Network wide visibility with Linux networking and sFlow

Neil McKee, Peter Phaal, Roopa Prabhu, Andy Roulin, Ido Schimmel

InMon Corp, NVIDIA
San Francisco, Santa Clara, USA
neil.mckee@inmon.com, peter.phaal@inmon.com, roopa@nvidia.com, aroulin@nvidia.com, idosch@nvidia.com

Abstract

Support for network switch ASICs in the Linux kernel exposes hardware dataplane measurements through standard Linux APIs. The open source Host sFlow agent makes use of the Linux APIs to gather and export telemetry using the industry standard sFlow protocol for network-wide visibility into packet flows.

Keywords

analytics, DDoS, Linux, metrics, switchdev, sFlow, telemetry, ASIC

Introduction

Network traffic control requires real-time traffic monitoring, analysis, anomaly detection and response. There are many real time network analytics tools available on Linux and networking hardware today. In this paper we talk about sFlow, an industry standard for real-time network monitoring. We will look at how sFlow can be used to monitor a data center fabric consisting of networking hardware running Linux (switches and routers) and Linux virtual nodes. We will dive into the details of sflow integration into the Linux stack, Linux kernel, ecosystem and oss software.

sFlow is supported by most networking hardware vendors. Linux native support for packet sampling was introduced in the kernel followed by integrations into hardware support for packet sampling [1,2]. We will look at sFlow data formats, recent extensions to include drops, latency and queue depth and use these to detect and respond to events in the network fabric.

Real-time sFlow analytics can be used to rapidly detect DDoS attacks and filter them (e.g. with BGP FlowSpec or tc rules) before they even ramp up. Buffer-depth and transit-delay as measurements more commonly associated with in-band telemetry, are now also available out-of-band in standard sFlow.

Switch ASICs and Linux

The Linux kernel has a very rich dataplane that is capable of bridging, routing, tunneling and ACLs, among other things. In recent years, Linux gained support for hardware offload of these network functions by programming forwarding entries (e.g., FDBs, routes) to switch ASICs.

These switch ASICs are capable of forwarding billions of packets per-second and at switching capacities of several terabits per-second, representing a significant improvement over traditional general purpose CPUs.

While offloading of the dataplane from CPUs to switch ASICs results in substantial performance gains, it also results in significant degradation in visibility, as forwarded and dropped packets are invisible to the CPU. The next two sections will describe how this degradation in visibility can be mitigated using recent advances in switch ASIC observability under Linux.

Packet sampling

`tcpdump(8)` [20] is a common tool for inspecting network traffic flowing through an interface. However, when used on interfaces (i.e., Linux net devices) that correspond to the switch ASIC's front panel ports (e.g., `swp1`), only a very small subset of the traffic flowing through the interface is visible. This traffic usually consists of control packets such as ARPs and exception packets such as those that hit an unresolved neighbour in the switch ASIC during routing. The majority of the traffic that is forwarded correctly through the interface is invisible to the CPU.

Trapping all the forwarded traffic to the CPU for inspection is suboptimal for several reasons. First, it will result in a severe degradation of the switching capacity, as the host CPU is usually only able to forward a few millions of packets per-second compared to the few billions of packets per-second that can be forwarded by the switch ASIC.

Second, the bus connecting the switch ASIC to the host CPU (normally, PCIe) has a limited bandwidth which is several orders of magnitude lower than the switching capacity of the switch ASIC. This will result in a random and constantly changing sampling rate due to congestion on the ASIC end of the bus.

Third, injecting the trapped packets to the kernel receive path via `netif_receive_skb()` only so that they are visible to packet taps, will result in wasted CPU cycles. If traffic is mirrored to the host CPU instead of being trapped, duplicated packets will appear on the wire due to the packets being forwarded by both the software and hardware dataplanes.

Fourth, packets sampled from the hardware dataplane will usually have extra metadata [9][21] associated with them, such as the egress port, egress queue, egress queue depth and transit delay. Netlink [22] is an ideal TLV-based protocol that can be used to communicate sampled packets along with their associated metadata to user space.

For above mentioned reasons, in version 4.11 the Linux kernel was extended with the ability to sample packets from the dataplane (offloaded or not) to user space [23].

Packet sampling support in Linux consists of two main kernel modules. The first, `act_sample`, is a tc [24] action that configures the sampling and can be attached to various tc classifiers such as `cls_flower` and `cls_matchall`. The module facilitates the control plane of the sampling operation. For example, the following filter will configure sampling of a specific flow from the egress of swp1:

```
# tc qdisc add dev swp1 clsact
# tc filter add dev swp1 egress pref 10 proto ip flower skip_sw dst_ip 198.51.100.2 action sample group 3 rate 300
```

The `skip_sw` keyword instructs the kernel to only configure sampling in hardware. If omitted, sampling is configured in both the software and hardware dataplanes.

The second module, `psample`, registers a new generic netlink family called "psample" through which sampled packets from the dataplane are notified to user space along with associated metadata. These netlink packets contain various TLV attributes such as `PSAMPLE_ATTR_DATA` and `PSAMPLE_ATTR_LATENCY` that encode the payload of the packet (potentially truncated) and its transit latency, respectively.

Sampled packets are passed from the `act_sample` module to the `psample` module by invoking the `psample_sample_packet()` function. When sampling is performed in hardware, the switch ASIC device driver is expected to invoke the function directly.

In user space, the sampled packets can then be dissected using a Wireshark [25][26] dissector for the "psample" generic netlink family:

```
# psample -w - | tshark -r - -V
```

An alternative to packet sampling to the host CPU is to mirror packets directly from the hardware dataplane to a monitoring server that is capable of processing a much higher rate of traffic compared to the host CPU attached to the switch ASIC. However, this approach results in limited observability, as there is no industry standard dictating the format of the metadata in the mirrored packets. Therefore, operators are left with a choice: Mirror packets without metadata (limited observability) or encode metadata in a vendor-specific way (vendor lock-in).

Packet drops monitoring

Packets are dropped by the kernel by invoking the `kfree_skb()` function. This is in contrast to packets that are freed as part of normal operation by invoking the `consume_skb()` function. In order to allow users and developers to debug packet drops in the kernel, the `kfree_skb()` function includes a tracepoint called `skb:kfree_skb`. This tracepoint can be used to generate a stack trace whenever a packet is dropped, in order to see the path the packet took inside the kernel before being dropped. For example:

```
# perf record -a -g -e skb:kfree_skb -- sleep 5
# perf report --stdio
```

Another option to consume information about dropped packets is to use the popular `dropwatch` utility from the DropWatch package [27]. This utility opens a netlink socket and registers to multicast notifications about dropped packets from the "NET_DM" generic netlink family. These notifications are generated by a kernel module called `drop_monitor` that registers its own probe on the `skb:kfree_skb` tracepoint in order to be alerted whenever a packet is dropped.

Since kernel 5.4, in addition to the instruction pointer where packets were dropped, the `drop_monitor` module can also be instructed to generate notifications with the payload of the dropped packets (potentially truncated) and with various metadata [28], in a similar fashion to the previously described `psample` module. These notifications - netlink packets - can then be dissected in user space using a Wireshark [29] dissector for "NET_DM" generic netlink family:

```
# dwdump -w - | tshark -r - -V
```

When the dataplane is offloaded to hardware, all the instrumentation around the `skb:kfree_skb` tracepoint loses its value as packets are no longer dropped by the kernel, but silently by the hardware. In order to allow users and developers to have visibility into hardware originated drops, capable devices can be instructed to trap dropped packets to the host CPU [11][30]. For example, to trap packets that were dropped due to a blackhole route:

```
# devlink trap set pci/0000:01:00.0 trap blackhole_route action trap
```

To avoid wasting CPU cycles, these packet traps are disabled by default. To disable them, instruct the device to silent drop such packets:

```
# devlink trap set pci/0000:01:00.0 trap blackhole_route action drop
```

Packet traps can also be rate-limited to avoid overwhelming the host CPU:

```
# devlink trap policer set pci/0000:01:00.0 policer 1 rate 1024 burst 256
```

When reaching the host CPU from the device, these packets trigger the `devlink:devlink_trap_report` tracepoint, which serves as the hardware counterpart of the `skb:kfree_skb` tracepoint for hardware originated drops.

Since kernel 5.4, the `drop_monitor` module can be used to also trace hardware originated drops by registering its probe function on the `devlink:devlink_trap_report` tracepoint. In a similar fashion to software originated drops, the netlink notifications contain the packet payload and various metadata such as the drop reason (e.g., "blackhole_route").

The `devlink:devlink_trap_report` tracepoint can also be used as a hook point for a BPF program that maintains aggregated per-{trap, flow} statistics in a BPF map [31]. The statistics can then be exported to a time series database (TSDB) such as Prometheus [32] and visualized using Grafana [33].

Linux sFlow

Figure 1 illustrates the overall architecture of sFlow monitoring. Agents embedded within network, host, virtual network, container, and application instances stream standard measurements in real-time to an sFlow analyzer. The analyzer converts the raw measurements into useful metrics that can be used to drive orchestration, operations, and controller applications.

The open source Host sFlow agent makes use of Linux instrumentation to stream standard sFlow telemetry from Linux hosts and switches to the central sFlow analyzer [3].

Counters

The Host sFlow agent periodically queries `/proc/net/dev` to discover switch ports and periodically retrieve port counters. Additional hardware counters and pluggable optics metrics are obtained using `ethtool`.

Each set of counters is encoded as an External Data Representation (XDR) structure and immediately sent in a UDP datagram to the sFlow analyzer [4,5,6,7]. sFlow is designed for real-time monitoring. UDP transport is used for low latency, and the sFlow protocol is designed to be tolerant of packet loss.

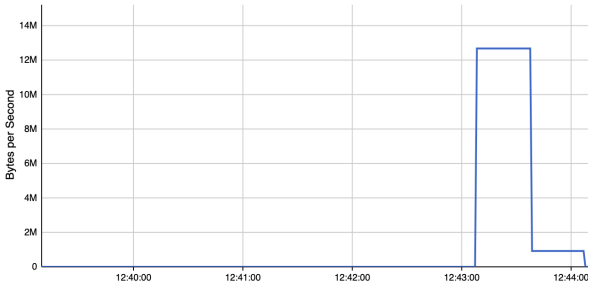

Figure 2: Trending value calculated from interface counters

Figure 2 shows a chart created by the browse-metrics application [8]. The chart trends the volume of traffic on a switch port computed from counters. The chart shows a spike in traffic, but also reveals a limitation of counters based metrics. Additional detail is needed, such as the source, destination, location, and type of traffic, in order to take action. Obtaining this detail from a network ASIC capable of forwarding billions of packets per second requires a different type of measurement.

Randomly sampled packets

To provide detail, sFlow also defines a mechanism for random sampling of network packet headers annotated with meta-data such as the ingress and egress ports, egress queue depth, transit delay and routing decision [5,9].

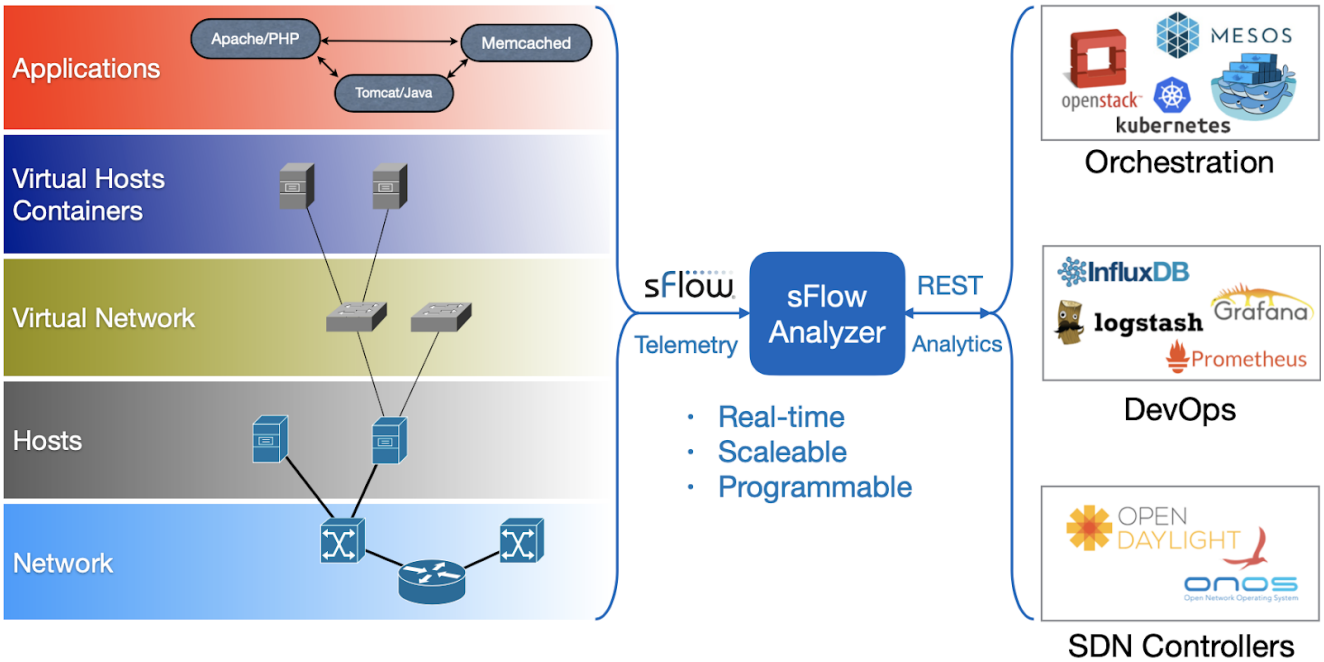

Figure 1: sFlow architecture

Support for sFlow's sampling mechanism is built into network switch ASICs. The switchdev driver exposes this hardware capability as part of the `tc` subsystem [2].

Packet sampling is configured for each switch port using a `tc matchall` filter to select ingress and/or egress packets. For example, the following filter enables ingress packet sampling on port `swp1`.

```
tc filter add dev swp1 ingress \
pref 1 matchall skip_sw \
action sample rate 10000 group 1 \
trunc 128 continue
```

The `skip_sw` flag pushed the configuration to the ASIC. Randomly sampled packet headers and associated metadata from the ASIC are directected to the `psample` netlink channel where they are received by the Host sFlow agent and immediately streamed along with the counters to the sFlow analyzer.

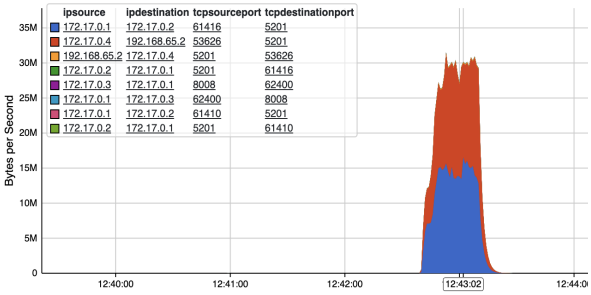

Figure 3: Trending flows calculated from packet samples

Figure 3 shows a chart created by the browse-flows application [10]. The chart provides an up to the second view of traffic flows, identifying the source, destination and protocol for each flow.

An interesting point to note when comparing Figures 1 and 2 is that packet sample metrics provide an immediate signal of traffic, while counters lag in proportion to the configured polling interval.

Packet-sampling is very effective for understanding network traffic flows. But what about network issues that prevent traffic from flowing?

Dropped packets

To troubleshoot packet-forwarding failures, sFlow also defines a mechanism to report dropped packets annotated with meta-data such as the drop-reason. Support for this mechanism is built into some current-generation network switch ASICs.

Packet drop monitoring is enabled by opening the netlink `drop_monitor` channel [11]. The Host sFlow agent listens for dropped packet notification and immediately forwards the packet header of the dropped packet along with associated metadata and drop reason.

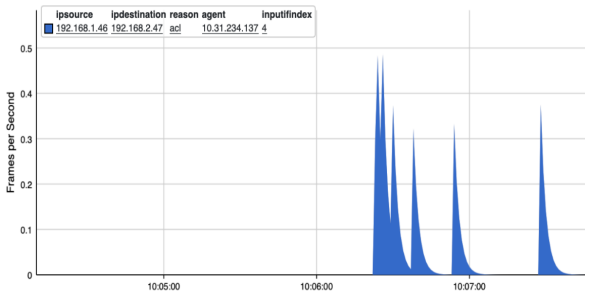

Figure 4. Trending discards calculated from drop events.

Figure 4 shows a chart created by the browse-drops application [13]. The application allows selected packet features and metadata to be trended an a chart that updates every second. In this case the source and destination addresses, drop reason, and drop location (ingress switch port) for a blocked TCP connection are shown. The exponential backoff pattern of TCP syn retries is clearly visible in the chart.

Packet drop notifications are useful for detecting and characterizing failure conditions such as black-hole routes, microburst buffer overruns, and MTU mismatches.

Configuration

Each Host sFlow agent requires minimal configuration. The following example shows typical settings from the configuration file, `/etc/hsflowd.conf`:

```
sflow {
 collector{ ip=172.20.20.1 }
 systemd { }
 psample { group=1 egress=on }
 dropmon { group=1 start=on sw=off hw=on }
 dent { sw=off switchport=swp.* }
}
```

The same configuration is used for every switch in the network, reducing the operational complexity of configuring sFlow monitoring.

Table 1: Default Host sFlow switch port settings

Link Speed	Sampling Rate	Polling Interval
1G	1-in-1,000	30 seconds
2.5G	1-in-2,500	30 seconds
5G	1-in-5,000	30 seconds
10G	1-in-10,000	30 seconds
25G	1-in-25,000	30 seconds
40G	1-in-40,000	30 seconds
50G	1-in-50,000	30 seconds
100G	1-in-100,000	30 seconds

400G	1-in-400,000	30 seconds

The default switch port settings shown in Table 1 ensure that large flows (defined as consuming 10% of port bandwidth) are detected within approximately 1 second. Counter polling and packet sampling are enabled on every port on every device for data center wide visibility.

DDoS mitigation

Distributed denial of service (DDoS) attack detection and mitigation is a common use case for sFlow telemetry. For example, the DDoS controller can be programmed to detect UDP amplification attacks based on sFlow packet samples.

Figure 5: DDoS mitigation using tc filter

When an attack is detected, specific attributes of the attack such as the target IP and UDP service are used to create a mitigation filter which is immediately installed on the switch. Figure 5 shows two attacks, The first is not mitigated. With the second attack, the instant that traffic matching the attack signature crosses the threshold, a control is triggered that filters the attack traffic. Real-time mitigation ensures that the full force of the attack never reaches the target.

tc filter

The article, DDoS mitigation using a Linux switch, describes how to use tc to filter denial of service attacks [14].

When an attack is detected, the controller makes use of a simple REST API to install a filter on the switch [15]. For example, the following filter is installed to block a DNS amplification attack against host `203.0.113.10`.

```
tc filter add dev swp1 ingress \
protocol ip pref 14 flower skip_sw \
ip_proto udp dst_ip 203.0.113.10
src_port 53 \
action drop
```

References

1. sFlow Industry Consortium, https://sflow.org
2. Ethernet switch device driver model (switchdev), https://www.kernel.org/doc/html/latest/networking/switchdev.html
3. Host sFlow, GitHub, https://github.com/sflow/host-sflow

In this case, the `skip_sw` flag in the filter instructs the switchev driver to offload the filter to hardware for line rate filtering of the attack traffic.

BGP RTBH/Flowspec

In production networks, BGP is the expected control-plane protocol. The controller peers with the router and can push remotely triggered blackhole (RTBH) routes or FlowSpec filters to mitigate attack traffic [17, 18]. The open source DDoS Protect controller is an example [16].

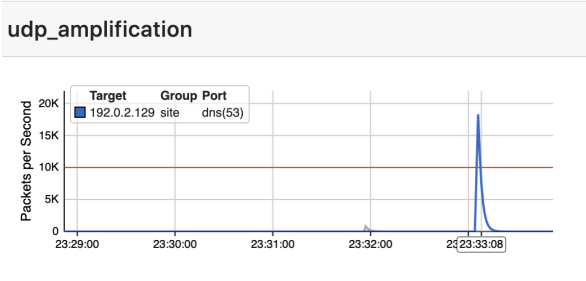

Figure 6: DDoS Protect

FRRouter is a widely-adopted open-source routing application [19]. Currently FRRouter propagates black hole routes as expected, but FlowSpec filters do not yet take advantage of the switchdev offload capability.

Conclusion

Native support for switch ASICs in the Linux kernel opens networking devices to Linux developers:

- Freedom to run any Linux distribution on switch
- Use standard Linux, tools, monitoring, and automation solutions
- Reduce complexity (and failures) by eliminating unnecessary services
- Opens up network hardware for developers to customize switches by adding measurement and control agents using standard Linux APIs
- Develop and test software on virtual machines / containers before pushing into production
- Large developer community (e.g. Stack Overflow, GitHub, etc.)

Support for ASIC counters, packet sampling, and dropped packet monitoring in the switchdev driver provides detailed visibility into hardware packet forwarding.

Streaming standard data plane measurements using sFlow integrates Linux switches into the existing sFlow analytics ecosystem where real-time network wide measurements support troubleshooting, reporting, and automation.

4. M. Eisler, Ed., XDR: External Data Representation Standard, IETF, May 2006, https://www.ietf.org/rfc/rfc4506.txt
5. Phaal, P. and Levine, M., sFlow Version 5, sFlow.org, July 2004, https://sflow.org/sflow_version_5.txt

6. Byford, J. Peterson, M., Joiner, S., and Phaal, P., sFlow Optical Interface Structures, sFlow.org, August 2016, https://sflow.org/sflow_optics.txt
7. Phaal, P. and Jordan, R., sFlow Host Structures, sFlow.org, July 2010, https://sflow.org/sflow_host.txt
8. Browse metrics, GitHub, https://github.com/sflow-rt/browse-metrics
9. Rozenbaum, C. and Phaal, P., sFlow Transit Delay Structures, sFlow.org, March 2021, https://sflow.org/sflow_transit.txt
10. Browse flows, GitHub, https://github.com/sflow-rt/browse-flows
11. Devlink Trap, https://www.kernel.org/doc/html/latest/networking/devlink/devlink-trap.html
12. Schimmel, I., Roulin, A., and Phaal, P., sFlow Dropped Packet Notification Structures, sFlow.org, October 2020, https://sflow.org/sflow_drops.txt
13. Browse drops, GitHub, https://github.com/sflow-rt/browse-drops
14. sFlow Blog, DDoS mitigation using a Linux switch, June 15, 2021, https://blog.sflow.com/2021/06/ddos-mitigation-using-linux-switch.html
15. tc_server, GitHub, https://github.com/sflow-rt/tc_server
16. DDoS Protect, GitHub, https://github.com/sflow-rt/ddos-protect
17. Turk, D., Configuring BGP to Block Denial-of-Service Attacks, IETF, September 2004, https://www.ietf.org/rfc/rfc3882.txt
18. Loibl, C., Hares, S., Raszuk, R., McPherson, D., and Bacher, M., Dissemination of Flow Specification Rules, IETF, December 2020, https://www.ietf.org/rfc/rfc8955.txt
19. FFRouting Project, https://frrouting.org/
20. man 8 tcpdump
21. sFlow Blog, Transit delay and queueing, March 17, 2021, https://blog.sflow.com/2021/03/transit-delay-and-queuing.html
22. J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov, Linux Netlink as an IP Services Protocol, July 2003, https://datatracker.ietf.org/doc/html/rfc3549
23. sFlow Blog, Linux 4.11 kernel extends packet sampling support, July 13, 2017, https://blog.sflow.com/2017/07/linux-411-kernel-extends-packet.html
24. man 8 tc
25. libpsample, GitHub, https://github.com/Mellanox/libpsample
26. Wireshark, GitLab, https://gitlab.com/wireshark/wireshark/-/commit/146578889e18da2f7b5d5dd680c5038bc87e2453
27. DropWatch, GitHub, https://github.com/nhorman/dropwatch
28. sFlow Blog, Visibility into dropped packets, July 16, 2020, https://blog.sflow.com/2020/07/visibility-into-dropped-packets.html
29. Wireshark, GitLab, https://gitlab.com/wireshark/wireshark/-/commit/a94a860c0644ec3b8a129fd243674a2e376ce1c8
30. man 8 devlink-trap
31. mlxsw, GitHub, https://github.com/Mellanox/mlxsw/blob/master/Debugging/libbpf-tools/src/trapagg_example.txt
32. Prometheus, https://prometheus.io/
33. Grafana, https://grafana.com/