UMON: Flexible and Fine Grained Traffic Monitoring in Open vSwitch

An Wang
George Mason University
awang10@gmu.edu

Yang Guo
NIST
Yang.Guo@nist.gov

Fang Hao
Bell Labs, Alcatel-Lucent
Fang.Hao@alcatel-lucent.com

T. V. Lakshman
Bell Labs, Alcatel-Lucent
T.V.Lakshman@alcatel-lucent.com

Songqing Chen
George Mason University
sqchen@cs.gmu.edu

Abstract
We study how to provide fine-grained, flexible traffic monitoring in the Open vSwitch (OVS). We argue that the existing OVS monitoring tools are neither flexible nor sufficient for supporting many monitoring applications. We propose UMON, a mechanism that decouples monitoring from forwarding, and offers flexible and fine-grained traffic stats. We describe a prototype implementation of UMON that integrates well with the OVS architecture. Finally, we evaluate the performance using the prototype, and illustrate UMON’s efficiency with the example use cases such as detecting port scans.

CCS Concepts
● Networks → Network monitoring; Network manageability; ● Security and privacy → Intrusion/anomaly detection and malware mitigation;

Keywords
SDN; Open vSwitch

1. INTRODUCTION
Fine-grained network traffic monitoring is an important capability for effective network management. Monitoring is typically done at routers, with the collected info being reported to a central collector where the network management applications are running. The scalability has been the main challenge for the network traffic monitoring. A physical router supports the ever increasing switching speed, and routes a large number of flows that easily surpasses millions. Under the limited memory and CPU resource available in a physical router, collecting fine-grained traffic stats required by various management applications accurately and speedily is extremely challenging. Various approaches, from packet sampling [1, 4], probabilistic based stats collection [7], to hardware enhanced solutions [14], have been developed to combat the scalability issue.

The Open vSwitch (OVS) [3] is a popular software switch widely employed by Software Defined Networks (SDN). OVS runs on a general purpose computer and acts as the edge router for the virtual machines (VMs) hosted on the same machine. Compared to a physical core router, an OVS routes at a slower speed, encounters a smaller number of flows, and has access to much more memory and CPU resources. The fine-grained traffic monitoring at an OVS is thus feasible. In addition, because the flows monitored at an OVS are either originated or terminated at the VMs, some management functionality, e.g., intrusion/anomaly detection, can be implemented at the edge, further reducing the stats collection overhead.

In this paper, we study how to provide fine-grained, flexible traffic monitoring in the OVS. The current OVS provides monitoring functions similar to physical routers, e.g., Netflow [1], sFlow [4], SPAN, and RSPAN. SPAN and RSPAN capture all packets going through a switch by port mirroring. While complete, such monitoring by mirroring has considerable costs for comprehensive monitoring. Alternately, NetFlow and sFlow utilize sampling to reduce overheads for monitoring. This is not appropriate for use cases that may want to track every packet of a flow - for security applications sampling
may miss needed information such as TCP SYN packets [15]. Recently, there has been a strong push [13, 16] to use packet and byte counts associated with flow entries in SDN switches to provide fine-grain visibility into network traffic. In [13], the authors show that by dynamically installing flow entries in SDN switches various monitoring tasks such as heavy hitter, hierarchical heavy hitter, and change detection can be done. The drawback here is that this type of flow monitoring is strongly linked to entries in the flow-table and hence implicitly to entries which are primarily meant for packet forwarding. The interrelated forwarding entries and monitoring entries force to combine the forwarding logic with the monitoring policy, which is not trivial as pointed out in [5, 9, 12].

We argue that monitoring packet and byte counts by association with flow entries in the forwarding table is neither flexible nor sufficient for supporting many monitoring applications. One reason is that the header fields that are of interest for packet forwarding may not always overlap with those that are of interest for monitoring. The chances of no overlap are likely to increase further as the number of header fields continues to grow beyond the 40 or so [6]. Furthermore, flows that require monitoring using flow definitions different from those used for forwarding may cause packets to be sent to the controller resulting in sub-optimal packet handling. For instance, define the subflows to be the fine-grained flows that belong to a mega-flow. Individual flows from differentSrcIPs to destination host A are subflows of the mega-flow defined by the rule DstIP=A with other routing fields being wildcarded. Subflow monitoring is useful for many applications, e.g., port-scan detection, heavy hitter identification, etc. If a user wants to monitor packet counts of individual subflows, then inserting a single mega-flow rule of DstIP=A does not work since it can only report the aggregated packet count. Subflow rules need to be installed reactively as the flows appear since the specific subflows are not known in advance. This forces the switch to send the first packet from every subflow satisfying DstIP=A to the central controller. The controller then installs the corresponding subflow rule into the SDN switch for monitoring purpose even though this rule is not needed for forwarding.

To address aforementioned issues in monitoring flexibility and granularity, we propose UMON, a user-defined traffic monitoring framework that decouples monitoring from forwarding. UMON defines a monitoring flow table in the OVS to separate monitoring rules from forwarding rules. Users can thus freely install monitoring rules without worrying about the possible interactions with forwarding rules. We also increase the monitoring granularity by providing packet counts based on the non-routing fields, e.g., TCP SYN, TCP FIN, etc., and enable the subflow monitoring that automatically monitors all subflows of a mega-flow without constant controller involvement. The fine-grained monitoring enables many local management applications, e.g., the port-scan detection and DDoS attack detection [15], in the OVS. In short, our contributions are three-fold: (1) we propose UMON that decouples monitoring from forwarding, and offers flexible and fine-grained monitoring in OVS; (2) we design and implement UMON that integrates well with the current OVS architecture. We also implement a port-scan detection module in the OVS as an example of local management applications; and (3) we conduct experiments using the prototype. The results show that UMON does not come at the cost of unacceptable memory and computational overhead.

The paper is organized as follows. Related work is summarized in Section 2. The UMON design and implementation are described in Section 3. Results from evaluation of UMON performance are presented in Section 4. Concluding remarks are in Section 5.

2. RELATED WORK

Flow-rule based measurements in conjunction with dynamic granularity adjustment have been used for anomaly detection [13, 16, 11]. The work in [11] shows that such dynamically adjusted measurements can improve DoS attack detection. In [16] an SDN based dynamic flow rule measurement framework is proposed. TCAM size limitations are considered in [13] where the authors propose making dynamic adjustments to TCAM entries that are used for individualized measurements without violating user-specified levels of accuracy. UMON is complementary to this prior work in providing mechanisms for monitoring user-defined flows on many fields and in reducing controller involvement.

The Frenetic project has developed networking programming languages, e.g., Pyretic [12], that use the high level of abstraction to enable the creation of modular SDN application software. Sophisticated SDN applications can be programmed using Python-like language without worrying about the complex interaction among them. The network measurement, for instance, can be programmed as a SDN application using Pyretic. However, such a network measurement application is running on top of a controller platform, thus it does not eliminate the constant involvement of the controller when conducting sub-flow monitoring. In addition, UMON allow more fields to be monitored and to run edge management functions. In a nutshell, Pyretic employs the traditional OpenFlow APIs, while UMON extends the

1The TCP flags were not routing fields when we conducted this study. They have been included in Open vSwitch 2.3.1. UMON, however, offers the flexibility to monitor any interested fields.
OpenFlow to facilitate the sub-flow monitoring, additional field monitoring, and edge management functionality. The detailed design of OpenFlow API for UMON is not included in this paper and will be addressed in the future work.

3. UMON DESIGN AND IMPLEMENTATION

UMON is designed for flexible and fine grained monitoring of user-defined flows in SDN. It allows users to flexibly define the flows to be monitored, and to freely choose the statistics to be collected. In addition, the design fits well with the OVS architecture. The UMON design strives to achieve three goals: (1) decoupling monitoring from forwarding; (2) supporting traffic monitoring based on non-routing fields; and (3) supporting sub-flow monitoring. Below we describe the UMON design in details.

3.1 Decoupling monitoring from forwarding

A straightforward way to achieve the decoupling is to have a separate monitoring flow table, where the monitoring rules are stored. The monitoring flow table bears the similarity with a flow routing table. The monitoring flow table consists of the monitoring rules while a routing table consists of the routing rules. Both monitoring rules and routing rules contain the matching pattern that determines if a packet matches a rule. The monitoring table, however, differs from a routing table in several aspects. First, in the routing table, the goal is to find the matching rule with the highest priority, while in the monitoring table, all matching monitoring rules need to be located and their stats need to be updated. Second, the actions supported in the routing table and in the monitoring table are different (see Section 3.2 and 3.3 for details). Finally, the monitoring table also supports sub-flow monitoring and edge management functionality.

Upon the arrival, a packet passes through the forwarding table to decide its routing. The packet is then presented to the monitoring table to collect stats. However, OVS adopts a unique two-tiered forwarding design, with the full-blown routing function being implemented in the user space, and the active forwarding entries being cached in the kernel for the performance improvement. Thus merely adding a monitoring flow table in the user space is not adequate.

The OVS kernel module contains one or multiple datapath modules, where each datapath is like a bridge. There is one flow table for each datapath. Flow table entries are generated and installed by the user-level routing module. A flow table entry contains three elements: matching rule, counters, and actions. The matching rule contains a key with 12 fields from the packet header that includes L2 and L3 addresses, ports and so on. Earlier OVS versions only allow micro-flow rules in the datapath which implies that all fields in the key must be used for flow matching. Mega-flow rules have been allowed since version 1.11.0. Mega-flow rules allow rule-aggregation by wildcard matching, which significantly reduces the flow table size. The counter includes both packet and byte counts. A flow entry also contains a set of actions such as Output (i.e. forward), Drop, etc. so that the packets in this flow are properly handled. The flow rules in the kernel do not have priority and a packet matches at most one flow rule.

The user-level ovs-vsportd module contains a flow table pipeline for routing (see Fig. 1) [3]. User-level routing supports priority-based wildcard matching: if a packet is matched with a higher priority rule, the lower priority rules in the same table are ignored. When a packet does not match any kernel rules, it is delivered to the user space. The user-level routing threads, called handler, start the lookup in Table 0. Based on the matching result, an action may be added and the packet may be submitted to another table in the pipeline for further processing. After the packet is processed according to the pipeline, the final action set associated with the packet is applied to the packet, so that the packet can be processed (forward, drop etc.) accordingly. The flow rule containing the packet header, the mask, and the action set, is installed at the kernel flow table.

![Figure 1: Pipelined flow tables in OVS.](image)

If no match is found, the default table miss action is invoked, which is typically configured to send the packet to the OpenFlow controller as a Packet_In message. The OpenFlow controller decides how to process this packet based on the programmed policy. A Packet_Out message is sent back to the switch, with the action list indicating how the packet should be processed by the switch. The controller may further issue a Flow_Mod message to add new flow entries into the OVS flow table pipeline. Note that such rules will be matched and installed into the kernel only when the next packet in the same flow arrives.

- **Decoupling in the user space.** A monitoring table as shown in Fig. 2 can achieve the decoupling in the user space. The monitoring table is similar to an OVS forwarding table and stores the monitoring flow entries that can be added or removed using utilities or via the commands from the OpenFlow controller. A monitoring flow entry has the same main components,
the matching rule, counters, and instructions, as a forwarding flow entry, with an extra pointer pointing to the table that can store the monitored subflows. The matching rule defines the flow or mega-flow that need to be monitored. The monitoring flow entry contains the `monitoring actions` as defined in the Section 3.2 and the Section 3.3. After a packet passes through the pipelined routing table, it goes through the monitoring table so as to collect the required stats.

\[ m_f^* \triangleq m_f \mid (\forall i \in I_f, m_i), \]

where
\[ I_f \triangleq \{ i \mid r_f \& m_f = r_i \& m_f,i,i \in I \}, \]

and \( i \in I_f, m_i \) is bitwise OR of \( m_i, i \in I_f \). In Eqn(2), \( r_f \& m_f = r_i \& m_f \) indicates the packet header bits unwildcarded by both the flow rule \( r_f \) and the monitoring rule \( r_i \) are equal, implying that the flow rule \( r_f \) and the monitoring rule \( r_i \) can potentially overlap with each other. The set \( I_f \) includes all monitoring rules that may overlap with the flow rule. The adjusted mask for the kernel rule, \( m_f^* \), is the bitwise OR (\( \{} \) of its original mask \( m_f \) and the masks of all potential overlapping monitoring rules. Thus the adjusted kernel flow rule \( (r_f, m_f^*) \) is finer than any overlapping monitoring rules, allowing to collect the required monitoring stats by mapping one or multiple kernel rules to a monitoring rule.

In OVS, the user-level ovs-vsctl module controls and interacts with the kernel module via the `netlink` interface. The main functions of ovs-vsctl are carried out by two types of threads: `handler` and `revalidator`. `Handler` contains the packet processing logic in user space, while `revalidator` manages the kernel flow table and retrieves flow stats from the kernel to user space. We modified the `handler` thread to handle the monitoring table and kernel rule generation, and modified the `revalidator` thread to collect the stats for the monitoring entries in the monitoring table.

### 3.2 Traffic monitoring of non-routing fields

We introduce new monitoring actions, Field Monitoring Action, to support collecting stats based on non-routing fields. For instance, SYN Monitoring Action reports the number of SYN packets associated with the flow entry. ACK Monitoring Action reports the number of ACK packets associated with the flow entry. The Field Monitoring Action is implemented analogous to other existing actions – it can be added to the action set of a flow entry, which is then installed in the kernel flow table. Whenever a packet matches the flow entry, the action is executed and the corresponding packet count is increased.

In the current OVS implementation, there is already a data structure that keeps track of the packet and byte counts of flow rules. We expand this data structure, in
both user space and kernel to include the packet count of monitored fields. We also modify the code to support new Field Monitoring Action, and collect the stats and save them into the updated data structure at both the user and kernel levels.

3.3 Subflow monitoring

Subflows are the fine-grained flows that belong to a mega-flow as defined by the monitoring rule. For instance, flows from the host A to the host B to different destination ports are subflows of the mega-flow that is sent from the host A to the host B. We introduce the Subflow Monitoring Action for the subflow monitoring. The subflows are defined using the subflow mask $s_i$. For a new flow with the packet header $r_f$, we first examine if $r_f$ falls into the mega flow of the monitoring rule $i$ by examining if $r_f$ \& $m_i = r_i \& m_i$. If true, flow $r_f$ needs to be monitored. We next check if the flow $r_f$ has already been monitored as a subflow. For the given monitoring rule $(r_i, m_i)$, we maintain a subflow table to keep track of individual subflows (see Fig. 2). We loop through the entries in the subflow, and check if $r_f$ \& $(m_i|s_i)$ is already in the subflow table. If yes, the stats of $r_f$ is counted toward the matching entry. If no, a new subflow entry of $r_f$ \& $(m_i|s_i)$ is added.

We next describe how to generate the kernel flow rule with the Subflow Monitoring Action on. The value of $m_i|s_i$ is the subflow mask. The kernel rule needs to be at the subflow granularity. Hence:

$$m_f^* = m_f \mid (\{i \in I_f \mid m_i \mid s_i\}).$$

(3)

Note that the subflow monitoring does not change $I_f$ as in Eqn.(2) because the union of all subflows is the original mega-flow. If a monitoring rule has no Subflow Monitoring Action, $s_i$ is set to be all zeros $m_i = m_i|s_i$.

3.4 Monitoring rule insertion/deletion

When a new monitoring rule $(r_n, m_n)$ is added, the revalidator will examine every kernel flow rule as follows. Denote by $(r_f, m_f^*)$ a kernel flow rule. Let $m_f^*|s_n = m_n$. The kernel flow rule interacts with the new monitoring rule if and only if $r_f \& m_f^*|s_n = r_n \& m_f^*|s_n$. If the kernel flow rule does not interact with the new monitoring rule, nothing needs to be done with this rule.

If the kernel flow rule interacts with the new monitoring rule, the new mask for the kernel rule is computed as $m_f^*|m_n$. If the new mask is “finer” than the original mask $m_f^*$, i.e., some fields are un-wildcarded in the new mask but not in the original mask, this rule is removed from the kernel. An appropriate new flow rule will be installed when the next packet misses the kernel flow table and is handled by the user-level handler. If the kernel flow rule’s mask remains the same, we further check if the kernel rule is used by the new monitoring rule for the stats collection. If no, nothing needs to be changed. If yes, we check if the kernel flow rule includes all the monitoring actions required by the new monitoring rule. If not, the missing actions will be added.

When removing a monitoring rule, we employ a lazy scheduling approach, and make no immediate change because the kernel flow rules have adequate granularity. We let these rules time out gradually and the appropriate new rules will be installed in the future.

4. EVALUATION

We run an Open vSwitch (version 2.3) on a stand-alone machine with 2.67GHz CPU (12 cores), 64G memory, and an Intel NIC of two 10G ports. We connect two machines to these two ports, one serving as the packet generator and the other as the sink. Both of them run Ubuntu 3.0.0-12-server kernel with the same hardware configuration as the first one. To generate packets at high throughput, we use DPDK pktgen [10] to replay the network pcap traces.

4.1 UMON overhead evaluation

To evaluate the overheads introduced by UMON, we compare the performance of the UMON-capable vSwitch with the default vSwitch and the default vSwitch configured to use only microflows in the kernel. The evaluation is driven by the pcap trace from DEFCON [2], with 272 hosts and an average packet size of 900 bytes. There are 4432 microflows in the trace using matching keys as defined by OpenFlow. In the experiments, we set the traffic sending rate to be around 2.2 Gbps so that no packet loss occurs for the default vSwitches. We randomly select 150 hosts to be monitored with the Subflow Monitoring Action on using the UMON monitoring table. The subflow mask is set to be all ones, i.e., microflows are monitored.

Figure 3 and Fig. 4 show the throughput and the packet rate of UMON enabled vSwitch. The results for default vSwitches with and without turning on microflow are similar. We collect the rates at the receiving port.
Throughput (Mpps)

Figure 4: Packet rate (Mpps) of UMON (on the NIC connecting to the traffic generator), at the vSwitch, and at the transmission port (on the NIC connecting to the sink) every five minutes, shown as $Rx$, $OVS$, and $Tx$ in the figures, and continue the experiments for 30 rounds. All three types of vSwitches can handle the arriving traffic without loss. Note that the throughput observed at the vSwitch is slightly lower than that at the receiving and transmission port. This is due to the fact that the Generic Receive Offload option (GRO) at the NIC is enabled by default. GRO aggregates multiple packets from the same session into a large packet so as to reduce the number of packets that has to be processed by the network stack. During the process, the headers of small packets are stripped off, which causes this small discrepancy. This is also evident in Fig. 4, where a vSwitch always observes a smaller number of packets due to this effect.

Table 1: CPU utilizations, kernel flow table size, and missed packet rate

<table>
<thead>
<tr>
<th></th>
<th>OVS</th>
<th>Microflow</th>
<th>OVS</th>
<th>UMON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handler</td>
<td>0.0%</td>
<td>0.15%</td>
<td>0.21%</td>
<td></td>
</tr>
<tr>
<td>Revalidator</td>
<td>0.60%</td>
<td>6.8%</td>
<td>9.9%</td>
<td></td>
</tr>
<tr>
<td>FlowTableSize</td>
<td>295</td>
<td>4381</td>
<td>4301</td>
<td></td>
</tr>
<tr>
<td>MissPktRate</td>
<td>0</td>
<td>30 pkt/sec</td>
<td>26 pkt/sec</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 reports the CPU utilizations of the handler threads, the revalidator threads, the kernel flow table size, and the missed packet rate (pkts/second). The machine running the vSwitch has 12 cores and vSwitch automatically allocates eight handler threads and four revalidator threads. The reported CPU utilization is the sum of all the same type threads unless indicated otherwise. Memory used is almost constant and so it is not reported here. As expected, the UMON vSwitch and the microflow enabled vSwitch use much larger kernel flow tables than the default vSwitch which uses aggregated (mega-flow) rules. The microflow enabled vSwitch and the UMON vSwitch also have some missed packets, i.e., the packets that the kernel does not know how to route. The CPU utilizations are low for all three types of vSwitches. The revalidator threads consume much more CPU resources than the handler threads. The revalidator’s CPU utilization of the UMON vSwitch is the highest, and reaches 9.9% of a core, compared to 0.6% of the default vSwitch. This is due to the monitoring activities of all microflows associated with the 150 hosts. The CPU utilization of UMON vSwitch, however, is very low given the available CPU resources on a powerful machine with multiple cores. Even if we increase the frequency of running the revalidator thread (by default it runs once every half second; the handler wakes up when there are missed packets), the CPU utilization is still acceptable.

4.2 Effect of monitoring rules

We also investigate the impact of the monitoring table size on CPU utilization. We increase the number of monitored hosts from 30 hosts to 272 hosts, at which point all hosts are monitored. Fig. 5 depicts the CPU utilizations of the handler threads and the revalidator threads. The curve is not exactly linear as individual hosts have different number of flows. Again, the revalidator threads consistently use much more CPU than the handler threads. In addition, as the number of monitored hosts increases, the CPU utilization of the revalidator threads increases. This suggests that when the CPU becomes a bottleneck, we can reduce CPU utilization by limiting the number of monitoring rules. In addition, if needed, we can control the kernel flow table size by limiting the number of monitoring rules.

4.3 Impact of anomaly detection module

Finally, we examine CPU utilization of the anomaly detection module. We implement the port scan detection function in UMON for this experiment (see Fig. 2). Assume host A is connected to OVS. To detect port
scans against A, the detection module needs to see all
SYN packets sent to different ports of host A. This can
be done by inserting the monitoring rule of DstIP=A
with Subflow Monitoring Action and SYN Monitor-
ing Action on. The subflow mask is set to be all ones.
We use the trace from MAWILab [8] that contains port
scan traffic. We extract attack traffic toward 50 hosts
and mix it with the previous trace.

![Figure 6: UMON vSwitch CPU utilization profiling when under attack](image)

Figure 6: UMON vSwitch CPU utilization profiling when under attack

The port scan detector manages to find out all the
port scan attacks. Figure 6 shows the CPU utilizations
for the four revalidator threads and for the port scan
detector thread. The CPU utilizations for the handler
threads are close to zero and are not shown. Since the
mixed trace contains a large number of flows, and the
UMON vSwitch monitors all hosts using the Subflow
Monitoring Action, the CPU utilization of individual
revalidator thread reaches 45% of a core. The CPU
utilization for the port scan detector is low, though.
It only consumes about 0.21%. Statistics collection in
UMON consumes the most CPU resources.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we propose UMON, a user-defined flex-
ible traffic monitoring framework that decouples mon-
itoring from forwarding. UMON provides efficient and
flexible monitoring of user-definable flows. We describe
UMON’s design and its implementation in OVS. We show
from the evaluation that monitoring overhead due to
this flexibility is acceptable. We are investigating the
implementation of UMON in a DPDK based OVS for
further performance gains. We are also investigating
applications which utilize a distributed UMON moni-
toring network.

6. ACKNOWLEDGMENTS

This work is partially supported by National Science
Foundation (NSF) under grants CNS-1117300 and CNS-
1524462 and ARO under W911NF-15-1-0262.

7. REFERENCES