xdpcap: XDP Packet Capture

Stefan Lachnit, Sebastian Gallenmüller*, Dominik Scholz*, Henning Stubbe*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: stefan.lachnit@tum.de, {gallenmu | scholz | stubbe}@net.in.tum.de

***Abstract*—Xpress Data Path (XDP) is a Linux kernel feature that allows high performance packet processing using eBPF programs which are executed before the normal network stack. This however prevents tools like tcpdump from capturing all traffic. Xdpcap is a recently released network capturing program, which uses filters compiled to eBPF and a hook in the tested XDP program to capture packets even if they are dropped by the XDP program.**

This paper explains how xdpcap is implemented and presents benchmarks, which compare xdpcap to tcpdump. We show that xdpcap is not able to achieve the same capturing bandwidth as tcpdump and should thus be used for debugging and capturing only when packets dropped or forwarded by an XDP program are of interest.

***Index Terms*—XDP, eBPF, packet capture**

1. Introduction

The ability to capture and filter network packets is an important aspect of debugging network applications. To assess and benchmark the performance of these applications, the recording of large amounts of traffic is needed. Xpress Data Path (XDP) is a Linux kernel feature, which allows users to run small programs to modify, pass, drop or redirect incoming network packets before they are processed by the rest of the networking stack. Using XDP yields performance benefits in applications like Firewalls [1] and DDoS mitigation [2] compared to other solutions like iptables. Additionally, when using XDP, traditional network capturing tools like tcpdump are not able to record all packets, because they could be dropped or modified before they reach the regular network stack. To solve this issue, Cloudflare developed xdpcap [3], a tool which can capture packets (filtered by a user specified expression) directly from an instrumented XDP program.

We performed benchmarks to test the performance of xdpcap and tcpdump by capturing generated test traffic and analyzing how many packets could be recorded. This paper describes these tests and evaluates their results.

The paper is structured in the following way. Section 2 describes the features used by xdpcap and gives an overview of how it is implemented. In Section 3 related work is presented, which discusses XDP and network capturing. After the software and hardware, which was used to benchmark xdpcap and tcpdump, is described in Section 4, the measured data is presented and evaluated in Section 5. In the last section the results are summarized and a conclusion is presented.

2. Background

This section describes the basic concepts of the Linux kernel features used by xdpcap and explains how xdpcap works.

2.1. cBPF and eBPF

Berkeley Packet Filter (BPF; later renamed to classic BPF (cBPF)) is a feature in the Linux kernel which is designed to allow high performance network packet filtering in kernel space. It introduced a virtual machine (VM) that allows users to attach small programs to a network interface, which can parse incoming packets and decide if they should be copied to userspace. One of its primary use cases is the tool tcpdump, which is used for filtering and capturing network traffic for measurements and debugging. It allows the user to specify filtering expressions which are then compiled to a BPF program. Matching packets are copied to userspace where they are stored to a file or parsed and printed [4].

In kernel version 3.15, extended BPF (eBPF) was introduced, which improved the original concept by modifying the VM to allow more complex programs and adding new features. eBPF programs are no longer limited to packet filtering and can now process events in other parts of the kernel. Additionally, the possibility to store persistent data by using maps, which can also be accessed in userspace, and the ability to call kernel helper functions were added. A special type of map allows eBPF programs to dynamically call other eBPF programs, with the limitation that the control flow cannot return to the original code (tail call). To ensure high performance, eBPF programs are compiled to machine code using a just-in-time (JIT) compiler. Since the eBPF program runs in kernel space, it is important to ensure the security of the executed program. This is done by a verifier, which statically analyzes a program before it is attached (e.g., prevents infinite loops, checks if memory accesses are in a valid range) [5].

2.2. XDP

Xpress Data Path (XDP) adds a hook to the Linux network stack, which can be used to run an eBPF program (XDP program) for every packet at the earliest possible moment after it is processed by the driver of the network card. On supported drivers, it is run in the context of the driver or can even be offloaded to specialized hardware on the NIC [6]. The eBPF program has access to the raw

doi: 10.2313/NET-2021-05-1_09

packet data and can parse and possibly modify it. It is also possible to add additional metadata to a packet. The action which is applied to the packet can be specified by the return code of the program. A packet can be dropped, passed on to the normal network stack, re-transmitted from the same interface or redirected (e.g., to a different network interface or to a userspace socket) [7].

2.3. xdpcap

Because XDP programs can modify or drop packets before they reach the Linux network stack, traditional packet capture tools like tcpdump are not able to record all traffic. Cloudflare recently released the tool xdpcap with the goal to recreate tcpdump for applications using XDP.

To achieve this, they wrote a compiler that transforms a cBPF program into equivalent eBPF bytecode. The cBPF code is generated from a user-specified filter expression by libpcap, which is also used by tcpdump.

Packets are captured and filtered by this additional XDP program (filter program), which is executed after all other processing steps of the original XDP program are completed. To be able to dynamically start capturing and change filters without removing the original XDP program, the filtering code is executed as a tail call from the original XDP program. This requires manual modification of the original program by adding a hook (a map of filtering XDP programs generated from the filter expression) and replacing returns by tail calls. To allow the execution of the originally specified action (without returning to the original program), xdpcap generates multiple filter programs with every possible return code hard-coded and chooses the matching program when executing the tail call.

When a filter program matches a packet, it has to be transferred to userspace. This is done by generating a perf event using the eBPF helper function *perf_event_output*, which can contain the packet data and additional metadata. Perf events are part of the Linux kernel, which are normally used for profiling and tracing. In userspace, the xdpcap tool creates a ring buffer where the data created by this perf event is put into. When this buffer is filled to a specified number of bytes (by the parameter watermark), it is read by this tool and printed or output to a file [3].

3. Related Work

XDP has been used in many applications, which require high performance packet processing. Firewall rules, which are faster than existing solutions using iptables [1] [8], efficient DDoS mitigation [2] and an XDP based L4 load balancer [9], are examples for such applications.

For capturing network traffic, different approaches exist. The most common tool for debugging and capturing is the software-based tool tcpdump [4]. Capturing tools, which bypass the Linux network stack, are capable of achieving a capturing rate of up to 120 Gbit/s [10] on commodity hardware. Additionally, hardware-assisted capturing based on FPGAs [11] or commercially available capturing hardware (e.g. Endace DAG cards [12]) allows recording of traffic at high data rates and precision.

TABLE 1: hardware setup

	loadgen	**DuT**
OS	Debian Buster	Debian Buster
Kernel	4.19.0-12-amd64	4.19.0-12-amd64
CPU	Intel Xeon E5-2620 v3	Intel Xeon E5-2630 v4
RAM	32 GB	128 GB
NIC	Intel 82599ES 10G	Intel 82599ES 10G

4. Experiment Setup

To measure and compare the performance of xdpcap in a reproducible way, benchmarks were performed on a hardware testbed managed by pos (plain orchestrating service) [13]. This tool manages all test servers using an orchestrating server, which handles the allocation of servers and the execution of benchmarks. Using a script, the required servers can be rebooted and set up automatically. Additionally, the execution of benchmarks and the collection of results are handled by this script. Since all the test servers run live systems in RAM, the required configuration is done automatically every time the benchmarks are executed [13].

The hardware setup of the servers, which were used for the benchmarks in this paper is described in Table 1. The layout of the benchmark servers is presented in Figure 1. Both servers are connected by a 10 Gbit/s connection and managed by an orchestrating server running pos. One of the servers acts as a load generator sending traffic to the other server, which is running xdpcap or tcpdump to capture this traffic.

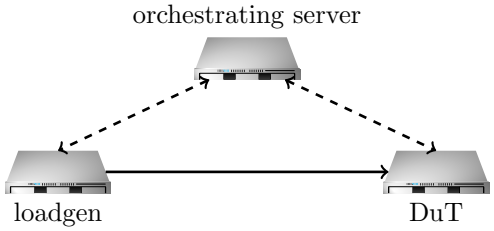

Figure 1: testbed

To generate test traffic on the loadgen server, Moongen [14] was used. It uses DPDK [15] to bypass the Linux network stack and generate up to 10 Gbit/s traffic on a single CPU core with high precision. The packets can be dynamically created and modified by a Lua-script, which controls the data of each generated packet [14]. The benchmarks in this paper use a modified version of the layer2 example script [16] to create Ethernet traffic with adjustable packet size at a specified bandwidth. For testing how filtering of traffic influences the performance of the tested capturing tools, the script was modified to change the ethertype of every given number of packets to a different value (0x1111 instead of 0x1234). The device under test (DuT) can then filter based on this field.

The other host was set up to capture the packets sent by the loadgen server. Multiple tests with both xdpcap and tcpdump were performed. For the benchmarks of xdpcap an XDP program with two functions (drop all traffic, pass all traffic) was added to the network interface, which is connected to the load generator. This program was modified to contain an xdpcap hook and tail calls, which

doi: 10.2313/NET-2021-05-1_09

are required for capturing using xdpcap. Additionally, the network interface had to be set to promiscuous mode to be able to capture packets, which do not match the MAC address of the network interface. When using tcpdump with default settings this happens automatically. Both tools write the incoming data into a pcap file, which is later analyzed using the command *capinfos* to collect performance metrics (number of packets, packet rate, captured bandwidth). All benchmarks performed the capturing for 40 seconds.

The results of both servers (analyzed packet capture, output of Moongen) are then uploaded to the orchestrating server where they were evaluated using an interactive Jupyter notebook.

5. Evaluation

In this section, three benchmarks which were performed using the setup described in Section 4 are presented.

5.1. Maximum Bandwidth

To analyze the maximum achievable capturing bandwidth using both tested tools, benchmarks without filtering expressions were performed. These tests were run with a packet size of 64 B. The bandwidth of the generated Ethernet traffic was scaled from 100 Mbit/s to 3 Gbit/s. For the benchmarks using xdpcap, both an XDP program, which drops all packets and a program which passes all received packets were tested.

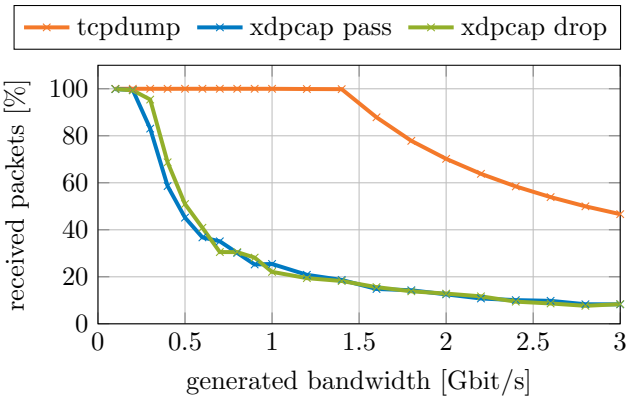

Figure 2: captured packet percentage

Figure 2 shows how many percent of the generated packets could be recorded on the DuT. Tcpdump was able to capture nearly all traffic up to 1.4 Gbit/s. Xdpcap only captured all packets for speeds lower than 200 Mbit/s. The results for xdpcap with different XDP programs (drop all packets, pass all packets) were nearly identical.

Figure 3 shows results of testing the impact of different packet sizes by additionally using 128 B packets. It plots the captured packet rate for transmitted packet rates from 0.1 million packets per second (Mpps) to 5.8 Mpps. Tcpdump was able to capture nearly all traffic up to 2.5 Mpps for a packet size of 64 B. When more traffic was generated, it was only able to record the same 2.5 Mpps and dropped the rest. When using packets with a size of 128 B the maximum number of packets which could be

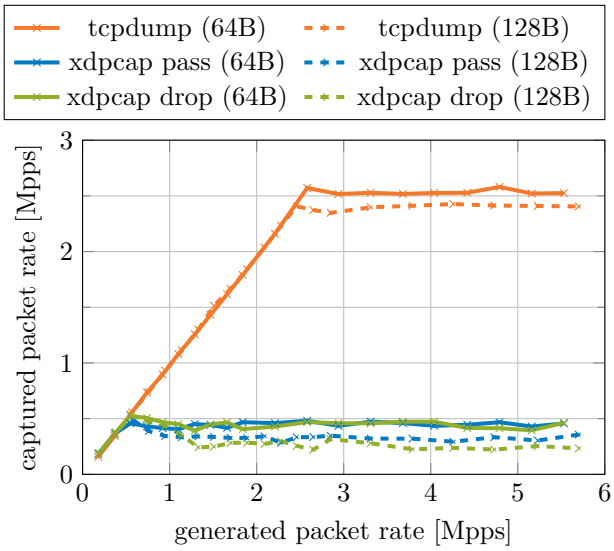

Figure 3: captured packet rate

captured only decreased by approximately 5 %, indicating that the capturing performance of tcpdump is mostly dependent on the number of captured packets and not on recorded bandwidth. Xdpcap showed similar results, but was only able to capture all packets at rates less than 0.5 Mpps with 64 B packets. The achievable packet rates of dropping or passing all packets in the XDP program were nearly identical for 64 B packets. Increasing the packet size to 128 B decreased the possible capturing speed by about 30 % when using xdpcap.

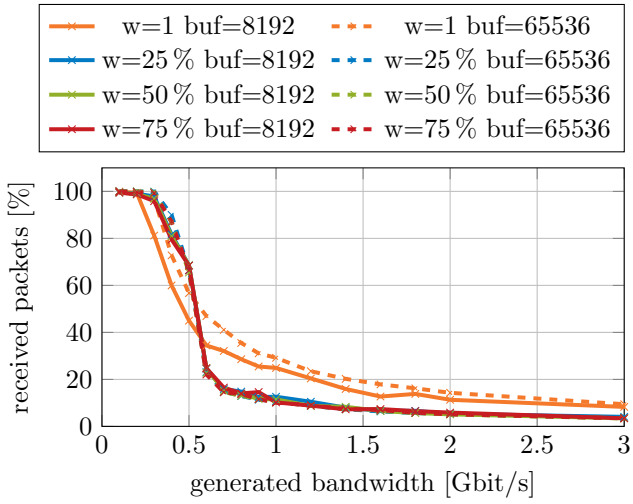

Figure 4: xdpcap parameters: capturing percentage

5.2. Xdpcap Parameters

The userspace xdpcap program offers parameters to tune the buffer size and the "perf watermark", which is used to specify when packets are read from the perf ring buffer. To test the impact of these settings, the percentage of captured packets for different combinations of parameters was tested. The generated bandwidth was scaled from 100 Mbit/s to 3 Gbit/s. The buffer size was set to (the default of) 8192 B and 65 536 B. The watermark was set

doi: 10.2313/NET-2021-05-1_09

to 1 (default: transfer immediately), 25 %, 50 % and 75 % of the buffer size.

Figure 4 shows the results of these tests. When the generated bandwidth is low (less than 200 Mbit/s), increasing the watermark value allowed xpdcap to capture more packets than with the default configuration. For bandwidths above 500 Mbit/s of test traffic, the best result was achieved when transferring packets immediately (watermark 1). Increasing the buffer size to 65 536 B with a watermark of 1 increased the number of captured packets by approximately 10 %. All other values of the watermark and other buffer sizes result in nearly identical or slower capturing speeds.

5.3. Filtered Traffic

To evaluate a more realistic measurement and benchmark scenario, where only a small part of the received traffic is of interest, we tested how filtering the incoming traffic would impact the ability to capture all (matching) packets at high data rates. For this, the modified Moongen script described in Section 4 was used to generate Ethernet traffic and set the ethertype of every 1000th packet to a different value. Only this traffic was recorded by specifying the filtering expression *"ether proto 0x1111"*. The packet size was set to 64 B and the generated bandwidth was scaled from 500 Mbit/s to 6 Gbit/s.

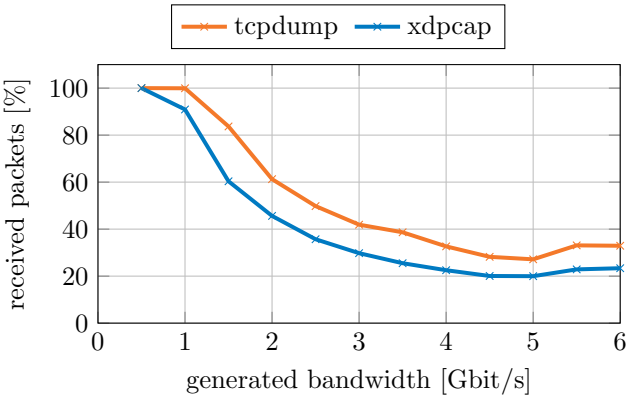

Figure 5: filtered packet capture rate

Figure 5 shows the results of this benchmark. When only capturing a small percentage of the traffic, xdpcap was able to capture all matching packets at rates less than 500 Mbit/s. This is a significant increase from the 200 Mbit/s which could be recorded when no filter was applied. The maximum recording bandwidth of tcpdump decreased compared to the test in Section 5.1. All packets could only be recorded for generated traffic of 1 Gbit/s or less. For every amount of generated traffic, tcpdump captured about 15 % more traffic than xdpcap.

6. Conclusion

The results of the presented benchmarks show that in the measured test scenario, xdpcap performs significantly worse when capturing all packets. Small improvements can be achieved by increasing the ring buffer size and keeping the watermark at the default value. When applying filters, tcpdump was affected more than xdpcap, but still yielded better performance.

Because of this, we conclude that xdpcap should not be used as a tcpdump replacement. However, when used for its intended purpose of debugging or monitoring existing XDP programs, it can be applied where packets are processed before tools like tcpdump can capture them.

References

[1] 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.

[2] "L4Drop: XDP DDoS Mitigations," https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/, accessed: 2020-12-29.

[3] "xdpcap: XDP packet capture," https://blog.cloudflare.com/xdpcap/, accessed: 2020-12-01.

[4] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93. USA: USENIX Association, 1993, p. 2.

[5] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3371038

[6] "Netronome Agilio SmartNICs," https://www.netronome.com/products/agilio-software/agilio-ebpf-software/, accessed: 2021-01-09.

[7] 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

[8] A. Deepak, R. Huang, and P. Mehra, "ebpf/xdp based firewall and packet filtering," in *Linux Plumbers Conference*, 2018.

[9] "katran," https://github.com/facebookincubator/katran, accessed: 2021-01-03.

[10] P. Emmerich, M. Pudelko, S. Gallenmüller, and G. Carle, "Flowscope: Efficient packet capture and storage in 100 gbit/s networks," in *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, 2017, pp. 1–9.

[11] Y. E. Kwasi and R. Rojas-Cessa, "High-resolution hardware-based packet capture with higher-layer pass-through on netfpga card," in *2014 23rd Wireless and Optical Communication Conference (WOCC)*, 2014, pp. 1–6.

[12] "endace," https://www.endace.com/, accessed: 2021-01-03.

[13] S. Gallenmüller, D. Scholz, F. Wohlfart, Q. Scheitle, P. Emmerich, and G. Carle, "High-performance packet processing and measurements," in *2018 10th International Conference on Communication Systems Networks (COMSNETS)*, 2018, pp. 1–8.

[14] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 275–287. [Online]. Available: https://doi.org/10.1145/2815675.2815692

[15] "DPDK," https://www.dpdk.org/, accessed: 2020-12-27.

[16] "Moongen l2-load-latency example," https://github.com/emmericp/MoonGen/blob/525d9917c98a4760db72bb733cf6ad30550d6669/examples/l2-load-latency.lua.