MASTER THESIS

Václav Šraier

Performance of Open vSwitch-based Kubernetes Cluster in Pathological Cases

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Jiří Benc
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2023

Title: Performance of Open vSwitch-based Kubernetes Cluster in Pathological Cases

Author: Václav Šraier

Department: Department of Distributed and Dependable Systems

Supervisor: Jiří Benc, Red Hat Czech s.r.o.

Abstract: With the adoption of cloud computing, horizontally scalable infrastructure, and containerized deployments, Software Defined Networking (SDN) became an integral part of data centers, Kubernetes and Open vSwitch (OVS) being one of the commonly deployed solutions. Our work explores the possible performance limitations of OVS under Kubernetes, focusing on pathological traffic patterns. We discovered several types of packets causing excess system load on the cluster nodes. We identified the root cause as a series of drop rules in OpenFlow and a design flaw in OVS that prevents their efficient evaluation. We investigated the impact of this problem and our research revealed a specific system configurations under which an adversary can use the discovered inefficiencies for a practical denial of service attack on the local cluster node, bringing the whole networking stack down for all neighbouring containers.

Keywords: Kubernetes Open Virtual Network (OVN) Open vSwitch (OVS) OVN-Kubernetes Linux networking

Contents

Introduction

Modern containerized cloud computing systems have complex requirements for their networking backends. Demand for features like seamless cross-data-center networking, multi-tenancy and security policies necessitated the use of the Software Defined Networking (SDN) concept and, by the nature of containerized systems, extensive use of virtualized networks.

The current shift to microservices and the resulting increase of endpoints and the need for rapid reconfiguration emphasized the SDN control plane performance and scalability.

A commonly deployed solution is Kubernetes for container orchestration and Open vSwitch for the virtualized SDN, used either directly or indirectly. However, it remains a question of how well these solutions are adapted to the networking needs of microservices.

This work explores the performance and scalability characteristics of Open vSwitch (OVS) based Kubernetes clusters and is focused on investigating performance in pathological scenarios. While our experimental Kubernetes clusters were configured with the OVN-Kubernetes networking plugin, our findings should be transferrable to any other SDN installation using Open vSwitch.

We explored methods for stressing the OVS's control plane and discovered several problematic traffic patterns caused by a design flaw in OVS. We measured OVS's behavior under stress and learned that in certain configurations, an attacker can use the discovered inefficiencies for an effective denial of service attack on the local cluster node.

This thesis is divided into several chapters. In the first chapter, we provide descriptions of all relevant technologies, how they interact and how they work internally. The second chapter describes the configuration of our experimental clusters to allow anyone to replicate our findings. We describe our experiments and analyze their results in chapter three. The last, fourth, chapter discusses the impact of our findings and suggests possible improvements.

Chapter 1

Theoretical background

1.1 Glossary

SDN	Software-defined networking. Additional details can be found in section 1.2.
OVS	Open vSwitch (`https://www.openvswitch.org/`). Details in section 1.4.
OVN	Open Virtual Network (`https://docs.ovn.org/en/latest/`). Details in section 1.5.
CNI	Container network interface [1]. Specification for writing networking plugins for Kubernetes. The plugin handles assigning IP addresses and routing packets across the cluster.
OVN-Kubernetes	CNI plugin using OVN (section 1.6).
Forwarding tables	Network switches use forwarding tables to decide where to forward a received packet. In Ethernet switches, the forwarding tables consist of MAC address to network port mapping. SDNs generalized forwarding tables so that they can match packets in any way deemed useful, most commonly on any header values from the second, third, and fourth layers of the OSI model.
OpenFlow	Network configuration protocol used between an SDN controller and SDN switches. OpenFlow allows remote configuration of forwarding tables in network switches. Details in section 1.3
RTT	round-trip time

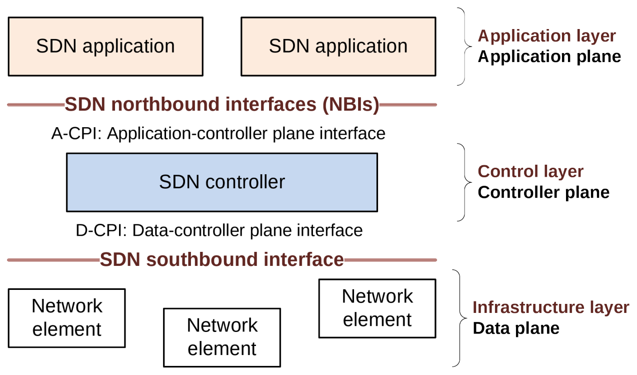

Figure 1.1 A schema of basic SDN components (originally from [2]).

1.2 Software-defined networking (SDN)

Software-defined networking is a loosely defined concept of separating the networking data plane (infrastructure layer) and the control plane (control layer) into separate components. In traditional networking architectures, each network device makes forwarding and/or routing decisions fully autonomously. SDNs separate the decision-making and data processing into distinct layers (see figure 1.1) and define two main interfaces. *SDN applications* provide the *SDN controller* with networking requirements using the *northbound interface*. The controller then configures the actual network equipment using the *southbound interface*.

1.3 OpenFlow

OpenFlow [3] is a protocol for configuring packet forwarders (generalized switches). OpenFlow is the de-facto standard protocol for the southbound interface in an SDN.

In OpenFlow, there are multiple forwarding tables for every network switch. Every table is identified with a number and contains a set of rules with priorities. Each rule consists of matching criteria, usage statistics and actions. The following is an example of an OpenFlow rule dumped from a running Open vSwitch using the `ovs-ofctl dump-flows $BRIDGE` command:

```
cookie=0x802cec73, duration=22932.301s, table=33, n_packets=0,
n_bytes=0, priority=75,
arp,metadata=0x7,dl_src=24:6e:96:3c:4c:5c,arp_op=1
actions=load:0x8004->NXM_NX_REG15[],resubmit(,37)
```

The third line in the example specifies the rule criteria. Generally, the matching criteria can include masks for header values from Ethernet, IP, transport layer protocols (TCP, UDP, ...), and more depending on the exact version of the OpenFlow protocol. The criteria can check for an exact value match or with a wildcard.

The action in a flow rule can forward the packet to a port, change a value of a header field, store an arbitrary value as metadata accompanying the packet in further processing, resubmit the packet to a different table and more. For the complete list of actions, see the OpenFlow specification [3] or the `ovs-actions` manpage [4].

In addition to the externally configured flow rules, the forwarding tables also contain statistics updated every time the flow rule is used. The OpenFlow controller can then query switches for these statistics.

1.4 Open vSwitch

Open vSwitch (OVS) [5] [6] [7] is a multilayer virtual switch. OVS runs as a software switch on all major platforms and supports hardware offload for its data processing layers. The supported OpenFlow protocol allows any SDNs to use OVS in its infrastructure layer.

The internal architecture of OVS mirrors the SDN architecture (see figure 1.2) with a control plane and data plane. `ovs-vswitchd` is the OVS's control process. `ovs-vswitchd` communicates via the OpenFlow protocol and stores the configured flow rules in a purpose-built Open vSwitch Database (OVSDB). Alternatively, the database can be accessed externally, and OVS can be configured without using the OpenFlow wire protocol.

A *datapath* is the primary part of OVS's data layer, the lowest component of OVS physically forwarding packets between configured ports. There are multiple datapath implementations, some of them implemented fully in the `ovs-vswitchd` process, some of them using extra kernel modules for improved performance. `ovs-vswitchd` translates the OpenFlow flow rules into a more efficient and simplified form. These simplified flow rules are then used by the datapaths to make forwarding decisions. We discuss datapath internals in more detail in section 1.7.

1.5 Open Virtual Network

Open Virtual Network (OVN) [8] [9] is an SDN combining Open vSwitch with network tunnels (by default GENEVE) for its infrastructure layer. Applications communicate with OVN through the `northdb`, an OVSDB instance storing high-level network configuration in terms of traditional networking concepts. The

Component	Description
`ovsdb`	OVSDB instance storing the OpenFlow flow rules
`ovs-vswitchd`	a process managing datapaths and providing OpenFlow configuration interface
datapath	a logical component in `ovs-vswitchd`, does the actual packet forwarding

Table 1.1 Overview of OVS components

`ovn-northd` process translates the configuration into logical datapath flows and stores the result in the `southdb` OVSDB instance. The `ovn-controller` then distributes the flows from the `southdb` into individual OVS databases.

Component	Description	How it runs
`northdb`	OVSDB instance storing network configuration in terms of traditional networking concepts	centralized or distributed
`ovn-northd`	process translating configuration into logical flows	centralized or distributed
`southdb`	OVSDB instance storing network configuration in terms of logical flows	centralized or distributed
`ovn-controller`	configures local OVS from the `southdb`	locally on every node

Table 1.2 Overview of OVN components

1.6 OVN-Kubernetes

OVN-Kubernetes [10] is a Kubernetes CNI plugin using OVN in the background. OVN-Kubernetes configures OVN and OVS on all cluster nodes and translates network reconfiguration requests from the CNI API into configuration changes for OVN.

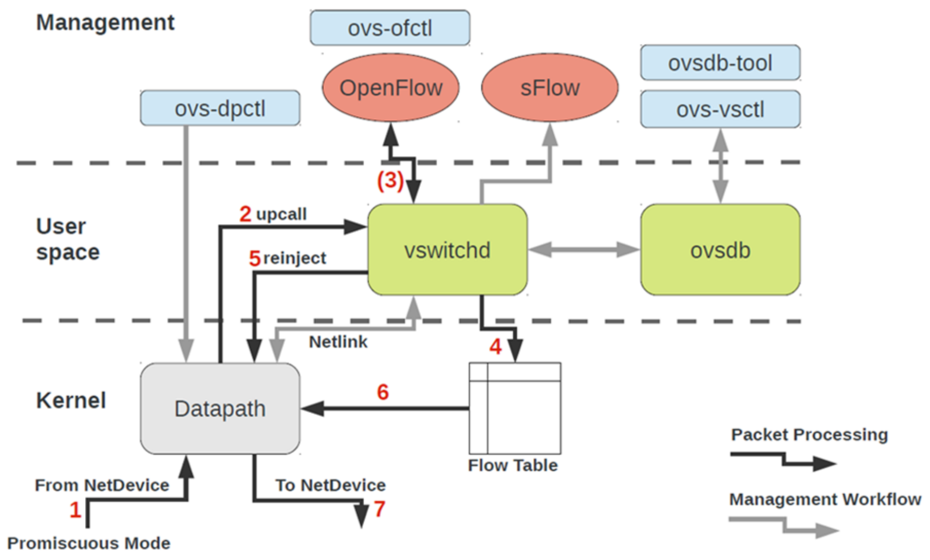

Figure 1.2 Schema of OVS internal architecture (originally from [11]).

1.7 Open vSwitch Datapaths

OVS processes packets in datapaths[1]. All datapaths implement a common interface hiding the implementation details of low-level packet processing from the layers above.

The two most commonly used datapath implementations are `netdev` and `netlink`. They are named after the mechanism used for integrating with the operating system. The `netlink` datapath is Linux-specific and uses a special kernel module. The `netdev` datapath is implemented in userspace.

Datapaths are optimized for high-performance packet processing and they completely handle most of the packets that come through them. Consequently, the forwarding tables in the datapaths are not equivalent to the OpenFlow forwarding tables and a translation mechanism between the two forwarding table formats is present. The translation is performed lazily. When a packet does not match any rule in the datapath forwarding table, an *upcall* is made and the packet is passed out of the datapath into the upper layers of `ovs-vswitchd`. `ovs-vswitchd` performs a full lookup through the OpenFlow forwarding tables and generates a new datapath flow rule which gets inserted into the datapath together with the packet itself.

We use the name *fast path* to refer to datapath-only packet processing. If the packet misses all datapath flow rules and generates an upcall, we say that it took the *slow path*.

[1]digital version of this text contains hyperlinks to the relevant source code

The `netdev` datapath The `netdev` datapath is a multiplatform datapath implementation entirely in user space. All major operating systems are currently supported.

The netdev datapath supports various configurations according to the system it is running on. The most basic deployment is rather slow as all packets have to cross the userspace boundary twice. However, the datapath can be configured with several types of accelerators significantly lowering the packet processing cost. The datapath can be accelerated using DPDK [12], AF_XDP sockets [13] or offloaded to hardware using TC [14].

Our research does not target the `netdev` datapath.

The `netlink` datapath The `netlink` datapath is Linux-specific. It resides partially in a kernel module and partially in user space. The kernel module does most of the packet processing. The userspace communicates with the kernel over a netlink socket, hence the name. Only the slow path passes data from the kernel space to the user space.

Our research is focused on the `netlink` datapath as it is the most commonly deployed datapath in Kubernetes clusters. Unless stated otherwise, the rest of the thesis always assumes the context of the `netlink` datapath.

1.7.1 Interactions between the fast path and the slow path

Packets always start their journey in the kernel. They enter the `openvswitch` module where they are checked against a set of flow rules stored in the flow table (OVS's forwarding table). If a flow rule matches the packet, its actions are executed. Otherwise, the packet is sent to the user space.

```
def process_packet_in_kernel(packet):
    for flow_rule in flow_table:
        if flow_rule criteria match the packet:
            execute the flow_rule actions
            return

    make an upcall to the user space
```

The slow path gets involved when the fast path fails (no flow rule match) or when the action in the flow rule explicitly asks for it. The packet is sent to the user space via a netlink socket in an upcall. `ovs-vswitchd` receives the packet, processes it according to the OpenFlow rules and reinjects it back into the kernel. Additionally, a new flow rule might be generated and installed into the kernel to speed up the future processing of similar packets.

The packets in the fast path are not buffered and they are processed immediately. The slow-path buffers packets when they are passed from kernel to user space, introducing additional latency.

1.7.2 The fast-path

The data structures

The flow table (`struct flow_table`) is an in-kernel data structure, which stores flows rules (`struct sw_flow`) and allows for fast matching with individual packets. Similar to OpenFlow flow rules, every datapath flow contains a list of actions (`struct sw_flow_actions`), statistics (`struct sw_flow_stats`), a flow key (`struct sw_flow_key`) and a flow mask (`struct sw_flow_mask`).

The flow key and the flow mask are the matching criteria. The combination of both of these enables fast matching with the possibility of wildcards. The key is a complex structure with parsed-out packet header values. It can be created from any packet with the function `key_extract()`. The mask specifies which bits are significant when comparing two keys.

```
def equals(key1, key2, mask) -> bool:
    return (key1 & mask) == (key2 & mask)
```

The matching algorithm

When the kernel processes a packet, it creates the packet's corresponding flow key and looks for matching flows in the flow table. The packet can match any number of flows, but only the first matching rule is always used. The userspace component is responsible for preventing overlapping conflicting flow rules.

The flow table stores the flows in a hash table. The lookup key is the masked `sw_flow_key`. Therefore, to find a flow for a packet, the kernel has to try several masks. The lookup procedure could look like this:

```
def lookup(flow_table, key):
    for mask in flow_table.mask_array:
        masked_key = apply mask to key
        if masked_key in flow_table.flows:
            return flow_table.flows[masked_key]
    return None
```

The real implementation (`ovs_flow_tbl_lookup_stats`) is in principle similar, but more optimized:

1. The kernel keeps mask usage statistics and the `mask_array` is kept sorted with the most used masks first (`ovs_flow_masks_rebalance()`). This happens periodically based on a time interval.

2. There is already a `sk_buff`[2] hash based on source/destination addresses and ports. The lookup procedure makes use of the hash by having a fixed size (256) hash table storing references to their masks. The cached masks are then tried first. If there is no match, a standard lookup over all masks follows and the cache entry is replaced with a new flow. This helps with burst performance.

1.7.3 The actions

The actions in a flow rule are similar to actions in OpenFlow. They are described in the manpages `ovs-actions(7)`.

Special attention should be given to the recirculation action, which corresponds to the `resubmit` OpenFlow action. This action resubmits the packet to the datapath again, updating its flow key's `recirc_id` to a new value. This effectively simulates having multiple flow tables in the datapath with only a single physical table.

1.7.4 The slow-path

The user space process (`ovs-vswitchd`) communicates with the kernel over a netlink socket. When a packet leaves the fast-path, it is temporarily buffered in a queue (`ovs_dp_upcall()`) when crossing the kernel boundary.

`ovs-vswitchd` reads packets from the kernel in several *handler threads*. The datapath interface defines a `recv()` function for receiving a single packet from the kernel. The netlink datapath implements it with the `dpif_netlink_recv()` function.

Higher up, the `recv()` datapath interface function is used in a generic `dpif_recv()` function which also provides an useful tracepoint for measurements. Even higher up the abstraction stack, the `recv_upcalls()` function in the file `ofproto-dpif-upcall.c` reads packets in batches, which are then processed by `handle_upcalls()`. The `handle_upcalls()` function essentially transforms the list of packets into a list of operations that should be executed on the datapath. This includes adding new flows to the datapath as well as simply sending packets where they belong.

[2]a kernel structure wrapping all packets

1.7.5 Additional `ovs-vswitchd` tasks

Parallel with upcall processing in the handler threads, OVS also runs several maintenance tasks:

- A balancing task makes sure that when the system is under stress, the most frequently used flow rules are in the kernel.
- The revalidator threads periodically dump statistics from the kernel and remove old unused flows. The number of revalidator threads scales with the number of available cores on the system.

Chapter 2

Experimental environment

This chapter describes the environment we used for running our experiments. The information provided here should allow anyone to replicate our measurements.

2.1 Hardware

For our research, we used two experimental clusters with different hardware configurations. One installation run virtualized on Proxmox VE with only a single physical host. The other installation used dedicated hardware. When we write about an experiment and do not specify where it runs, we are presenting results from the cluster on dedicated hardware. However, most of the measurements can be replicated in a virtualized environment without a significant impact on the outcome.

We always used a three-node cluster and we named the nodes `kb1`, `kb2`, and `kb3`.

Virtualized installation We used the virtualized environment for development and debugging. The physical host was running Proxmox VE 7.4-3, and it was configured with an Intel® Core™ i7-3770 CPU running at 3.40 GHz with 4 cores, 8 threads, and 31.23 GiB of RAM.

The virtual machines used for the cluster nodes were each configured with 4 virtual cores and 4 GiB of RAM. We had initially started experimenting with 2 CPU cores per node to prevent overprovisioning, but our workloads always fully stressed only one node (always `kb2`), and we were mostly interested in system behavior with multiple threads. The overprovisioning did not seem to cause any problems.

The cluster nodes were equipped with 2 virtual network interfaces connected to two virtual Linux bridges on the host. We used one interface for system management and WAN access. The other bridge was used for cluster interconnect. We did not artificially limit the throughput or latency of the link between nodes. When measured between `kb2` and `kb3` using `iperf3` with the default configuration and `ping` with 100 samples, the throughput was 14.5 Gbps and the average round trip time 0.383 ms.

Dedicated hardware We used the cluster with dedicated hardware to validate the results observed in the virtual environment. The nodes were Dell PowerEdge R730 servers, configured with Intel® Xeon® CPU E5-2690 v4 running at 2.60GHz with 14 cores, 28 threads, and 131 GB of RAM.

Similar to the virtualized nodes, the dedicated nodes had 2 network interfaces. One management 1 GbE network interface was connected to the Internet, and a different 10 GbE interface was connected to a switch. We configured the switch to use VLANs to isolate the cluster traffic from all other ports. While we did not have the opportunity to use a fully dedicated switch, the switch should have had enough capacity so that it would not be a bottleneck. The average round trip time between `kb2` and `kb3` (`ping -c 100`) was 0.104 ms.

2.2 Software environment

We run our experiments and measurements on a Kubernetes cluster with OVN-Kubernetes as the CNI plugin and Docker as the container runtime. OVN-Kubernetes was installed using containerized setup following the official installation guide [15]. We used Fedora 38 as the base Linux distribution. To allow reproducibility, we fully automated the installation procedure. Installation scripts with usage instructions can be found in appendix B.

In our experiments, the three-node cluster always had one node dedicated as a control node (`kb1`). We didn't test cluster installations configured for high availability (HA). We focused on the internals of OVS, a part of low-level networking infrastructure. We do not expect any significant difference between HA and non-HA clusters.

Our experiments always run on`kb2`.

2.2.1 OVN-Kubernetes configuration

We deployed a fully containerized OVN-Kubernetes, meaning that even OVS was deployed in a privileged container. The default OVS container spec file contains this resource limit:

```
resources:
    requests:
        cpu: 100m
        memory: 300Mi
    limits:
        cpu: 200m
        memory: 400Mi
```

We have manually removed this limit for most of our experiments. We have also tested with the limit active and we always explicitly mention it when this is the case.

2.2.2 OVS modifications

Instead of the default OVS container, we configured our systems with a modified version to improve observability. Our changes included:

- We added development tools (e.g. `gdb`, `perf`, ...) to the container image.
- We recompiled `ovs-vswitchd` to include additional user statically-defined tracing (USDT) [16] probes
- We included debug symbols in the `ovs-vswitchd` executable

Implementation details and instructions on how to replicate our build are in appendix A.

2.2.3 Kubernetes configuration

We always deployed three different pods to the Kubernetes cluster. The spec files are attached to this work. The pods were:

- `arch` on `kb2` – a pod running the latest Arch Linux Docker image. We used this pod for the main part of our experiments.
- `victim` on `kb2` – again, a pod with the latest Arch Linux image. We used it to measure the impact of our experiments on pods sharing the same node.
- `reflector` on `kb3` – a privileged Arch Linux container with a custom raw-socket-based packet reflector and `iptables` rule preventing the kernel from sending ICMP connection refused packets. The reflector swapped the Ethernet and IP addresses in the received packets and sent them out again.

2.2.4 Experiment implementation

We developed our experiments mainly in Rust. Everything is contained within one project in a tool we call the `analyzer`. In the few cases when Rust was not the best language for the task, we embedded scripts in other languages (mainly Bash and Python) in the Rust executable itself. Our Rust code always provides an entry point.

The main reasoning behind the language and architecture choice was personal preferences and ease of distribution – the ability to create a single statically-linked executable that would work almost anywhere.

Implementation details and usage instructions are in appendix C.

2.2.5 Clock synchronization

All our experiments use Linux's `CLOCK_MONOTONIC` clock for timekeeping. Because we measure everything on a single node, `kb2`, the clock is perfectly consistent across different containers. There could be small discrepancies due to synchronization between CPU cores and scheduling, but we are mostly concerned about larger time scales so we assume these discrepancies will not affect our results.

Chapter 3

Experiments

Origin of the experiments At the beginning of our research into the performance and scalability of common Kubernetes and Open vSwitch configurations, we did not know which part of the networking stack will lead to interesting results and where to focus on. So we started practically experimenting with packet handling and always continued based on our previous results. In the end, our exploration led us to focus on upcall handling and flow rule management.

As we showed in chapter 1, OVS inserts flow rules into datapaths on demand after upcalls. Therefore packets generating upcalls are more expensive to process than the other packets. We tried to create a network traffic, which would reliably generate upcalls to study the behavior of OVS under stress. This chapter describes the details of our research.

The experiments We split our research into the following parts:

1. We investigated flow rule handling in OVS. If we sent the same packet twice, the second will always hit a rule in the datapath and it will not generate an upcall. Except for cases, when the flow rule is already removed. When does that happen? When are flow rules removed from the datapaths? We investigated this in section 3.1.

2. What is the cost of an upcall? How much slower is the slow path compared to the fast path? Can we infer whether upcall-only traffic will be a significant performance problem? (section 3.2)

3. Once we knew the flow rule timeout, we could investigate packet types generating upcalls. Which packet header values lead to a generation of new flow rules? (section 3.3)

4. The last and most significant part - what is the impact of artificial upcall-only traffic on the performance of the whole system? We investigated this in section 3.4.

Most of the questions above can be approached using both static and dynamic analysis (analyzing the source code or analyzing the behavior of a running system). At the beginning of our investigation, we were not familiar with the relevant code and we also did not know what exactly to focus on, so we always started with a dynamic analysis, because it allowed us to observe the behavior of the whole networking stack. After the initial practical experiment, we searched for the code causing the observed behavior.

In the following sections, we discuss the design of our experiments and their results. Every following section corresponds to one of the investigation parts mentioned above.

3.1 Flow eviction timeout

We can observe an effect of an upcall using the `ping` tool. The first packet has higher latency than the rest of the ICMP packets because it generates an upcall and a new datapath flow rule (can be verified by `ovs-dpctl dump-flows`).

```
[root@kb2 ~]# ping -c 5 kb3
PING kb3 (192.168.1.223) 56(84) bytes of data.
64 bytes from kb3 (192.168.1.223): icmp_seq=1 ttl=64 time=1.19 ms
64 bytes from kb3 (192.168.1.223): icmp_seq=2 ttl=64 time=0.404 ms
64 bytes from kb3 (192.168.1.223): icmp_seq=3 ttl=64 time=0.365 ms
64 bytes from kb3 (192.168.1.223): icmp_seq=4 ttl=64 time=0.424 ms
64 bytes from kb3 (192.168.1.223): icmp_seq=5 ttl=64 time=0.304 ms

--- kb3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4062ms
rtt min/avg/max/mdev = 0.304/0.536/1.185/0.326 ms
```

Listing 3.1 Output of the `ping` command in the virtualized environment

To repeat the observation, we must not send similar ICMP packets for several seconds. The higher latency is measurable only when we wait. This behavior can be explained by a flow rule eviction timeout that removes the rule from the datapath's forwarding table.

Assuming the timeout stays constant, we can measure it by varying the interval between ICMP packets. The dependency between the measured latency and the time delay should be constant except for one sharp increase in latency when the delays get longer than the timeout.

Our experiment We experimented in the following way:

1. Generate a random number D in the interval $[8; 12]$
2. sleep for D seconds
3. run `ping -c 3 -i 0.01 192.168.1.221`
4. log D and both round trip times
5. go to step 1 until we have enough samples

We chose the interval based on non-rigorous preliminary experiments. After plotting RTT's dependence on the delay D, we expect to see:

- The first RTT data points will form a line with a sharp increase at a value of D corresponding to the eviction timeout
- The second RTT data points will form only a single horizontal line.
- The third ping will have the same results as the second.

The practical nature of this experiment allows us to check OVS's configuration even on systems where we do not have privileges to access OVS directly.

3.1.1 Results

The eviction timeout The measured latencies of the experiment (figure 3.1) show the flow rule eviction taking effect about 10 seconds after the last datapath rule installation. The observed value is not exact, but it is probably reasonable to assume the developers chose a nice-looking number. We hypothesize that the noise in the location of the increase is introduced mainly by infrequent eviction checks.

Following these experimental findings, we searched the source code and it specifies exactly 10 seconds as the default eviction timeout. The experimental findings are consistent with the source code. The default rule timeout is defined in the file `ofproto/ofproto.h` as:

```
#define OFPROTO_MAX_IDLE_DEFAULT 10000 /* ms */
```

The file `ofproto/ofproto-dpif-upcall.c` contains the code enforcing the timeout in the function `revalidate()`:

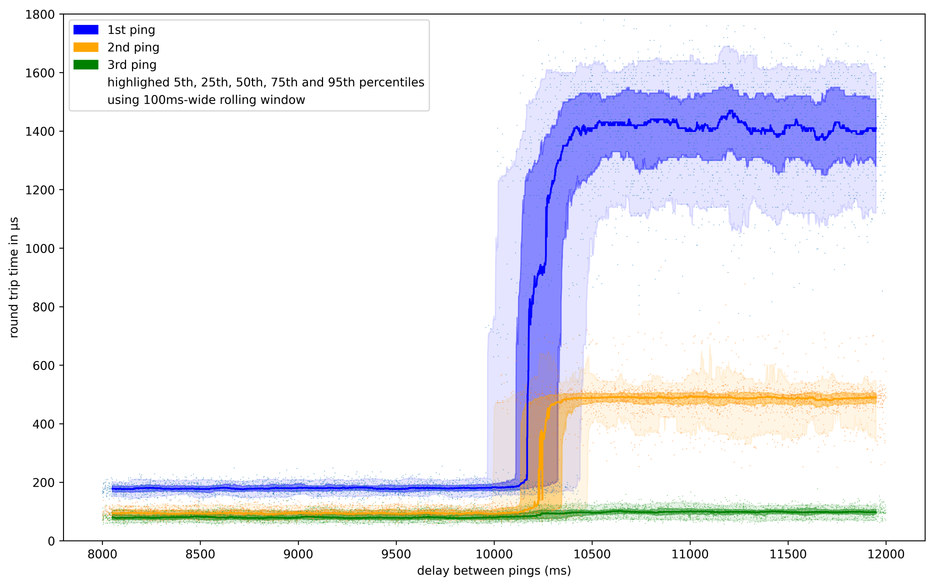

Figure 3.1 Results of the eviction timeout experiment

```
if (kill_them_all || (used && used < now - max_idle)) {
    result = UKEY_DELETE;
}
```

The eviction is handled by the revalidator threads, which run periodically roughly every 500ms. This confirms our hypothesis about the 500ms spread of the observable timeout effect.

```
// in file ofproto/ofproto.h:321
#define OFPROTO_MAX_REVALIDATOR_DEFAULT 500 /* ms */

// in the ofproto/ofproto_dpif_upcall.c:1052
// in function udpif_revalidator()
poll_timer_wait_until(start_time + MIN(ofproto_max_idle,
                                        ofproto_max_revalidator));
```

Difference in latency below timeout Figure 3.1 shows a difference between round trip times for the first and second packet even for delays shorter than the eviction timeout. The difference is small, so we assume that this can be caused by CPU caches. We did not investigate this further.

Increase in round trip time of the second ping Figure 3.1 also shows an increased latency for the second ping after the eviction timeout. We hypothesize that it could be caused by the fixed size flow lookup cache described in section 1.7.2. The upcall installs the new flow, but the lookup cache is filled only after the first use (i.e. the second packet of the flow, the second ICMP ping). In other words, if we would send multiple packets, their journey through the `openvswitch` module would be as follows:

1. A cache miss, followed by a flow table miss. Leads to an upcall and a new flow rule.
2. A cache miss, which requires a full flow table scan. The cache is filled.
3. Cache hit.
4. All consecutive packets hit the cache unless the cache entry is somehow overwritten.

The third RTT measurement behaves as expected by our hypothesis. We shortly experimented with even more consecutive measurements and they were all the same.

3.2 Cost of an upcall

Eelco Chaudron investigated the cost of an upcall in OVS [17] using eBPF probes in the Linux kernel. His experiments show that processing a packet through the slow path can take anywhere between 150 µs and 10 ms extra compared to the fast path. However, as we saw in the previous section, upcalls have visible effects outside of the kernel and we can measure them directly.

The unchanged experiment devised in the previous section provides us with a direct measurement of the observable upcall effect. The difference between the first packet RTT and the second packet RTT should correspond to how much time an upcall costs.

This experiment assumes that ICMP packet processing in OVS is the same as for all other types of packets. We believe this is most likely correct because OVS generates datapath flow rules for handling ICMP packets the same as it does for all other packets. Also, we did not find any evidence of ICMP packets being handled specially.

3.2.1 Results

Upcall processing cost As discussed in chapter 3, the round trip time increase can be attributed to the cost of upcall processing. In the previous section, we noticed an increased latency with the second RTT measurement, possibly a cache miss in the kernel datapath. We can estimate the cost of an upcall using this data. We calculated the difference between the first and the second ping measurements and assume the result is the lower bound for the upcall cost. Either our hypothesis about the cache miss is correct. Then the difference would be exactly the time spent processing the upcall. Or, we are wrong and the higher second ping latencies are caused by some additional factor that is not normally present. In that case, the upcall cost would be certainly equal to or higher than our result. For our calculation, we used only latency measurements with the delay since the previous measurement larger than 10.5 seconds.

	First ping RTT	**Second ping RTT**	**Difference**
#samples	1596	1596	1596
Mean	1399.94	483.66	916.28
SD	162.74	59.73	154.52
Min	769	250	149
25th percentile	1300	467	814
Median	1410	488.0	935
75th percentile	1520	504	1031
Max	2030	878	1460

Table 3.1 Statistics of measured round trip times when the interval > 10.5 seconds

The mean of the latency difference is within 916.28 ± 7.59 µs with 95% confidence.

Comparison to previous results We can not directly compare our results with Eelco Chaudron's because we do not have the same test environment. Under simulated normal conditions, he concluded that the time from the kernel's upcall invocation til the kernel's packet execution is on average 149.83 µs ($n = 56446$). Our results are 6x higher. In the same article, he notes:

> My sample runs have shown what I want to emphasize in this article: The way the upcalls are generated highly influences the outcome of the upcall costs.

And he shows that under stress, the average upcall cost can be 29367.91 µs ($n = 13807$). Therefore, all we can say is that our results are in the range of possible upcalls costs according to his article.

Due to the variance, these results also do not allow us to estimate the impact of an upcall heavy traffic on the overall system performance. According to Chauldron's article, the upcall processing is batched and therefore individual upcalls have a different associated latency cost than an upcall flood.

3.3 Packets generating upcalls

To stress-test the slow path, we have to be able to generate upcalls consistently. We have to find types of packets that will repeatedly miss all installed flow rules in the OVS datapath.

Our experiment sends varying packets in batches based on their type and monitors the upcalls and flow table changes using kprobes in the kernel and user statically-defined tracing (USDT) probes in `ovs-vswitchd`.

The flow rules in OVS datapaths use the `struct sw_flow_key` to represent the flow key. For every field of that structure, our measurement tool sends 1000 packets with the corresponding packet header field randomized and everything else fixed at arbitrary values.

For example, let us consider the IPv4 TTL field. Given any valid IPv4 packet, we want to send the same packet (bytes) 1000 times, each time with random bits in the location of the TTL header field. The values of all other header fields except for TTL stay fixed, so that their changes do not generate any new flow rules.

We used the Scapy project to generate the packets. The following code snippet is an example from our tool:

```
tag("IP(ttl)")
sendp(
    Ether(dst="aa:bb:cc:dd:ee:ff") / IP(
        dst="10.244.1.1",
        ttl=RandByte()
    ),
    count=1000
)
sleep(11)  # more than the eviction timeout
```

Because we probed a running system and we did not analyze all code paths, we cannot be certain that we covered all possible cases for generating new flow rules. We can draw some general conclusions from the measurement results and look into the source code for additional insights. There is always the possibility that we missed something. However, a complete static analysis would be extremely time-consuming. The results depend on OVS, the whole SDN, and its configuration. The search space is just too large. For that reason, we did not attempt to use static analysis to verify or expand upon our results.

3.3.1 Results

Figure 3.2 shows the number of upcalls generated after sending 1000 crafted packets differing only in a value of a specified header field. We are interested

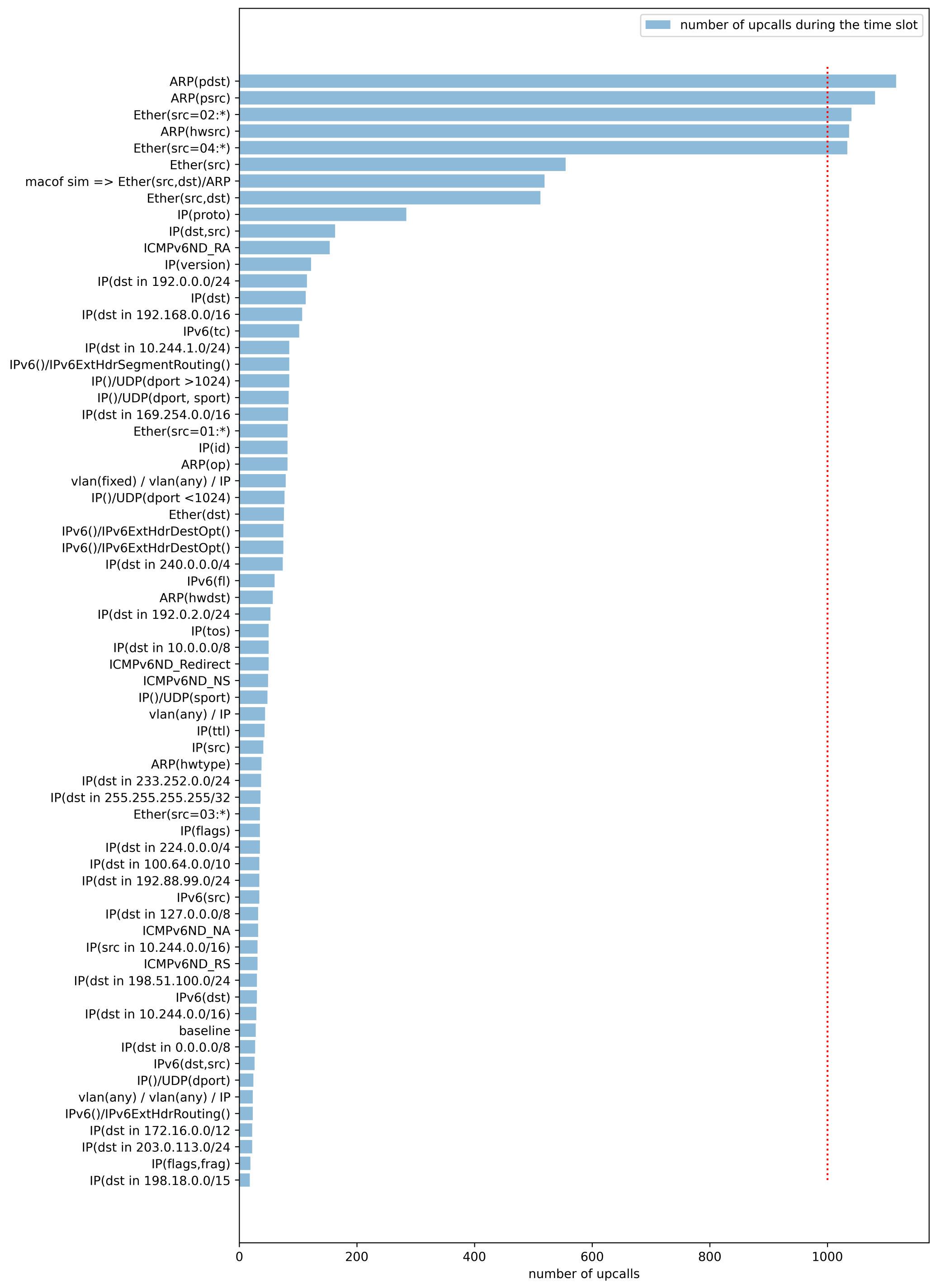

Figure 3.2 Number of upcalls generated by varying certain header fields in packets

in the peaks of the plot, as they signify packet types that consistently generate upcalls. The most significant being varying Ethernet source addresses and address fields in ARP packets.

3.3.2 Analysis

Ethernet source addresses

Varying the Ethernet source address in the unicast range (the least significant bit of the first byte has to be 1) generates new upcalls. The flow rules being inserted into the flow table are similar to the following example[1]:

```
recirc_id(0),in_port(9),
    eth(src=04:6a:68:88:2b:eb),eth_type(0x0800),ipv4(frag=no),
    packets:0, bytes:0, used:never, actions:drop
```

OVN has a feature called port security which can be enabled for logical switch ports. By enabling port security on a port, MAC spoofing is prevented and all packets with the wrong MAC address are dropped. OVN-Kubernetes enables this feature. The command `ovn-nbctl list Logical_Switch_Port` prints configuration for all logical switch ports, including information about port security. The following snippet is part of the command's output, a description of the logical switch port used for our test pod called `arch`.

```
_uuid               : cc7a2d01-52b4-4529-a026-55bf9d46dc56
addresses           : ["0a:58:0a:f4:01:05 10.244.1.5"]
dhcpv4_options      : []
dhcpv6_options      : []
dynamic_addresses   : []
enabled             : []
external_ids        : {namespace=default, pod="true"}
ha_chassis_group    : []
mirror_rules        : []
name                : default_arch
options             : {
    iface-id-ver="40c48294-7f78-4cc0-8a74-cffd9ecec647",
    requested-chassis=wsfd-netdev65.ntdv.lab.eng.bos.redhat.com
}
parent_name         : []
port_security       : ["0a:58:0a:f4:01:05 10.244.1.5"]
tag                 : []
tag_request         : []
type                : ""
up                  : true
```

[1]dumped with `ovs-dpctl dump-flows`

Quoting the OVN documentation [18], a section about the port security option[2]:

> This column controls the addresses from which the host attached to the logical port (”the host”) is allowed to send packets and to which it is allowed to receive packets. If this column is empty, all addresses are permitted.
>
> Each element in the set must begin with one Ethernet address. This would restrict the host to sending packets from and receiving packets to the ethernet addresses defined in the logical port’s port_security column. It also restricts the inner source MAC addresses that the host may send in ARP and IPv6 Neighbor Discovery packets. The host is always allowed to receive packets to multicast and broadcast Ethernet addresses.

The port security flow rules are added in OVN, in `controller/lflow.c` in function `consider_port_sec_flows()`. OVN adds multiple OpenFlow rules into multiple flow tables to implement port security.

Because OVS’s datapath flow rules are much simpler than OpenFlow flow rules, there is no 1-1 mapping between them. Moreover, the datapath flow rules allow only positive matches (see section 1.7.2). They cannot express negative matches. Therefore, when we send packets with varying MAC addresses, `ovs-vswitchd` evaluates the packets against the OpenFlow rules and finds out that the packet should be dropped. The newly generated datapath flow rule checks for an exact match on the random MAC address and is not generic to match any other packets.

Looking at the problem from the other side, OVS’s datapath is designed to assign packets to logical flows and execute the flow actions in as few instructions as possible. There are no precedence rules in the datapath flow table, the first matching rule is used and the order of insertion is not maintained. There are also only positive matches, no negative matches. Therefore, a single rule to drop all packets except for those with a given MAC address is impossible to construct.

ARP packet fields

ARP packets with varying hardware addresses behave the same as ordinary Ethernet packets with varying source hardware addresses because port security in OVN applies to them as well.

[2]The documentation calls it a column because the configuration is stored in a database column

Other types of packets

OVN's documentation states that IPv6 Neighbour Discovery (ND) packets are also covered by the port security setting and we should therefore be able to observe the same behavior as with ARP packets. However, we did not generate any upcalls with the ND packets. There are two likely options why this is the case:

- We did not configure the cluster properly for dual-stack networking with both IPv4 and IPv6.
- We might have crafted invalid IPv6 ND packets.

Either way, we expect that IPv6 ND packets are also a potential problem and they should be checked as well when developing a fix.

3.4 Impact of upcall-heavy traffic

Stress testing tool From the previous experiment, we learned that randomized unicast Ethernet source addresses generate upcalls. We used this knowledge to write a custom tool for sending minimalistic Ethernet frames. The Ethernet header needs only 14 bytes, but we used 16-byte packets due to slightly better performance. Instead of randomization, we filled the source Ethernet addresses with an increasing integer sequence.

Our tool is optimized for controlled and regular packet generation. Configured with a time interval between packets or a desired packet frequency, it tries to send packets as regularly as possible:

```
start_time = clock()
sent = 0
while True:
    now = clock()
    while we should have sent more packets than we sent:
        sent a new packet
        sent += 1

    sleep( until next packet is scheduled )
```

Only instead of sending the packets one by one every time, we send them in batches using `io_uring` if it is possible. When we set the interval to 1 ns (zero is not possible due to division by zero), we can reach roughly 210k packets per second in a single thread on the dedicated test server.

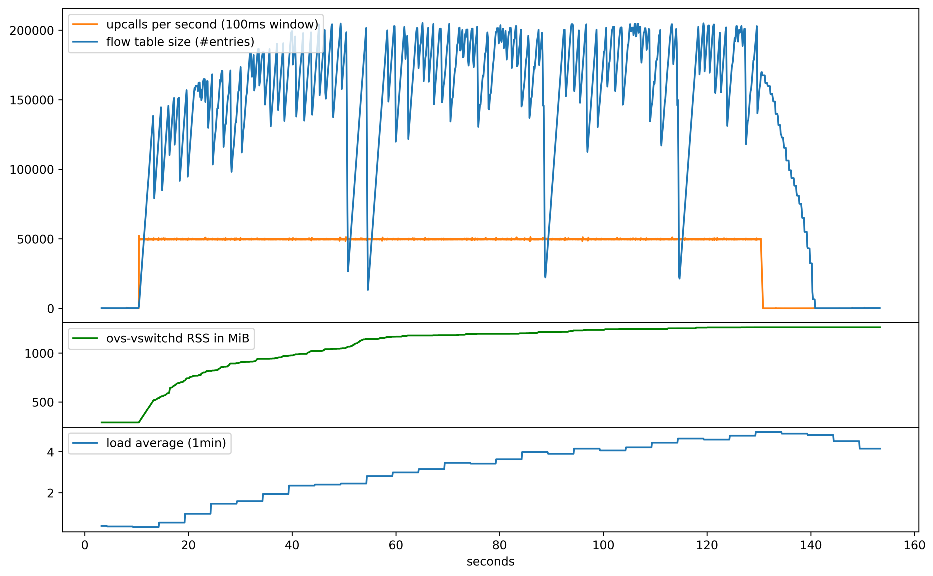

Figure 3.3 `ovs-vswitchd` stressed with 50k upcalls per second

The experiment For our experiment, we stressed OVS using our tool and monitored the system. We mainly measured OVS's memory consumption, the load average of the whole system, latencies from the `victim` pod to the node `kb1` (ICMP) and to the `reflector` pod on `kb3` (UDP). We mainly expected an increased processing time and therefore increased latencies compared to latencies in an idle system. We also focused on memory consumption and CPU load because we noticed resource limits placed upon OVS containers in the default OVN-Kubernetes deployment configuration. The experiments always started with a freshly restarted `ovs-vswitchd`.

3.4.1 Performance with unlimited resources

Flow table size Figure 3.3 and figure 3.4 show the recorded behavior of `ovs-vswitchd` stressed with a significant number of upcalls. Assuming the 10-second flow rule timeout would be the only criteria for evicting flow rules, the flow tables should have reached 500k and 150k flow rules respectively. However, `ovs-vswitchd` has a default limit of 200k flow rules in the flow table which explains the lower-than-expected table size in figure 3.3. Moreso, the flow table size limit is dynamic based on measured system performance (more detail in section 3.4.2) and 200k is only a hard limit beyond which the table will never grow. The dynamic increase of the flow limit until the hard limit is reached is visible at the beginning of figure 3.3.

In both experiments, the flow table size fluctuates. With fewer upcalls, when

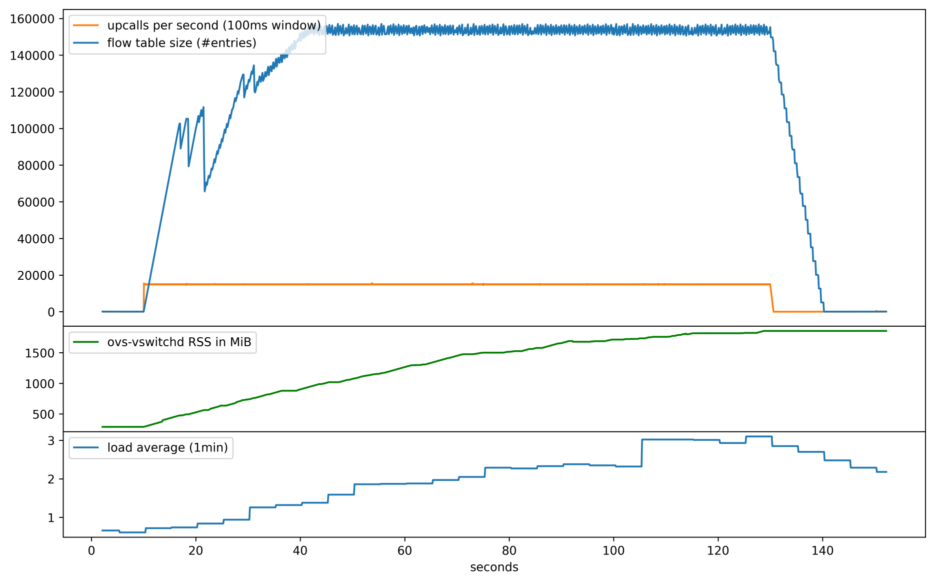

Figure 3.4 `ovs-vswitchd` stressed with 15k upcalls per second

only the timeout is at play, the fluctuations are small and can be explained by periodic timeout enforcement in the revalidator threads in `ovs-vswitchd`. When the flow table size hits the 200k limit, an additional regulatory mechanism in the revalidator threads activates. Quoting a comment[3] from OVS's source code:

> In normal operation we want to keep flows around until they have been idle for 'ofproto_max_idle' milliseconds. However:
>
> - If the number of datapath flows climbs above 'flow_limit', drop that down to 100 ms to try to bring the flows down to the limit.
> - If the number of datapath flows climbs above twice 'flow_limit', delete all the datapath flows as an emergency measure. (We reassess this condition for the next batch of datapath flows, so we will recover before all the flows are gone.)

In other words, the revalidator thread checks flow rules in batches and whenever the flow table grows above the limit, a shorter eviction timeout is applied to the current batch. We hypothesize that the high amplitude of the fluctuations is caused by synchronization between multiple threads. There are multiple revalidator threads, each has its flow rule dump thread and the actual rule deletion is

[3]`https://github.com/openvswitch/ovs/blob/474a179aff6c/ofproto/ofproto-dpif-upcall.c#L2771-L2782`

handled in the kernel after receiving a deletion command over a netlink socket which has an internal queue. We did not confirm nor disprove the hypothesis.

System load Because our packet flooding tool is a single-threaded application, it directly contributes to the load averages only by a value of 1 or less. However, we observe a significantly higher load average in both figure 3.3 and figure 3.4. Except for our running experiment, the system is idle. Hence, we attribute the additional system load to OVS. As we discussed before in section 1.7.5, OVS uses multiple threads during normal operation. By default, the revalidator threads iterate over the whole flow table every 500 ms and command the kernel to delete flow rules. The upcall handlers continuously translate OpenFlow rules to datapath rules and command the kernel to insert new flow rules. All of this busy work causes the extra system load.

Worryingly though, the additional system load is not attributed to the process causing it. In Kubernetes, a container can have a resource limit [19]. Flooding the system with upcalls from a resource-limited container would allow it to stress the system beyond its allowance.

Memory usage In both figure 3.3 and figure 3.4, the resident set size of `ovs-vswitchd` increases and stabilizes at slightly less than 2GB. The allocated memory is never released back to the operating system and stays allocated to the process even after the flooding stops. The used memory seems to be proportional to the flow table size.

Same as with the system load, memory usage bypasses resource accounting. A container or a VM with minimal resource allowance can thus use more resources than the system administrators configured. However, the memory usage of `ovs-vswitchd` is capped at a constant regardless of the number of containers used to flood the system. Therefore, this is not a significant attack vector.

Impact on latency

Figure 3.5 shows the results of a different run of the same experiment. The plot shows only the latencies measured from the `victim` pod. The ICMP tests target the node IP address of `kb1`, and the UDP tests target the `reflector` service provided by a pod on `kb3`. The horizontal lines mark the ranges of samples used for a statistical test.

While there does not seem to be any substantial difference between the stressed and non-stressed systems in the round-trip times, the non-stressed UDP samples ($n = 1876$, $\bar{x} = 175980 \pm 1320$ ns[4]) are smaller than the stressed UDP

[4] 95% confidence interval, assuming normal distribution

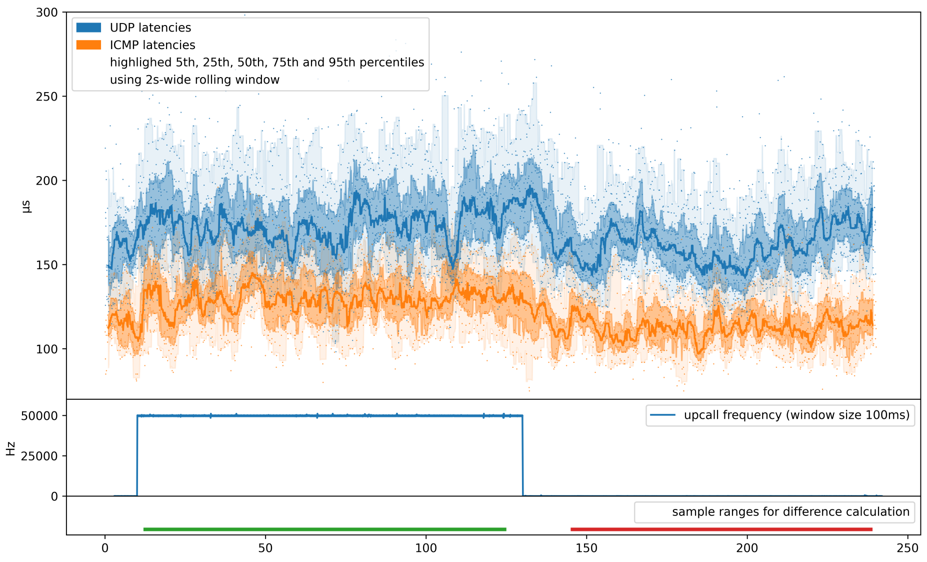

Figure 3.5 Round trip times during 50k upcalls/sec stress test

samples ($n = 2255$, $\bar{x} = 164404 \pm 1174$ ns) with statistical significance (Welch's t-test, $p = 2.17 \cdot 10^{-37}$). The difference between the means is 11576 ns.

3.4.2 Performance when resource limited

In section 3.4.1, we talked about extra memory and CPU usage of `ovs-vswitchd`. The default containerized deployment of OVN-Kubernetes limits `ovs-vswitchd` to 400 MiB of physical memory and two-tenths of a single CPU core. In the remainder of this section, when we write about limiting memory or CPU, we mean using Kubernetes to enforce the resource limit just as the default containerized deployment does it.

Memory limit

We limited the memory usage of `ovs-vswitchd`, leaving the CPU time unlimited. We used the default values for the memory limit, leaving us with the following configuration of the OVS container:

```
resources:
    requests:
        memory: 300Mi
    limits:
        memory: 400Mi
```

When we flooded the system with upcalls, ovs-vswitchd crashed without leaving any meaningful error message behind. During normal operations, everything seemed normal.

We tried to limit the number of flow rules using the following command:

```
ovs-vsctl set Open_vSwitch . other_config:flow-limit=10000
```

The limit helps and when the upcall frequency is not too high (double or triple the limit), everything works. However, once we flood the network with even more packets, the flow limit is overshot and ovs-vswitchd again crashes.

Our theory is that the crash happens during the revalidator dump phase, when ovs-vswitchd makes a copy of all in-kernel flow rules in the userspace. This allocates more memory than allowed and the container is subsequently terminated by the OOM killer. Quoting Kubernetes Documentation [20]:

> A Container can exceed its memory request if the Node has memory available. But a Container is not allowed to use more than its memory limit. If a Container allocates more memory than its limit, the Container becomes a candidate for termination. If the Container continues to consume memory beyond its limit, the Container is terminated. If a terminated Container can be restarted, the kubelet restarts it, as with any other type of runtime failure.

The decreased flow limit in the configuration of OVS does not help much to prevent crashes, because we can inject too many flows in between the revalidator thread runs (we can inject about 100k flow rules in 500ms, ovs-vswitch crashes even with less).

CPU limit

When CPU bound, ovs-vswitchd does not crash. It tries to do the opposite and dynamically changes the datapath's flow rule limit based on the revalidator thread's performance:

```
duration = MAX(time_msec() - start_time, 1);
udpif->dump_duration = duration;
if (duration > 2000) {
    flow_limit /= duration / 1000;
} else if (duration > 1300) {
    flow_limit = flow_limit * 3 / 4;
} else if (duration < 1000 &&
            flow_limit < n_flows * 1000 / duration) {
    flow_limit += 1000;
}
flow_limit = MIN(ofproto_flow_limit, MAX(flow_limit, 1000));
```

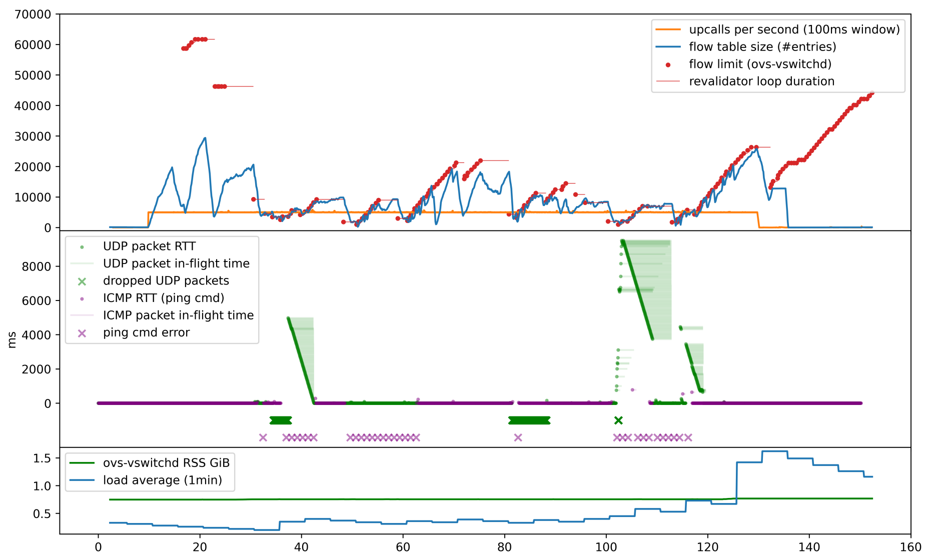

Figure 3.6 5k upcalls/sec stress test

We can see the effect of this code in figure 3.6. The red dots signify the dynamic flow limit. The horizontal lines going to the right of the dots signify the time the revalidator run took. We can see that a long revalidator run leads to a decreased flow limit.

When compared to the previous experiments:

- The rate of upcalls is only 5000 per second, one-tenth of the experiments before.
- Memory usage stays almost constant.
- Load average (1 minute) peaks at the value 2, but stays below 1 most of the time.
- The size of the datapath flow table varies a lot more. We can see the same higher frequency variations as before, but now they are combined with lower frequency variations created by the dynamically changing flow limit.
- Latency measurements are added to the plot (beware of the unusual units). The measurement method is the same as when we talked about latency before, now combined into a single experiment. The highest round-trip time observed is around 10 seconds long. Sometimes, sending a packet completely fails (indicated by thicker dots below the number line). The lines signify when the packet was in-flight.

The extremely high round-trip times can be explained by the upcall buffer queues. When the revalidator threads and other tasks use most of the available resources, the handler threads might not get scheduled and the packets in upcalls are buffered until the handler threads run again. This explains the regular packet delivery pattern (the green diagonal lines on the plot). They are caused by a regular packet-sending interval and a single instant when all the packets are delivered in one big batch.

Flood with 200k packets per second When we send 200k packets per second (close to the limit of our tool's single-threaded performance), `ovs-vswichd` does not crash, but no network traffic is getting through. The `victim` pod on the same host was unable to receive any packet.

Interestingly, when we stop the stress test, the number of flow rules still oscillates for a couple of seconds before dropping to normal levels. This indicates that our hypothesis about queues in front of the handler threads was correct and that there are a lot of packets queued for processing.

CPU and memory limit

When we use the default configuration with both CPU and memory limits, `ovs-vswitchd` is still killed by OOM killer under extreme stress (200k upcall generating packets per second), even though the memory usage stays below the limit most of the time. The crashes are however not reliable and sometimes, `ovs-vswitchd` runs normally for a while. The problem seems to be caused by a race condition between the flow limit calculation in revalidator threads and the handler threads inserting new flow rules. The crash happens when we change something - either at the beginning of a stress test or at the end, regardless of how long it is.

3.4.3 Overloading `ovs-vswitchd` without resource limits

The crashes of `ovs-vswitchd` led us to search for a possibility of a crash when overloaded without resource constraints. We tried to spawn multiple instances of our packet flooding tool, intentionally more threads than we had CPU cores available. On the 28-core dedicated test servers, we did not manage to crash `ovs-vswitchd` with anywhere between 1 and 80 instances of our stress tool.

We believe that `ovs-vswitchd` is safe from crashes as long as it runs without resource constraints. On our experimental clusters, `ovs-vswitchd` is automatically configured as a higher priority process than the ordinary processes and therefore it gets enough CPU time. But even if we manually decrease the priority to the same level or below ordinary processes, `ovs-vswitchd` does not crash.

Chapter 4

Discussion

4.1 Generality of our results

To test our results in the real world, we have arranged a meeting with a large computing infrastructure provider[1] using OVS. Their OVS instances are managed by OpenStack's Neutron [21] instead of our OVN-Kubernetes. They use VMs whereas we used containers. The biggest common factor between their system configuration and ours is the use of OVS.

We did not get access to the hypervisors (nodes of the cluster) and OVS directly, but we were given a full VM to run the tests. And for a limited time, we talked with one of their administrators who probed the hypervisors following our guidance. We have conducted the following experiments:

- First, we have confirmed the presence of OVS by running the flow eviction timeout experiment. The results on their system were as expected with an observable increase in latencies at the 10-second mark. Interestingly, the extra latencies caused by upcall processing were observable only within the cluster itself, not when communicating over the default gateway. We do not know why.

- We tried running our packet flooding tool with the varying MAC addresses. According to the administrator, it did not affect the number of installed datapath flows. However, due to an increase in the CPU load caused by QEMU processes, we hypothesize that the packets were dropped by `libvirt` before they reached OVS.

- After the unsuccessful stress test, we tried looking for packets causing upcalls and installation of new flow rules. Because we did not get access to

[1]intentionally unnamed

OVS itself, we had to rely on the administrator's observations. We did not run the experiment to its completion due to time constraints. But we still found that packets with VLAN tags (even nested) generate new flow rules, specifically drop rules for individual VLAN tags. Using this knowledge and a simple Scapy script, we managed to increase the flow table size from around 100 entries to 90000, but we could not get any higher than that. We were told that there are some QoS rules preventing network abuse that we might have run into. Another limiting factor might have been the limited performance of Scapy which prevented us from generating large enough traffic.

Based on this short experiment, we conclude that our findings apply to different OVS configurations and generalize well.

4.2 Possible OVS improvements

Throughout our experiments, we have identified several areas in OVS that we believe could be immediately improved upon:

- In in `ovs-vswitchd`, instead of a fixed flow eviction timeout, enforce the flow limit by using a PID controller and a variable eviction timeout.
- Free unused memory
- Remove the resource limits in OVN-Kubernetes default container configuration

Other areas require better investigation before they can be improved, but we believe that it is worth the effort:

- Gracefully handle out-of-memory conditions
- Investigate how best to prevent upcalls when using OVS as a firewall

4.2.1 Flow limit with a PID controller

In section 3.4.1, we explored how `ovs-vswitchd` enforces the limit of the number of flows in the kernel flow table. In section 3.4.2, we followed up with additional details on how the limit is calculated. We believe that a PID controller would prevent the unstable oscillating behavior we saw in figure 3.6.

There are multiple possible implementations and we do not have enough data to say which one would work the best. We would suggest doing one of the following:

- There could be a single PID loop taking the runtime of the revalidator threads as input. The output would be the current size limit of the flow table. Another PID loop would take the real number of flows in the flow table and output the flow eviction timeout.

- A single PID loop could generate the eviction timeout. An additional emergency mechanism would be needed to prevent accidental overfilling.

4.2.2 Free unused memory

In section 3.4.1, we observed that memory is never released in `ovs-vswitchd` after it has been allocated. It is technically not leaked, because `ovs-vswitchd` will reuse the memory whenever there is a need. But under normal circumstances, there are not that many flows.

While there could be reasons for not releasing the allocated memory (e.g. it would require synchronization), we think that it could be a problem in deployments on hardware with limited resources.

Alternatively, the maximum number of flows could be by default dependent on the amount of system memory. Defaulting to using 10% of system memory at most seems as a better solution compared to allocating fixed 2GB.

4.2.3 Default OVN-Kubernetes configuration

In section 2.2.1 we talked about the default resource limit of OVS deployment in OVN-Kubernetes. We suggest completely removing the limit. Unless the crashes and extremely high latencies can be prevented, limiting OVS's resources currently introduces more problems than it solves.

To limit memory usage without strict limits, default maximum size of the flow table could be decreased from 200k to a small fraction of it.

4.2.4 Out-of-memory conditions

In section 3.4.2, we observed crashes of `ovs-vswitchd` when we limited the available physical memory. While we understand, that reacting to limited memory is not simple, we believe that a graceful degradation of functionality would be preferred when it comes to networking infrastructure. Moreover, there seems to be a correlation between the number of flows in the flow table and memory usage. It should be possible to limit the number of flows not only based on computation time but also based on the amount of available memory.

While appealing in theory, we think that a practical implementation would require better insight into how memory is allocated and used. Therefore improvement of this requires further research.

4.2.5 OVS as a firewall

In section 3.3.2, we have found a design flaw in OVS that prevents OVS from effectively blocking traffic. OVS allows only positive matches in its datapath flow rules. Negative wildcard matches can currently be handled only in userspace. Out of the issues we have discovered, we believe that this is the most complicated one to fix.

Fix without changing the datapath interface

Changing the datapath interface might be undesirable because a similar interface is also implemented in hardware [22]. Additionally, the hardware offloaded processing is very likely to have the same issue and we want to prevent upcall generation there as well. Therefore, we might want to generate ordinary datapath flow rules which would drop the invalid packets. This is however impossible without eliminating the possibility of a significant growth in the number of flow rules.

Exponential upper bound In general, let us assume an OpenFlow table with a default drop rule and several exact match rules accepting the packet (e.g. resubmitting it to a different table). Every rule matching a packet is equivalent to a logic formula where the variables represent bits of the flow key. Specifically, the formula is a conjunction. The whole table can be therefore expressed as a single DNF formula matching valid packets. Let us call the formula F.

Now, let us express the formula matching packets that should be dropped. $\neg F$ is true for all invalid packets. However, it is not in DNF anymore, so we cannot translate it directly back. Using De Morgan's laws, we can rewrite $\neg F$ as a CNF formula of the same length. Then, we have to convert it back to DNF. However, the conversion of a CNF formula to a DNF formula could potentially lead to exponential growth in the number of clauses. Therefore, a potentially exponential explosion in the number of flow rules we need to drop the packets.

Practical implementation It might still be practical to use this technique to implement simple port security and allow only a single MAC address on a switch port. That would require only 49 rules - 48 to drop invalid packets, a single rule for recirculation of valid packets. However, expressing anything more complicated is probably impractical due to drops in performance.

Fix with a change in the datapath interface

If we allow ourselves to change the datapath interface and in turn the kernel module, the fix can be implemented relatively easily. We could leave the processing as is and add packet filtering before the datapath makes an upcall. For example, in the kernel module, we could add a second, negative, flow table for upcall filtering. Any packet it would match would get dropped and not sent to the user space. The used flow key and most of the other data structures could stay the same, only the interface would have to include new commands for manipulating the negative flow table.

4.3 Recommendation for public cloud providers

Our findings certainly impact public cloud providers using OVS. We would recommend at the very least:

1. Verify, that `ovs-vswitchd` is running without any resource constraints.
2. Monitor the number of flows in the flow tables (i.e. `ovs-dpctl show`) and trigger a warning whenever the number of flows reaches the limit.

In case the cloud provider uses OVS for firewalling, we would recommend switching to some other technology. OVS is in our opinion currently unsuitable for blocking undesirable traffic with high performance.

4.4 Results coverage

As we mentioned before, the SDN networking stack consists of multiple complex components all interacting with each other. The problems we identified are by no means the complete list of problems of SDNs using OVS. We can only say that we found some problems and there might still be many more.

If we limit ourselves to OVS only, the same still applies. We cannot say with high confidence that our research covered all possible issues caused by pathological traffic patterns. There might still be more to discover.

4.5 Future work

We suggested practical improvements to OVS in section 4.2. There is also still space for more theoretical and experimental research, even though we do not know about any directions likely to yield significant results. We did not pursue investigations into the following areas and they may be a potential topic for additional research:

- We have not verified the translation mechanism between OpenFlow and datapath flow rules in `ovs-vswitchd` for optimality. There is a possibility that inefficiencies in the translation mechanism could be abused to generate upcalls similar to our results.
- Similarly, we have not explored what happens during network reconfiguration and whether the transition can cause performance issues.

Conclusion

We started our research with the goal of finding traffic patterns causing performance issues in OVS and OVN-Kubernetes. We successfully achieved this goal by discovering OVN's port-security feature, which can be abused to generate an upcall and install a new flow rule for every received packet. We showed that this behavior cannot be easily prevented and that it requires either changes to the datapath interface or changes to flow rule generation in `ovs-vswitchd`.

Additionally, using our findings, we stressed OVS with upcalls and discovered, that the default resource limits for OVS in OVN-Kubernetes cause more problems than they prevent. Namely, they lead either to crashes of OVS due to unhandled out-of-memory conditions or complete network denial-of-service when `ovs-vswitchd` is CPU limited. With unconstrained resources, OVS handled the stress test rather well.

Unfortunately, due to the inherent complexity of OVS and software-defined networks in general, we cannot rule out the existence of other performance issues in the OVS's kernel datapath. In particular, we cannot prove that it is not possible to generate different kinds of packets which will reliably generate upcalls. As we have seen with our results, the problem is directly caused by OpenFlow configuration generated with an external tool to OVS (e.g. OVN). While this led us to discover a design flaw in OVS, it might be possible to directly exploit a specific set of OpenFlow rules to generate upcalls and stress OVS in a similar manner.

As a result of our work, we have outlined the possible improvements in chapter 4. We do not know whether our suggestions are directly applicable as improvements, however, at the very least we provide a list of areas in OVS's code base that could be improved.

Bibliography

[1] *Container Network Interface (CNI) Specification.* 2022. URL: https://github.com/containernetworking/cni/blob/dc0779e8/SPEC.md (visited on 06/27/2023).

[2] Open Networking Foundation. *SDN architecture, issue 1.* 2014. URL: https://opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf (visited on 06/12/2023).

[3] Open Networking Foundation. *OpenFlow® Switch Specification.* URL: https://opennetworking.org/sdn-resources/openflow-switch-specification/ (visited on 06/12/2023).

[4] *OVS manpage: ovs-actions (7).* URL: https://www.man7.org/linux/man-pages/man7/ovs-actions.7.html (visited on 06/19/2023).

[5] *Open vSwitch website.* URL: http://www.openvswitch.org/ (visited on 06/12/2023).

[6] OVS contributors. *Open vSwitch source code repository.* URL: https://github.com/openvswitch/ovs (visited on 06/19/2023).

[7] Linus Torvalds et al. *Linux kernel source code: The openvswitch module.* URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/openvswitch?h=v6.2 (visited on 06/19/2023).

[8] *RedHat OpenStack documentation: Open Virtual Network.* URL: https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/13/html/networking_with_open_virtual_network/open_virtual_network_ovn (visited on 06/12/2023).

[9] OVN contributors. *Open Virtual Network (OVN) source code repository.* URL: https://github.com/ovn-org/ovn (visited on 06/19/2023).

[10] *OVN-Kubernetes source code repository.* URL: https://github.com/ovn-org/ovn-kubernetes (visited on 06/19/2023).

[11] Ye Yin. *The introduction to OVS architecture.* 2018. URL: https://hustcat.github.io/an-introduction-to-ovs-architecture/ (visited on 06/22/2023).

[12] *Data Plane Development Kit (DPDK).* URL: https://www.dpdk.org/ (visited on 06/28/2023).

[13] *The Linux kernel documentation: AF_XDP.* URL: https://www.kernel.org/doc/html/latest/networking/af_xdp.html (visited on 06/28/2023).

[14] *OVS documentation: Flow Hardware offload with Linux TC flower.* URL: https://docs.openvswitch.org/en/latest/howto/tc-offload/ (visited on 06/28/2023).

[15] OVN contributors. *OVN-Kubernetes installation guide.* URL: https://github.com/ovn-org/ovn-kubernetes/blob/master/docs/INSTALL.KUBEADM.md (visited on 06/12/2023).

[16] *OVS documentation: USDT probes.* URL: https://docs.openvswitch.org/en/latest/topics/usdt-probes/ (visited on 06/20/2023).

[17] Eelco Chaudron. *Investigating the cost of Open vSwitch upcalls in Linux.* 2022. URL: https://developers.redhat.com/articles/2022/02/07/investigating-cost-open-vswitch-upcalls-linux (visited on 06/12/2023).

[18] *OVN manpages: ovn-nb (5).* URL: https://www.man7.org/linux/man-pages/man5/ovn-nb.5.html (visited on 06/12/2023).

[19] *Kubernetes Documentation: Resource Management for Pods and Containers.* URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ (visited on 06/12/2023).

[20] *Kubernetes Documentation: Assign Memory Resources to Containers and Pods.* URL: https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/#exceed-a-container-s-memory-limit (visited on 06/12/2023).

[21] *OpenStack Documentation: Neutron.* URL: https://docs.openstack.org/neutron/latest/ (visited on 07/04/2023).

[22] *Open vSwitch hardware offloading.* URL: https://docs.openstack.org/neutron/rocky/admin/config-ovs-offload.html (visited on 06/12/2023).

Appendix A

`ovs-vswitchd` container image

A.1 Build

There is a script `build_ovs_container.sh` attached to this thesis. Running this script will download and compile OVS, OVN and OVN-Kubernetes and at the end, wrap the result into a container usable for deployment into a cluster. We recommend reading the script first before running it. It is not long and it might be desirable to tweak it.

To run the script, make sure that you have these dependencies installed on your system:

- `podman`, ideally configured for rootless operation
- `git`
- `go` compiler

The script creates a container image by default called `ovn-kube-f:latest`. To use it further in the cluster, push it to an accessible container registry[1] and note its fully qualified name.

A.2 Our changes to OVS

All changes we have made to OVS can be seen in the `ovs-usdt-probes.patch` file. We have added only new USDT probes. They are compiled only as `nop` instructions and should not significantly impact the performance unless used.

[1]We run a private registry for this purpose.

Appendix B

Cluster installation instructions

Start by preparing 3 standard Fedora 38 installations, all in a single LAN, ideally with 2 network cards as described in section 2.1. All systems should have an accessible root shell. Any time we write about running a command, we always assume it is run from the root shell.

In the attachment, there are two scripts prefixed with `setup`. First, we advise editing both of the scripts:

- Edit the `setup1-general.sh` script and change the network configuration function, `configure_systemd_networkd`, to match your network environment. Especially, the names of your network interfaces.

- Edit the `setup2-master.sh` script and change `$IMAGE` variable with the qualified name of your OVS container (instructions on how to build it in appendix A).

Once you changed the script, follow these steps:

1. Run the `setup1-general.sh` script on all machines. Provide `kb1`, `kb2` or `kb3` as an argument to set the hostnames.

2. Wait for all the systems to reboot.

3. On the machine you chose as `kb1`, run the `setup2-master.sh` script.

4. On `kb1`, run `kubeadm token create --print-join-command`

5. Append the `--cri-socket=unix:///var/run/cri-dockerd.sock` option at the end of the output of the previous step and run the command on both `kb2` and `kb3`.

6. Wait for the cluster to initialize, i.e. the command `kubectl get nodes` on `kb1` should show that all 3 nodes are in the `Ready` state.

7. Copy pod specification files from the `kube_configs` directory to `kb1`. Edit the `reflector.yaml` file and change the image reference to your build of the `analyzer` container (see appendix C). Create the pods by calling `kubectl create -f $file`.

8. Check the pods deployment status using `kubectl get pods`. Make sure that all pods are in the `Ready` state before proceeding.

9. At this point, you can run the experiments. To get a root shell in the `arch` pod, use the `kubectl exec -ti arch -- /bin/bash` command. Similarly for the `victim` pod.

Appendix C

analyzer - the tool for running experiments

C.1 Build

The source code is located in the analyzer directory in the attachment. Only the Rust toolchain is required for compilation:

```
# in the analyzer/ directory
cargo build --release --target=x86_64-unknown-linux-musl
# the executable will be located at
#    analyzer/target/x86_64-unknown-linux-musl/release/analyzer
```

Continue by building the container image:

```
# again in the analyzer/ directory
podman build -t analyzer .
```

Push the image to your container registry of choice and remember the qualified container name. The image is required for the reflector pod, more in appendix B. Also, copy the binary to the $PATH on all nodes and pods.

C.2 Usage

The analyzer is generally a collection of smaller tools, all invoked via subcommands. The tool can be always supplied with the --help option and it will print out a help message. The following subsections will describe how to use the analyzer for the discussed experiments.

Data collection The measurement results are stored in files in the `analyzer`'s current working directory. Multiple files are usually created. The file names start with a human-readable identifier of the data series. The second part of the file names is a timestamp (identical for all files created in a single `analyzer` run).

The `analyzer` can upload results to an HTTP server when provided with the `--push-results-url` argument. When the URL is provided, for every data file it creates, the `analyzer` calls the `curl -T [FILE] {URL}` command.

Data format The result files are usually using the CSV format. All data series use the same time source (see section 2.2.5). Therefore, all measurements can be easily aligned even when the collected data came from different `analyzer` runs (i.e. in a pod and on a host at the same time).

C.2.1 Eviction timeout measurement

The result of this experiment is a single CSV file with timestamps and the measured round-trip times.

```
# on arch (pod)
analyzer randomized-eviction-timeout \
--target-ip 192.168.1.221 \
--count 3
```

C.2.2 Packet fuzzing

This experiment creates multiple data files, not all of which are relevant. From the `node-logger` subcommand, we are interested in upcall statistics located in the `kernel_flow_table_trace*.csv` file. The `tags*.jsonl` file from the `packet-fuzz` subcommand provides us with timestamps allowing us to separate the upcalls into categories.

```
# on kb2
analyzer install-dependencies
analyzer node-logger --only-upcalls

# on arch (pod)
analyzer install-dependencies
analyzer packet-fuzz
```

C.2.3 Packet flood

In this experiment, we were not looking for anything in particular, therefore all the result files can be of interest.

```
# on arch (pod)
analyzer packet-flood --count 500000

# on victim (pod)
analyzer victim

# on kb2
analyzer install-dependencies
analyzer node-logger --only-upcalls
```

C.3 Processing of results

Scripts for creating plots are located in the `analyzer/postprocessing` directory. To use them, it is necessary to install Python, matplotlib, Polars, Pandas, numpy and scipy.

There is a script for every experiment type. The packet flood experiment has multiple similar scripts with slightly differently preconfigured plots, the others have only a single script. The scripts, especially the packet-flood related, contain extra commented-out code to plot additional data.

All scripts expect two arguments - a path to a directory containing all results from a single experiment and the name of the output file.