Seufert, and P. Tran-Gia, "A discrete-time model for optimizing the processing time of virtualized network functions," *Computer Networks*, vol. 125, pp. 4–14, 2017.
- [49] <https://github.com/attractivechaos/klib>, [Online; accessed 18-January-2023].
- [50] <https://attractivechaos.wordpress.com/2008/10/07/another-look-at-my-old-benchmark/>, [Online; accessed 18-January-2023].
- [51] <https://ebpf.io/>, [Online; accessed 18-January-2023].
- [52] <https://thrift.apache.org/>, [Online; accessed 18-January-2023].
- [53] <https://iiperf.fr/>, [Online; accessed 18-January-2023].
- [54] <https://man7.org/linux/man-pages/man8/ip-netns.8.html>, [Online; accessed 18-January-2023].
- [55] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of CPU pinning and partial CPU loads on performance and energy efficiency," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 1–10.
- [56] <https://www.ieee802.org/1/pages/802.1Q-2014.html>, [Online; accessed 18-January-2023].
- [57] <https://man7.org/linux/man-pages/man1/dd.1.html>, [Online; accessed 18-January-2023].
- [58] <https://httpd.apache.org/>, [Online; accessed 18-January-2023].
- [59] <https://varnish-cache.org/>, [Online; accessed 18-January-2023].
- [60] <https://httpd.apache.org/docs/2.4/programs/ab.html>, [Online; accessed 18-January-2023].
- [61] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.
- [62] https://www.ntop.org/products/packet-capture/pf_ring/, [Online; accessed 18-January-2023].
- [63] 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.
- [64] <https://github.com/pktgen/Pktgen-DPDK>, [Online; accessed 18-January-2023].
- [65] <https://github.com/cisco-system-traffic-generator/trex-core>, [Online; accessed 18-January-2023].