Monitoring Network Flows in Containerized Environments

Matteo Repetto[1(✉)] and Alessandro Carrega[2]

[1] Institute for Applied Mathematics and Information Technologies (IMATI), CNR, Genoa, Italy
matteo.repetto@ge.imati.cnr.it
[2] S2N National Lab, CNIT, Genoa, Italy
alessandro.carrega@cnit.it

Abstract. With the progressive implementation of digital services over virtualized infrastructures and smart devices, the inspection of network traffic becomes more challenging than ever, because of the difficulty to run legacy cybersecurity tools in novel cloud models and computing paradigms. The main issues concern *i*) the portability of the service across heterogeneous public and private infrastructures, that usually lack hardware and software acceleration for efficient packet processing, and *ii*) the difficulty to integrate monolithic appliances in modular and agile containerized environments.

In this Chapter, we investigate the usage of the extended Berkeley Packet Filter (eBPF) for effective and efficient packet inspection in virtualized environments. Our preliminary implementation demonstrates that we can achieve the same performance as well-known packet inspection tools, but with far less resource consumption. This motivates further research work to extend the capability of our framework and to integrate it in Kubernetes.

Keywords: Network flow monitoring · Cloud computing · eBPF · Cloud-native applications

1 Introduction

With the growing adoption of cloud technologies and Internet of Things (IoT) in the implementation of digital services, monitoring and inspection tasks for security purposes are becoming more challenging, because legacy cybersecurity appliances might not be straightforward to deploy and operate in these environments. Typical issues include limited computing and memory resources, unavailability of software and hardware acceleration frameworks, heterogeneity of the execution environments, and cloud-native software paradigms. For instance, in a Kubernetes environment, monitoring files in a different container of the same pod is tricky, and becomes unfeasible in different pods. Similarly, the networking model is based on a mesh pattern instead of common bus, and the implementation of the Container Network Interface (CNI) [8,14] does not allow to inspect all the traffic from a single location.

© The Author(s) 2022
J. Kołodziej et al. (Eds.): Cybersecurity of Digital Service Chains, LNCS 13300, pp. 32–55, 2022.
https://doi.org/10.1007/978-3-031-04036-8_2

Packet inspection is a typical problem in the domain of cybersecurity, although it is really challenging in high-speed networks. To overcome this issue, hardware and software acceleration is often used, together with sampling techniques to monitor network packets at line speed, but these artifacts are usually unavailable in cloud infrastructures and IoT devices. Moreover, they usually provide many monitoring, inspection, and transformation features, resulting in a non-negligible resource footprint. This overhead is perfectly acceptable for monitoring big infrastructures for a large number of threats, but it becomes unnecessarily overwhelming when the analysis is restricted to threats for a specific service. This is the case, for instance, of the collection of basic features to apply Machine Learning algorithms [16].

We believe that recent advances in code augmentation techniques for the Linux kernel, represented by the enhanced Berkeley Packet Filter (eBPF), have a great potential to create packet inspection processes for cloud native applications that are efficient, portable across infrastructures, and programmable. In this respect, we investigated the feasibility and performance of implementing network flow monitoring with eBPF in virtualized services. For this purpose, we designed bpfFlowMon, a simple tool that collects common statistics that are necessary for flow analysis. The preliminary implementation considers the most common use-cases, by parsing IPv4/6, ICMP, TCP, and UDP headers; future releases will support additional protocols and will provide native integration with Kubernetes. Our work represents a valuable contribution to the implementation of security agents for the GUARD framework, since bpfFlowMon fits containerized and serverless computing environments better than existing tools while basically providing the same set of information elements (namely, traffic characteristics and measures). This extends the scope of detection services that make use of traffic features, for instance the MLDDoS detector [16].

Our approach only leverages the eBPF technology which is already built-in in any recent Linux kernel (4.19 and 5.x), which is the common options for all installations. This ensures predictable performance in environments provided by different providers, hence giving more freedom in the deployment and migration processes. We carried out extensive performance evaluation of our tool, and compared it with well-known appliances, namely nProbe and Zeek. Our evaluation shows that CPU usage and memory footprint of our implementation are almost one order of magnitude lower than the selected alternatives, while reporting the same level of information.

The rest of the Chapter is organized as follows. We provide a brief understanding of the flow monitoring problem in Sect. 2, including the tools taken for reference and the usage of eBPF for monitoring and inspection. Then, we describe the bpfFlowMon tool in Sect. 3 and report performance evaluation and comparison in Sect. 4. We also discuss the limitations and open issues of our approach in Sect. 5. We also provide a short survey of the current scientific literature about the usage of eBPF-based tools for security purposes in Sect. 6. Finally, we give our conclusion in Sect. 7, which includes the road away.

2 Flow Monitoring

Flow monitoring provides a high-level view on network traffic, by grouping packets into logical communications (i.e., "flows"), which can be used for management, debugging, and security purposes of both local and wide area packet networks. Packets can be mapped to flows according to many characteristics (e.g., field values in protocol headers, ingress/egress interface, next hop - see RFC 3954 and 3917 [3,15]); a very common approach is the identification of flows based on the network and transport endpoints, namely the 5-tuple composed of source network address, destination network address, protocol, source transport port, and destination transport port. For each flow, aggregate statistics are measured, as total number of bytes/packets exchanged, average rate of bytes/packets, duration of the flow, average latency and jitter, relevant flags and field values seen.

Despite the relative simplicity of this definition, tracking network flows is a cumbersome process because all packets must be parsed in order to extract the relevant fields and to detect the begin/end of the flow. In fact, performing this operation at line speed is extremely challenging especially in case of small packets and link rates of 10 Gbps and above. For this reason, typical implementations usually rely on dedicated hardware, sometimes integrated in the same network devices; in addition, sampling is often used in network equipment with fast interfaces (usually above 1 Gbps). Flow monitoring appliances can directly visualize data to users or forward it to remote processes, where further processing may occur; in this second case, there are specific protocols to exchange this information (NetFlow [3], IPFix [4], sFlow [13]).

2.1 Existing Tools and Limitations

There are different tools available for packet inspection and flow analysis. Some of them mostly work as "simple" network probes that just collect the necessary measures and make them available for further processing, either locally or to a remote location. Besides proprietary implementations in network equipment, nProbe[1] is a pure software flow exporter compatible with NetFlow/IPFIX. It has Deep Packet Inspection (DPI) capabilities and support for over 250 applications; however, the set of measurements and data that can be collected are hard-coded in the application and cannot be extended for specific needs. Unfortunately, nProbe is not open-source, even if there is a free version with some limitations on capabilities and usage.

Other tools provide a complete framework for flow and packet analysis. *Zeek*[2] (formerly Bro) is fully open-source and it provides a flexible framework that is easy to extend by a scripting language. Zeek does not export flow descriptors, but it only creates compact transaction logs, file content, and fully customized output, suitable for manual review analyses by a Security and Information Event Management (SIEM) system.

1 https://www.ntop.org/products/netflow/nprobe/.
2 https://zeek.org/.

Packet processing at line rate is a challenging task, especially on general-purpose hardware, because of the delay introduced by packet copies, interrupt handling and context switches. Many monitoring tools improve their efficiency by using software acceleration frameworks that bypass the built-in kernel networking stack and give direct access to hardware queues (e.g., PF_RING, Netmap, DPDK, OpenOnLoad). Both nProbe and Zeek, for instance, use PF_RING to increase performance. Unfortunately, all these technologies are not par of vanilla kernel releases and require modification of the running kernel; therefore, both the availability and the effectiveness of such technologies in virtualized environments is questionable, especially in the case of public infrastructures.

Even if the volume of traffic within a virtualized service could be processed without acceleration frameworks, the deployment of traditional network monitoring appliances might not be efficient or straightforward, especially in container runtime environments, where they should be deployed in every pod for ensuring full visibility. In such scenarios, it is important to reduce as much as possible resource consumption (CPU, memory and storage), to make the deployment process faster and to decrease the cost in case public cloud infrastructures are used.

2.2 The Extended Berkeley Packet Filter

Monitoring the behavior of an Operating System and its applications has always been a hot topic both for performance and security reasons. User-space applications can easily monitor the integrity of files, the behavior of the system users and other applications (by looking at logs). However, support from the kernel is necessary to gain visibility over its operation, including the processing of network packets and the execution of system calls and other internal functions. Kernel monitoring has been based for long on third-party extensions; unfortunately, this approach requires notable effort to update and re-compile additional modules every time the kernel changes, not to mention the difficulty to support many different kernel versions and the risk of introducing instability.

The extended Berkeley Packet Filter (eBPF) was introduced as a flexible and *programmable* mechanism to monitor kernel activity. It extends the scope of original BPF (sometimes denoted as "classical" BPF – cBPF) from network packets to software functions as well, by introducing a common interface to easily access multiple kernel subsystems. Its implementation consists of an embedded virtual machine that executes a specific set of instructions, which scope is deliberately restricted to avoid harming the kernel itself. eBPF gives access to multiple data sources present in the kernel, by attaching eBPF programs to "probes" (kprobes, uprobes) and "tracepoints" (USDT,[3] kernel tracepoints, lttng-ust[4]), as well as to "network hooks" (XDP,[5] TC[6]). Differently from other tools that use the same

[3] Userland Statically Defined Tracing.
[4] Linux Trace Toolkit Next Generation Userspace Tracer.
[5] eXpress Data Path.
[6] Traffic Control.

hooks (e.g. LTTng or SystemTap), a key added-value of eBPF is that it does not require any additional kernel modules. This allows augmenting the running kernel with custom code in a safe way, without the need to load additional modules or to recompile the whole kernel.

The execution of eBPF programs is triggered by specific events of each hook (i.e., the reception of a network packet for XDP/TC, the invocation of a function for other software probes) and their size is limited to a few instructions for safety and performance reasons. As a matter of fact, they are only expected to collect raw measurements and events, as well as to perform enforcement actions. Such data can be stored in shared data structures (*maps*) between the kernel and the user-space; further processing and delivery to intended recipients can therefore be implemented more effectively in userland applications. For more complex tasks, which exceed the limit of the stack size, a *tail call* mechanism is available that allows to chain multiple programs while sharing the context, in order to perform the task in multiple stages. Interestingly, eBPF can trace kernel functions but cannot drop their execution, whereas this is instead possible for network packets.

A development toolchain is available that allows writing programs in C code, which are then compiled into the eBPF assembly. Programs are then verified and compiled into bytecode only at loading time, to ensure that they are loop-free and do not access data outside of their boundary. Developing eBPF applications is not trivial for beginners, but common operations are available through more user-friendly frontends, like the BPF Compiler Collection (BCC[7]) and the bpftrace interpreter.[8]

2.3 eBPF for Monitoring, Inspection, and Enforcement

Though it was originally conceived for investigating performance issues, eBPF is also valuable for security monitoring, network filtering and for tracing, profiling, and debugging the software (see the related work discussed in Sect. 6). Many commercial and open-source frameworks have already integrated eBPF, including Suricata,[9] Sysdig Falco.[10]

eBPF programs can also be used to better support the integration of security features in containers. For observability, ntop and InfluxData have partnered to offer eBPF monitoring for containers, while Netdata is offering out-of-the-box eBPF monitoring for system and application monitoring.[11] Differently from these frameworks that focus on observability, Cilium[12] uses eBPF to provide secure network connectivity and load balancing, leveraging the ability of eBPF to filter and drop packets based on rules. Kubectl-trace is a Kubernetes command line

[7] https://github.com/iovisor/bcc.
[8] https://github.com/iovisor/bpftrace.
[9] https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html.
[10] https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/.
[11] https://containerjournal.com/topics/container-management/using-ebpf-monitoring-to-know-what-to-measure-and-why/.
[12] https://cilium.io/.

front end for running bpftrace across worker machines in Kubernetes cluster. However, to the best of our knowledge, there is no solution yet to activate metric capturing during the deployment of a containerized application with Kubernetes or equivalent [2].

3 A New Flow Monitoring Tool: bpfFlowMon

To investigate the feasibility, limitations, and performance of monitoring network flows by eBPF, we implemented a tool named bpfFlowMon[13] composed of the following components:

- an eBPF *Flow Inspector* for the TC hook (namely, the queue management system associated to each network interface), which builds a raw table of unidirectional flows seen in a specific direction (either ingress or egress);
- a userland *Flow Handling* application that periodically scans the raw flow table, merges and removes unidirectional flows belonging to the same conversation once terminated, and exports them.

Additionally, a *Flow Management* daemon is also present for loading/unloading the eBPF and userland programs. The logical separation between packet inspection and flow handling perfectly fits the design pattern for eBPF programs, leading to an efficient approach. The overall structure of our eBPF-based flow monitoring implementation is sketched in Fig. 1.

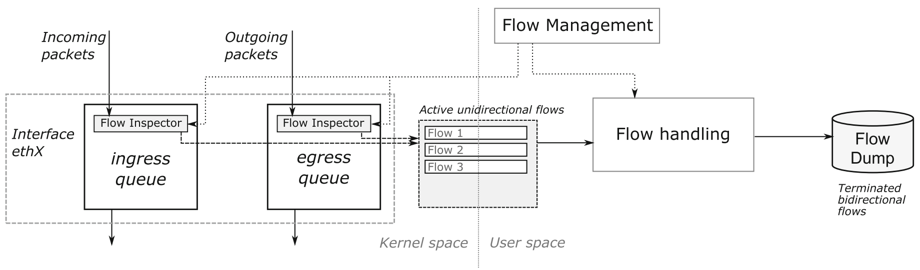

Fig. 1. Software architecture of bpfFlowMon.

The implementation of the Flow Inspector is rather simple, according to the specific nature of eBPF programs. It is basically a parser of network packets, from Ethernet up to TCP/UDP headers. As already noted beforehand, this preliminary implementation was mostly conceived for performance investigation, so the support for additional protocols and deep-packet inspection is left for future extensions. Unfortunately, the code for parsing Ethernet/IP/TCP/UDP headers already consumes all of the small stack available to eBPF programs, and

[13] https://github.com/mattereppe/bpf_flowmon.

this does not include all possible options. Future extensions must address this issue, for example by decomposing the Flow Inspector into different programs for each protocol, and by chaining them with the *tail call* mechanism. Since the number of stacked protocols for each packet is rather limited, this approach should not increase significantly the processing delay. An alternative approach could be to forward the relevant header portion to the userspace, and to parse there the fields, similarly to what happens in the Zeek framework. This would bring more flexibility in defining parsing and aggregation tasks, but would likely increase resource usage, as shown for Zeek in Sect. 4.2.

The Flow Monitoring program follows common practice for flow analysis. Each flow is identified by the 5-tuple <*src_ip_addr*, *dst_ip_addr*, *proto*, *src_port*, *dst_port*>, where the last two parameters are only present where applicable.[14] We point out that this definition identifies "unidirectional" flows; in general, for each *forward-direction* flow <*addr1*, *addr2*, *proto*, *port1*, *port2*> there will always be a *backward-direction* flow <*addr2*, *addr1*, *proto*, *port2*, *port1*> with IP addresses and protocol ports inverted. To keep the implementation simpler, our eBPF program identifies unidirectional flows and stores them in a shared map, but does not merge them. Indeed, looking for a complementary flow at each packet reception would unnecessarily waste time and CPU cycles, because this aggregation is not requested for the computation of any metric.

For each flow, both its identifier and a relevant set of measurements are stored in the shared map. This set includes all the necessary measurements to derive the same NetFlow/IPFIX flow information elements as nProbe for IP and UDP/TCP fields.[15] Other metadata concerning routing information, BGP domains, inspection engine, DNS resolution are not taken into account because they are not retrieved from packet inspection and therefore do not bring significant performance issues; additionally, they are mostly useful in case of wide-area networks and management issues, which is beyond the scope of network analytics for virtualized network services [16,17]. Table 1 summarizes the supported fields at the time of writing; the objective is to make them easily extendable as the work progresses.

The second building block of the bpfFlowMon framework is the Flow Handling utility. This is a C program running in user-space that periodically scans the shared map of unicast flows, merges them, dumps and removes terminated flows. The usage of the C language instead of the BCC framework is mostly motivated by the need for more efficiency, especially with a large number of flows. This is evident when comparing resource usage given in Sect. 4.2 with similar analysis we did on a BCC tool for the detection of steganographic channels in network headers [21].

As common practice, flows are considered terminated either after explicit messages (FIN or RST flags have been seen for TCP) or an inactivity timeout (which works for both TCP, UDP, and ICMP). The choice of the inactivity

[14] The *src_port* field is used with a different meaning for ICMP protocol, to identify the message type.

[15] https://www.ntop.org/guides/nprobe/flow_information_elements.html.

Table 1. Information elements for bpfFlowMon.

Name	Context	Meaning
first_seen	General	Epoch of the first packet of this flow (ns)
last_seen	General	Epoch of the last packet seen so far (ns)
jitter	General	Cumulative delays between packets
pkts	General	Cumulative number of packets
ifindex	General	Capture interface
version	IP	Version (4/6)
tos	IP	TOS/DSCP (IPv4) or Traffic class (IPv6)
fl	IP	Flow label (IPv6)
bytes	IP	Cumulative number of bytes
min_pkt_len	IP	Smallest IP packet seen in the flow
max_pkt_len	IP	Biggest IP packet seen in the flow
pkt_size_hist[6]	IP	[0]: pkts up to 128 bytes; [1]: pkts from 128 to 256 bytes; [2]: pkts from 256 to 512 bytes; [3]: pkts from 512 to 1024 bytes; [4]: pkts from 1024 to 1514 bytes; [5]: pkts over 1514 bytes
min_ttl	IP	Min TTL (IPv4) or Hop limit (IPv6)
max_ttl	IP	Max TTL (IPv4) or Hop limit (IPv6)
pkt_ttl_hist[10]	IP	[0]: pkts with TTL $= 1$; [1]: pkts with TTL > 1 and TTL ≤ 5; [2]: packets with TTL > 5 and ≤ 32; [3]: packets with TTL > 32 and ≤ 64; [4]: packets with TTL > 64 and ≤ 96; [5]: packets with TTL > 96 and ≤ 128; [6]: packets with TTL > 128 and ≤ 160; [7]: packets with TTL > 160 and ≤ 192; [8]: packets with TTL > 192 and ≤ 224; [9]: packets with TTL > 224 and ≤ 255
next_seq	TCP	Last sequence number seen (used for computing retransmissions)
last_id	TCP	Last ipv4 identification value for last_seq
cumulative_flags	TCP	Cumulative TCP flags seen in all packets so far
retr_pkts	TCP	Total number of retransmitted packets
retr_bytes	TCP	Total number of retransmitted bytes
ooo_pkts	TCP	Total number of out-of-order packets
ooo_bytes	TCP	Total number of out-of-order bytes
min_win_bytes	TCP	Min TCP window
max_win_bytes	TCP	Max TCP window
mss	TCP	TCP max segment size
wndw_scale	TCP	TCP window scale

timeout affects the accuracy of flow identification. More in detail, short timeouts may result in the same conversation to be seen as two separate flows; on the other hand, long timeouts may result in more conversations merged into a single flow and delay flow reporting.

When two unidirectional flows are merged together, the result is a full conversation; its forward direction is conventionally assumed by taking the source according to the first packet seen. Informative elements (i.e., measures taken by the eBPF program) are kept separate for the forward and backward direction, since this is a typical requirement from analytics and detection algorithms. This represent an extension with respect to, e.g., nProbe, which only provides aggregate metrics for both directions. The current version dumps flows to text files (or to the standard output), either in plaintext or JSON format.

Finally, there is also a Flow Management component, which takes care of loading the eBPF programs and starting the userland utility. It attaches the eBPF program to the ingress and/or egress queue of one or more interfaces, and configures user-space options (polling parameters, filename and folder for dumping, etc.). The purpose is to support different scenarios, where the virtual function acts either as end node or forwarding device. The current implementation of this component facilitates the integration with SysVinit and Systemd for automatically starting and stopping the whole framework.

4 Evaluation

Evaluation of the proposed approach was carried out by comparing both how accuracy and performance vary with respect to existing tools. In the first case, we investigated if and under what conditions there are deviations in the detected flows with respect to nProbe. In the second case, we compared the maximum transfer rate and the overhead in terms of resource consumption, including both CPU time and memory with respect to *i)* the baseline scenario (where no monitoring tool is run), *ii)* to nProbe, and *iii)* to Zeek.

Since our main application target are virtualized services, we performed our experiments in Virtual Machines (VMs). In this way we can account for any limitations that packet acceleration software experiences in virtualized environments. The next step would be the validation with containers, by transforming the current management script into some Kubernetes artifact (e.g., DaemonSet[16] or new operator). However, we do not see any reason to expect different performance in a containerized setup.

Evaluation was carried out under the same testbed topology built in an OpenStack installation: one sender that generates packets, one receiver that performs measurements, one intermediate node that runs the flow monitoring tool. Flows were monitored on both interfaces of the intermediate node, which is a typical condition in case multiple interfaces are present. All nodes ran on the

[16] A Kubernetes DaemonSet is a resource object that ensures that a given pod is instantiated in each cluster node. See the documentation: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.

same hypervisor, 2x Intel Xeon CPU E5-2660 v4@2.00 GHz with 14 cores and hyperthreading enabled, 128 GB RAM, 64 GB SSD storage. The configuration of the three OpenStack servers is reported in Table 2.

For investigating accuracy, we used Pcap traces available from the Internet, which include a mix of ICMP, TCP and UDP traffic. More specifically, we used two network traces of different size designed to test IPFIX/NetFlow, indicated as smallFlows and bigFlows.[17] For performance we generated network packets with iperf3,[18] while varying the main parameters that might affect the inspection process.

Table 2. Configuration of OpenStack servers used for testing.

Node	vCores	vRAM	vStorage
Sender	1	1 GB	16 GB
Receiver	1	1 GB	16 GB
Forwarder	4	2 GB	32 GB

4.1 Accuracy

We define the "accuracy" of a flow monitoring tool as its ability to assign individual packets to the correct flow. This association should not be ambiguous, because it is based on the value of a few header fields. However, wrong classification sometimes occur due to the presence of connectionless protocols without an explicit signaling (UDP, SNMP, ICMP, etc.), abnormal flow termination, as well as delayed or duplicated packets that arrive after the estimated termination of the flow.

We preliminary investigated if and how the flows detected by bpfFlowMon differ from what reported by nProbe. The two tools perform almost the same way, and most differences are only due to different configuration settings. For instance, the flow inactivity timeout, namely the time to wait without seeing any packet before considering the flow closed, largely affects the reporting. With a small value, the risk is that a single flow is seen as two separate flows, when the gap between the transmission of consecutive packets is large enough. On the other hand, with a large value the risk is that two independent flows are seen as the same conversation; however, this is unlikely with TCP, because the end of the connection is also detected by the header flags (both in case of normal closing and aborting/resetting). Besides the inactivity timeout, we also found some differences in the granularity of flows for ICMP and SNMP, the usage of protocol flags and additional header fields to recognize the end of the flow, and the presence of routing options.

[17] Both traces are available at https://tcpreplay.appneta.com/wiki/captures.html.
[18] https://iperf.fr/.

In the following, we summarize our main findings in this respect.

ICMP. Notably, bpfFlowMon uses the ICMP Type field as source port (even if this may look like an abuse of the classification), so it distinguishes between different ICMP flows involving the same couple of peers. Instead, nProbe sets both protocol source and destination ports to 0, hence failing to give indication about the type of ICMP messages.

From our experiments, bpfFlowMon successfully correlates ICMP requests with responses, while the same behavior is not guaranteed for nProbe (we observe several cases where two separate flows are reported).[19]

TCP. Resetting a TCP connection might have side effects, because this operation does not usually imply mutual agreement from the peers. The challenging aspect is that multiple reset packets may be sent, in case the remote peer continues to acknowledge previous packets. Under this (not so uncommon) circumstance, nProbe terminates the flow on the first reset, so any following reset/ack packet is accounted as a different flow. Our implementation partially mitigates this problem, because the userland utility only dumps a flow after scanning the map of unidirectional flows, which happens periodically. This way, any additional (re-)transmission in a short timeframe is counted as part of the previous flow.[20] This largely avoids the incorrect reporting of tiny flows with a few ack/reset packets.

In a similar way, it is possible that duplicated packets are received after the exchange of FIN messages, and get counted as different flows. In this case, the problem was observed for bpfFlowMon, due to the fact that the map of active flows was scanned just after the exchange of FIN messages.[21] The problem could be easily solved by waiting for the inactive timeout before purging the flows, but this brings two drawbacks: i) the dumping is delayed (and this may be a problem, if we are trying to identify malicious flows in real-time), and ii) two consecutive flows might be merged together (the split between user and kernel space tasks does not allow to easily take into account the presence of two consecutive flows with the same 5-tuple).

UDP. We didn't observe relevant differences between nProbe and bpfFlowMon for UDP traffic, but we noticed a potential flaw with source routing. As a matter of fact, both nProbe and bpfFlowMon are transparent to the source routing option, so they are somehow cheated because they only see the "inner" intermediate node and not the final destination of the packet. This means that if only

[19] This is the case, for instance, for packets numbered 108593 and 108755 (according to Wireshark) in the bigFlows trace.

[20] This happens, for instance, for TCP stream number 19032 (as reported by Wireshark), again from the bigFlows trace.

[21] This was observed for TCP stream number 16749 (as reported by Wireshark), again from the bigFlows trace.

part of packets belonging to the same flow are sent with the loose source routing option, whereas the remaining are not, bpfFlowMon reports two separate flows (and nProbe does the same).[22] This, in some way, affects the statistics of traffic towards given destinations.

In this case, we used a third tool to better investigate this problem, namely tshark, the terminal-based version of Wireshark. Interestingly, this tool is able to parse the full range of options and reports a single flow towards the correct destination, as after all Wireshark does too.

SNMP. We found another unexpected behavior in case of SNMP flows. This time, the oddity was found for tshark, which reported two separate UDP flows for the same SNMP conversation, even if also Wireshark grouped the messages in the same flow.[23] We guess this might be due to some interpretation of the SNMP, but we didn't investigate the issue in detail because deep packet inspection is out of scope for our work and our tool behaves well in this scenario.

4.2 Performance and Overhead

Performance and overhead were investigated by considering both the impact on packet transmission as well as resource consumption. We considered both UDP and TCP flows, by varying the main parameters that are expected to affect packet processing:

- packet size and transmission rate for UDP flows;
- maximum segment size (MSS) for TCP streams.

For packet size, we considered 4 values that are representative of the following cases:

- 16 bytes is the smallest value allowed by iperf, and this is the worst condition for packet forwarding;
- 1470 bytes is representative of the biggest Ethernet packets, also accounting for the presence of tunneling in the underlying virtual network;
- 8192 bytes is the reference size for jumbo frames in Ethernet, which is a common situation in all installations;
- 65507 bytes is the maximum size allowed by UDP and the best condition for packet forwarding, but it is only feasible on loopback interfaces (therefore, it can only be used when VMs are running on the same host).

For the transmission rate, we considered a broad range of different load conditions, from 10 Kbps to the unfeasible (at least for our installation) rate of 10 Gbps. For the MSS, we again selected 4 values that this time are representative

[22] This was observed for UDP stream 1164 (according to Wireshark numbering), from the bigFlows trace.

[23] This happens for UDP stream 3604 (again, according to Wireshark numbering), bearing an SNMP conversation from the bigFlows trace.

of: the smallest value accepted by iperf3 (88 bytes), the minimum value that should be used on IP links (536 bytes), the typical value used for Ethernet links (1460 bytes) and jumbo frames (8192 bytes).

For comparison, we took both nProbe and Zeek, briefly introduced in Sect. 2. In addition, we assumed the transmission without flow monitoring tools as the "baseline," which is the best value we can achieve in the target scenario, without the usage of hardware or software accelerators and other modifications of the vanilla OpenStack installation. In case of nProbe, we run the application both with and without PF_RING, to verify software acceleration is ineffective in a standard virtualized setup. Beside that, we note that a single nProbe instance can get packets from multiple interfaces when PF_RING is used; however, two separate instances must be when this module is not used. For completeness, we point out that multiple instances of our eBPF program and Zeek are also run, one for each interface.

To make the comparison fair, we collected the same set of information fields (IP protocol version, IP source and destination addressed, L4 protocol, L4 source and destination ports, flow start/end timestamp, flow duration, forward/backward packets, forward/backward bytes), and we dumped the information to file. Experiments were run for 10 min each, so to mitigate as much as possible interference with other parallel activity, both in the network and within the guest/host systems.

Impact on Packet Transmission. Our analysis focuses on the parameters reported by iperf3, namely the transmission rate, packet error rate and jitter. Figure 2 shows how the measured bitrate at the receiver changes for different settings of packet size and transmission bitrate for UDP flows. There are not meaningful differences between the different mechanisms, even if nProbe with PF_RING performs slightly better in case of bigger packets.

Figure 3 measures the percentage packet loss at the receiver under the same conditions. As expected, packet loss is higher with smaller packet sizes and faster transmission rates. Both bpfFlowMon and nProbe without PF_RING perform almost the same way as the baseline scenario, whereas the behavior of nProbe with PF_RING and Zeek is much more unpredictable. Zeek seems to perform better than other tools almost in all conditions, but it suffers a packet loss close to 100% in case of 65507-byte packets at the fastest transmission rate. We do not have a clear motivation for this anomaly, but for sure it is persistent across many replicas of the experiment.

The last UDP measurement is the average packet jitter, shown in Fig. 4. The variation in the inter-packet delay is negligible for practical applications (below 0.1 ms), and it is likely to be highly affected by external factors (including perturbations in the generator and the receiver). Similarly to previous indexes, there is no tool which wins over all in every condition.

Oddly, in the baseline scenario we achieved worse performance than with flow monitoring, and this is rather unexpected, especially when compared to bpfFlowMon and nProbe without PF_RING. We do not think these are measurement

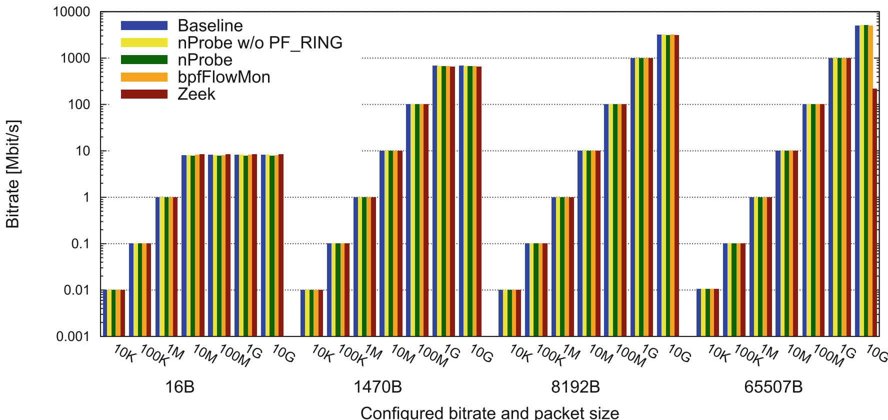

Fig. 2. Measured bitrate at the receiver, while varying packet size and the transmission bitrate for a UDP flow.

errors, since the same behavior is present in all conditions and occurred even when replicating the experiments.

Finally, we show in Fig. 5 the transmission rate achievable by TCP when generating packets for the whole duration of the experiment. Not surprisingly, higher bitrates are possible with larger MSS, because this has a beneficial impact on the TCP flow control mechanism. Again, no meaningful differences are visible between the considered tools, even if in this case the baseline scenario achieves slightly higher bitrate, in line with what expected.

By looking at the performance indexes reported in this Section, we can conclude that our eBPF-based mechanism does not affect packet transmission in a significant way.

CPU Usage. CPU and memory usage are important to understand what is the impact on the operation of other applications. We expect significant differences in the usage of CPU between bpfFlowMon and the other tools, due to the different architectural design. As a matter of fact, Zeek and nProbe are basically user-space applications. The default capture driver for Zeek is libpcap,[24] whereas nProbe builds on the PF_RING module in kernel space, falling back to libpcap if PF_RING is not available. On the other hand, our bpfFlowMon tool leverages in-kernel eBPF programs, hence the usage of CPU from kernel and userspace is expected to be rather different.

This is largely confirmed by the breakdown of CPU usage reported in Figs. 6, 7, 8 and 9 for a UDP flow, with different payload size. Our measurements show much higher CPU usage for user-space (*%user*) in case of nProbe and Zeek, especially for small packet sizes (which is the worst scenario for packet processing),

[24] There are alternative capture drivers that can be used with Zeek, including raw sockets, libpcap, and PF_RING. We only considered libpcap in our study.

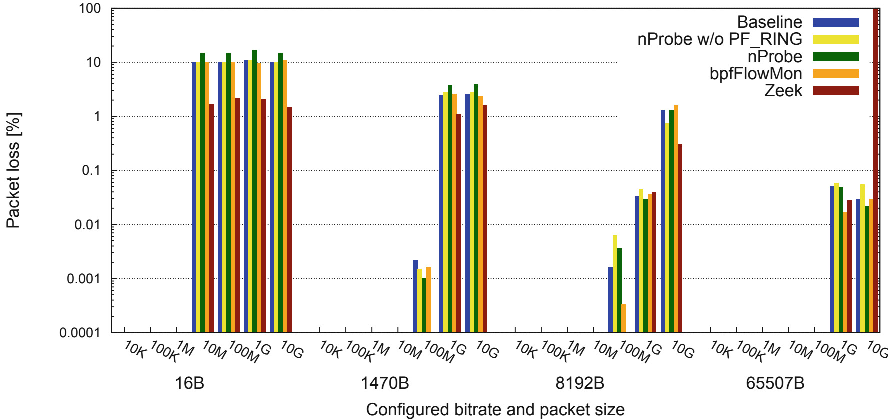

Fig. 3. Measured packet loss at the receiver, while varying packet size and the transmission bitrate for a UDP flow.

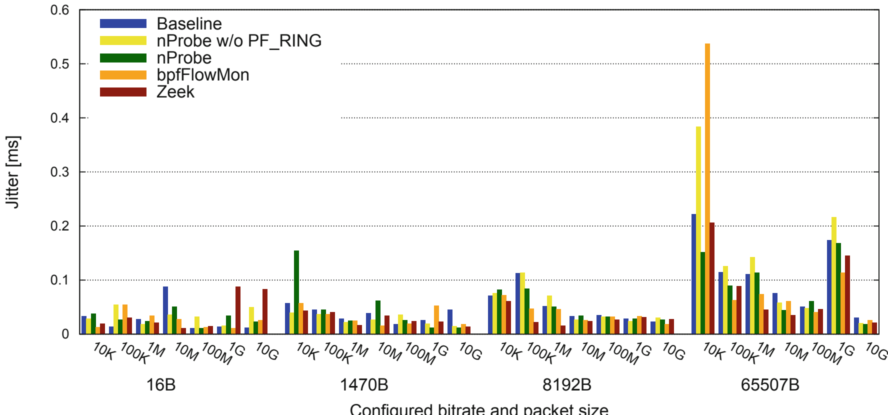

Fig. 4. Measured packet jitter at the receiver, while varying packet size and the transmission bitrate for a UDP flow.

whereas bpfFlowMon always brings a small overhead with respect to the baseline. However, the CPU consumption in user-space is not directly compensated by a corresponding lower kernel usage (*% system*), even in case of low traffic; this can probably be ascribed to the usage of polling in both PF_RING and libpcap. When the traffic increases, the benefit of PF_RING is evident, but the total overhead is still far beyond our eBPF-based tool. Overall, we note that the time spent in other states (*%nice*, *%iowait*, *%steal*) is negligible.

Similar considerations hold in case of TCP, which measurements are shown in Fig. 10.

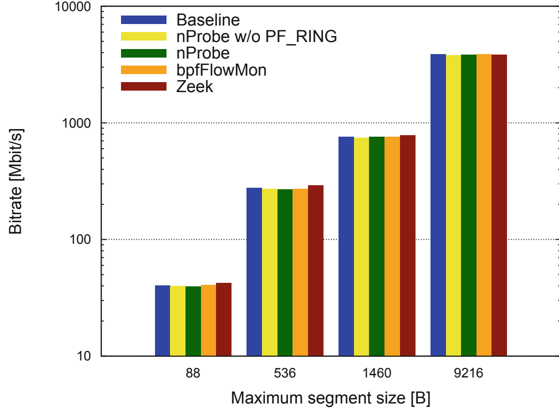

Fig. 5. Measured bitrate at the receiver, while varying the MSS for a TCP flow.

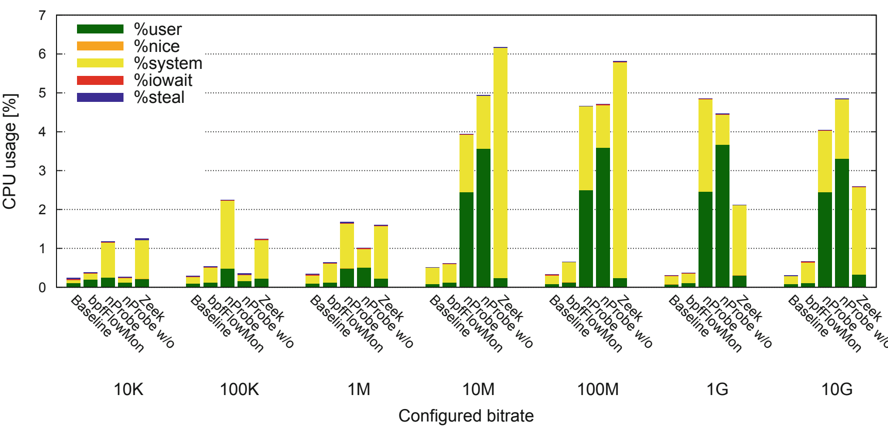

Fig. 6. Cumulative CPU usage measured at the intermediate node for a UDP flow. Payload size is 16 bytes.

Memory Allocation. When comparing the different considered in our investigation, the first thing to consider is the need for additional kernel modules. Indeed, nProbe requires the PF_RING kernel module (which is proprietary and not released as open-source), whereas bpfFlowMon needs the cls_bpf module (which is part of the vanilla kernel); Zeek does not require any kernel extension (even if PF_RING support is available as option). We note that the size of the PF_RING module in our configuration is 737.280 KB, while the cls_bpf is only 24.576 KB.

Going on, the next step is memory consumption in user space (not including, therefore, the kernel modules). We consider the Virtual Memory Size (VMS), the Resident Set Size (RSS), the Proportional Set Size (PSS), and the Anonymous share (Anon). The first corresponds to the overall address space allocated by

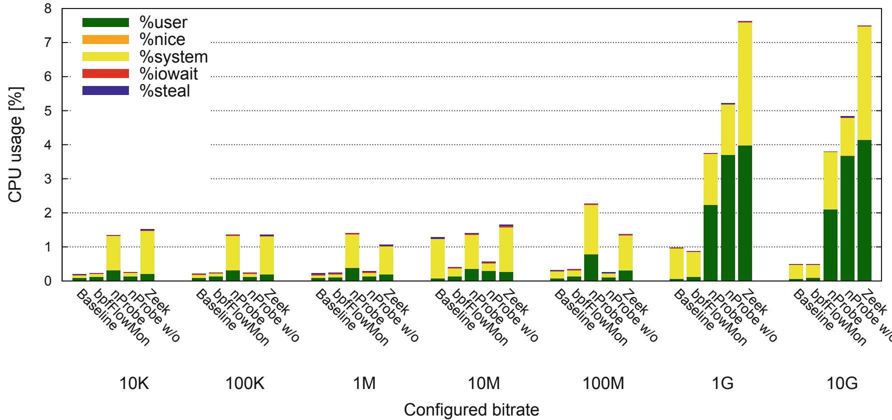

Fig. 7. Cumulative CPU usage measured at the intermediate node for a UDP flow. Payload size is 1470 bytes.

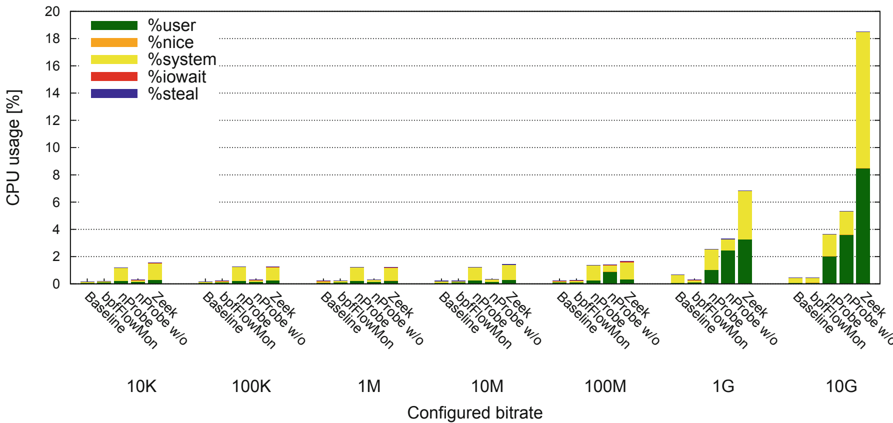

Fig. 8. Cumulative CPU usage measured at the intermediate node for a UDP flow. Payload size is 8192 bytes.

the program, the second is the real size of physical memory allocated (including shared libraries), the third is again the physical memory but with proportional attribution of shared libraries (namely, their memory allocation is divided by the number of programs that use them), and the forth is memory mappings not backed by a file. The amount of memory is further broken down according to the corresponding mapping type (e.g., file, library, stack, heap, process, socket, etc.), to give the more picture of memory usage. Figure 11 shows that Zeek has the largest memory address space (VMS) of all considered tools, even if its real memory usage (RSS) is comparable with nProbe, especially when the latter runs without the PF_RING module. The reason is that nProbe can only monitor one

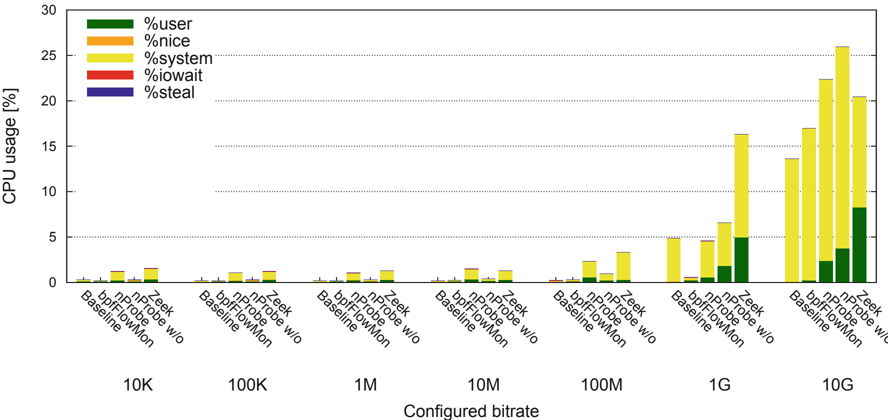

Fig. 9. Cumulative CPU usage measured at the intermediate node for a UDP flow. Payload size is 65507 bytes.

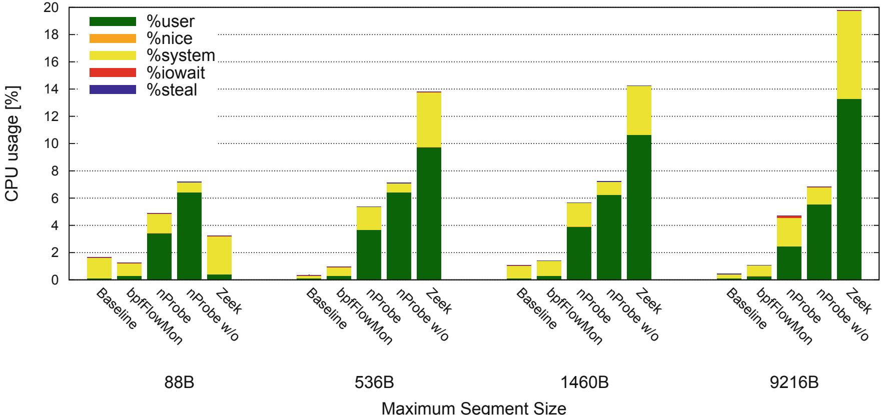

Fig. 10. Cumulative CPU usage measured at the intermediate node, for a TCP flow.

network interface, so to keep the comparison functionally equivalent with the other cases we have to run two parallel instances, one for each interface. This is necessary to see all traffic intended to the host. If additional network interfaces are monitored, the memory usage will scale accordingly. The same issue applies to Zeek: in this case, we do not have to manually start two instances, but they are anyway launched when multiple interfaces are selected in the configuration file. Memory consumption of our tools is negligible with respect to nProbe and Zeek (only a few kilobytes); actually, it cannot even be seen due to the scale of the graph. We remark that the larger memory required by Zeek and nProbe is also due to the many features available in these applications. In general, we

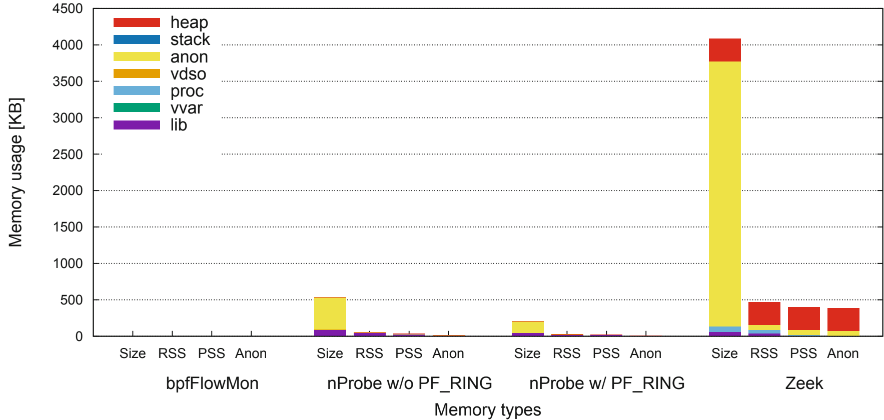

Fig. 11. Memory allocation for the different user-space tools.

do not say this allocation is disproportionate to what the application does, but simply it is not efficient when only simple measurements are needed.

5 Limitations of the Proposed Approach

The eBPF framework represents an efficient alternative for monitoring and inspection than existing tools, especially for virtualized environments. However, both our preliminary implementations and the framework in general are not exempt from limitations that devote further investigation in the future.

One main drawback of eBPF programs is the limited stack size available, which limits the number of functions and instructions. Though the original stack size of 512 bytes may be extended in future releases, parsing several network layers remains a challenging issue, especially when variable options are present. We were able to parse common fields in IPv4/IPv6/UDP/TCP headers, but inspecting all TCP options requires at least to drop support for one IP version. To overcome this limitation, next releases of bpfFlowMon will leverage the *tail call* mechanism, which allows to invoke a chain of eBPF programs. This approach would bring the opportunity for deep packet inspection, and it is also expected to improve the flexibility and composability of the system, as new parsers can be easily added to the base framework.

Another well-known drawback is the eBPF verifier that, on the one hand ensures there are no loops and the program always terminates, but on the other hand still has many limitations, including high rate of false positives, poor scalability, and lack of support for loops [7]. As a matter of fact, the verifier does not scale to programs with a large number of paths, its algorithm is not formally specified, and no formal argument about its correctness is given; as a matter of fact, multiple bugs have been already discovered.[25] Programmers have to "infer"

[25] https://www.openwall.com/lists/oss-security/2017/12/21/2.

the way that the verifier performs sanity checks, for their code to be accepted. This is a major problem, because developers often waste a lot of time for this process, e.g., by inserting redundant sanity checks on pointers. This definitely represents a major barrier for quickly extending the framework, especially when parsing headers with a number of variable options. A better verifier, grounded in state-of-the-art verification theory and practice, would enable a wider range of eBPF use cases and would dramatically simplify the development process [7].

6 Related Work

There are several examples in the literature of usage of eBPF technology for monitoring and inspection tasks. One possible application is to augment the context for flow monitoring, by providing additional data that cannot be retrieved in packet headers. Deri et al. [5] note that existing monitoring and tracing tools (Sysdig, Falco, Cilium) mostly lack the ability to match network with system events in order to provide complete visibility both at system and network level, while preventing unwanted network communications to take place similar to what IPSs currently do. Based on this consideration, they extend their existing tool *ntopng* with events generated by *libebpfflow*, a library that enriches network-layer data (e.g. source and destination IP addresses) with system metadata (e.g. source and destination processes and system users). Their objective is to support the definition of custom policies to drop unwanted connections, by affecting the return value of the kernel functions that are used to create network flows. However the Linux kernel supports override returns only on a small group of functions, which does not include the ones concerning network activities. For this reason, the kernel has been patched to enable this mechanism for network communications. A major limitation of this approach is that only local processes and users are visible through eBPF. As a future work, small ePBF probes will be created and disseminated in the network with the aim of circumventing this limitation and have visibility also on remote clients and servers.

A more common application is to improve the efficiency of dropping malicious DoS attacks. eBPF can be used to break up the conventional packet filtering model in Linux, moving the inspection process in the XDP, where ingress traffic can be processed before the allocation of kernel data structures which comes along with performance benefits [10]. This provides a first line of defense against traffic that is in general unwanted by the host, e.g., spoofed addresses or Denial-of-Service flooding attacks [1]. Independent studies have shown that XDP can yield four times the performance in comparison to performing a similar task in the kernel using common packet filtering tools [18], and can be even integrated with existing well-known management interfaces (i.e., iptables) [11]. This approach is made even more interesting given that dedicated platforms for packet filter offloading based on FPGAs like HyPaFilter [6] and hXDP [19] or SmartNICs supporting eBPF [12] have emerged.

In the context of network tracing, Suo et al. [20] propose a framework where a master node translates user inputs into configuration files, eBPF agents are

used to monitor network packets of specific connections at given tracepoints (e.g., virtual network interfaces), and measurements are centrally collected and analyzed. vNetTracer supports instrumenting kernel functions, return of kernel functions, kernel tracepoints and raw sockets through kprobe, kretprobe, tracepoints and network devices, hence providing a monitoring tool that works across the boundary of single domains.

Beside parsing and inspection, eBPF has also been used to support the redirection of the traffic to existing monitoring appliances. In [9] the authors propose an eBPF-based implementation for duplicating packets at the OpenvSwitch. They tackle the use case of monitoring the traffic between VMs without specific hardware appliances for duplicating packets. Their analysis shows that duplicating packets with an eBPF program in the TC hook achieves better throughput than native Port Mirroring of OpenvSwitch, especially in case of large packets.

Finally, there are also examples of management systems that are able to deploy eBPF programs and collect measurements, beyond the limited capability of the tools discussed in Sect. 2. Cassagnes et al. [2] designed a system for deploying eBPF programs and collect their measurements in containerized userspace applications. They used tools like Prometheus, Performance Co-Pilot, and Vector, and developed specific eBPF programs and their userland counterparts for monitoring the garbage collector, identifying HTTP traffic, and IP whitelisting. Nevertheless, they are only able to spot local information, and a global end-to-end view on distributed applications is still missing.

7 Conclusion

In this Chapter, we have proposed an alternative approach for network flow monitoring, based on eBPF technology. Our work was mainly motivated by the need for efficient yet effective solutions for virtualized environments, where hardware/software acceleration techniques are not available or largely ineffective, and resource usage is a performance and cost matter.

The extensive performance and functional validation demonstrated that our tool performs as well as other more traditional approaches, but with a far smaller resource utilization footprint. Indeed, almost the same set of information elements is available for L3/L4 headers as state-of-the-art solutions, but with great savings in terms of CPU usage and memory allocation. We do not claim that our bpfFlowMon outperforms nProbe and Zeek, especially in terms of functionality and absolute performance for physical systems. However, we have demonstrated that a more lightweight and portable approach is possible, especially for monitoring the traffic of virtualized applications in public cloud infrastructures.

There are a number of aspects that we are considering as part of our future research work. First of all, the aforementioned architectural change, from one single program to multiple programs linked by the tail call mechanism. This would also simplify the possibility to dynamically create eBPF code based on the required measures, since only the relevant protocols could be included. Second, we are interested in integrating our tool in Kubernetes CNI, likely Calico

or Cilium, to demonstrate that it could be used by different users without confidentiality and privacy concerns (i.e., each user should only be able to inspect its own traffic). Third, beyond specific aspects related to the eBPF technology, the current userland utility is rather limited in terms of export interfaces. It only dumps flows to file, both in plaintext and JSON; this can be easily composed with other common agents (FileBeat, Logstash) for sending data to the centralized platform, but it would also increase the overhead. Exporting data directly to Kafka is an additional feature that is worth integrating in the next release. All these extensions would allow to expose the our monitoring tool as an additional agent in the GUARD ecosystem, and to make it controllable by an external analytics platform.

Acknowledgment. This work was supported in part by the European Commission under Grant Agreement no. 786922 (ASTRID) and no. 833456 (GUARD).

References

1. Bertin, G.: XDP in practice: integrating XDP into our DDoS mitigation pipeline. In: Netdev 2.1, The Technical Conference on Linux Networking, Montreal, Canada, 6–8 April 2017 (2017). https://netdevconf.info/2.1/papers/Gilberto_Bertin_XDP_in_practice.pdf
2. Cassagnes, C., Trestioreanu, L., Joly, C., State, R.: The rise of eBPF for non-intrusive performance monitoring. In: 2020 IEEE/IFIP Network Operations and Management Symposium (NOMS 2020), Budapest, Hungary, 20–24 April 2020 (2020)
3. Claise, B.: Cisco systems netflow services export version 9. RFC 3954, October 2004. https://www.rfc-editor.org/rfc/rfc3954.txt
4. Claise, B., Trammell, B., Aitken, P.: Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information. RFC 7011, September 2013. https://www.rfc-editor.org/rfc/rfc7011.txt
5. Deri, L., Sabella, S., Mainardi, S.: Combining system visibility and security using eBPF. In: Proceedings of the Third Italian Conference on Cyber Security (ITASEC 2019). cEUR Workshop Proceedings, Pisa, Italy, 13–15 February 2019, vol. 2315, pp. 50–62 (2019)
6. Fiessler, A., Hager, S., Scheuermann, B., Moore, A.W.: HyPaFilter-a versatile hybrid FPGA packet filter. In: ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2016), Santa Clara, CA, USA, 17–18 March 2016 (2016)
7. Gershuni, E., et al.: Simple and precise static analysis of untrusted Linux kernel extensions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019), Phoenix, AZ, USA, 22–26 June 2019, pp. 1069–1084 (2019)
8. Hausenblas, M.: Container Networking - From Docker to Kubernetes, 1st edn. O'Reilly Media, Sebastopol (2018)
9. Hong, J., Jeong, S., Yoo, J.H., Hong, J.W.K.: Design and implementation of eBPF-based virtual TAP for inter-VM traffic monitoring. In: 2018 14th International Conference on Network and Service Management (CNSM), Rome, Italy, 5–9 November 2018 (2018)

10. Høiland-Jørgensen, T., et al.: 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 (CoNEXT 2018), Heraklion, Greece, 4–7 December 2018, pp. 54–66 (2018)
11. Miano, S., Bertrone, M., Risso, F., Bernal, M.V., Lu, Y., Pi, J.: Securing Linux with a faster and scalable iptables. ACM SIGCOMM Comput. Commun. Rev. **49**(3), 2–17 (2019)
12. Netronome: Avoid kernel-bypass in your network infrastructure. Blog post, January 2017. https://www.netronome.com/blog/avoid-kernel-bypass-in-your-network-infrastructure/
13. Phaal, P., McKee, N., Panchen, S.: InMon corporation's sFlow: a method for monitoring traffic in switched and routed networks. RFC 3176, September 2001. https://www.rfc-editor.org/rfc/rfc3176.txt
14. Qi, S., Kulkarni, S.G., Ramakrishnan, K.K.: Understanding container network interface plugins: design considerations and performance. In: 2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), Orlando, FL, USA, 13–15 July 2020 (2020)
15. Quittek, J., Zseby, T., Claise, B., Zander, S.: Requirements for IP flow information export (IPFIX). RFC 3917, October 2004. https://www.rfc-editor.org/rfc/rfc3917.txt
16. Sanchez, O.R., Repetto, M., Carrega, A., Bolla, R.: Evaluating ML-based DDoS detection with grid search hyperparameter optimization. In: IEEE International Conference on Network Softwarization, Tokyo, Japan (Virtual), 28 June–2 July 2021 (2021)
17. Sanchez, O.R., Repetto, M., Carrega, A., Bolla, R., Pajo, J.F.: Feature selection evaluation towards a lightweight deep learning DDoS detector. In: IEEE International Conference on Communications, Montreal, Canada (Virtual), 14–23 June 2021 (2021). https://zenodo.org/record/4967143#.YMtxw26xXUI
18. Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., Carle, G.: Performance implications of packet filtering with Linux eBPF. In: 30th International Teletraffic Congress (ITC 30), Vienna, Austria, pp. 209–217 (2018)
19. Spaziani Brunella, M., et al.: hXDP: efficient software packet processing on FPGA NICs. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020), 4–6 November 2020, pp. 973–990 (2020)
20. Suo, K., Zhao, Y., Chen, W., Rao, J.: vNetTracer: efficient and programmable packet tracing in virtualized networks. In: IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2–6 July 2018, pp. 165–175 (2018)
21. Zuppelli, M., Carrega, A., Repetto, M.: An effective and efficient approach to improve visibility over network communications. J. Wirel. Mob. Netw. Ubiquit. Comput. Dependable Appl. (JoWUA) **12**(4), 89–108 (2021). https://doi.org/10.22667/JOWUA.2021.12.31.089

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.