vPROM: vSwitch Enhanced Programmable Measurement in SDN

An Wang*, Yang Guo†, Songqing Chen*, Fang Hao‡, T.V. Lakshman‡, Doug Montgomery†, Kotikalapudi Sriram†
* George Mason University ,† NIST ,‡ Bell Labs, Nokia

***Abstract*—While being critical to the network management, the current state of the art in network measurement is inadequate, providing surprisingly little visibility into detailed network behaviors and often requiring high level of manual intervention to operate. Such a practice becomes increasingly ineffective as the networks grow both in size and complexity. In this paper, we propose vPROM, a vSwitch enhanced SDN programmable measurement framework that automates the measurement process, minimizes the measurement resource usage, and addresses several significant technical challenges faced by early works. vPROM leverages the SDN programmability and extends the Pyretic runtime system and OpenFlow network interface to achieve the measurement automation. The required measurement resources are minimized by only acquiring the necessary statistics, made possible with instrumented Open vSwitches [1] with user defined monitoring capability. By decoupling monitoring from routing, vPROM reduces the interference between the measurement applications and other applications, and eliminates the frequent involvement of the controller. A vPROM prototype is implemented with DDoS and port-scan detection applications. The performance of vPROM is evaluated and the comparison results with other existing programmable measurement approaches are also presented.**

I. Introduction

SDN is an emerging networking paradigm that enables the programming of the underlying network. Network measurement and monitoring is an important network application that can take advantage of the SDN's programmability. The SDN programmable measurement automates the measurement process, minimizes the resource usage by acquiring only the necessary statistics, and is able to utilize SDN switches as the measurement points across the networks. The SDN programmable measurement measures network traffic by actively installing rules for the flows of interest in the SDN routers' forwarding tables. The flow stats, such as packet and byte counts of the flows of interest, are collected through the flow entry counts. The measurement is controlled by the traffic measurement application programmed using network programming languages, and can be dynamically adjusted based on measurement needs. The initial endeavor on the SDN based programmable measurement has shown promises. In [1], network measurement policies are provided that allow users to query the network and conduct the measurement function such as sub-flow monitoring. NetAssay [2] pursues the so-called intentional network monitoring to capture the minimal set of traffic that satisfies the operator's monitoring goal.

While promising, the current SDN programmable measurement faces significant technical challenges: *(1) The interference between monitoring and other applications, e.g., forwarding, is nontrivial.* Each application has its own goal and a set of policies to enforce. Flow rules installed/removed by one application often interfere with overlapping rules installed/removed by other applications [3]. Hence any changes made by any application may require the run-time system to recompile to solve the conflicts. The newly generated forwarding entries then need to be installed into the switches' forwarding tables - resulting in significant overhead on the run-time system, the controller, as well as the SDN switches. In fact, such frequent recompilation negatively affects the system scalability as shown in [4]; *(2) The programmable measurement may require the continuous involvement of the controller.* For instance, define the *subflows* to be the fine-grained flows that belong to a mega-flow. Subflow monitoring requires the switch to send the first packet of every subflow to the central controller since the specific subflows are not known in advance. Such constant controller involvement is undesirable. *(3) Monitoring packet and byte counts by association with flow entries in the forwarding table is neither flexible nor sufficient for supporting various 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 40 or so [5]; *(4) The amount of Ternary Content-Addressable Memory (TCAM) at hardware switches is limited.* TCAM, widely used for fast packet forwarding, is expensive and power hungry, which limits its amount inside a physical switch. The available TCAM may not be sufficient for the measurement purpose;

In this paper, we propose to build vPROM, a vSwitch enhanced SDN programmable measurement framework that addresses the aforementioned issues. vPROM runs on the instrumented Open vSwitches [6], [7] that decouples monitoring from forwarding and can support user-defined monitoring capability. Furthermore, we extend Pyretic to Pyretic+ run-

[1]Certain commercial equipment, instruments, or materials are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

978-1-5090-6501-1/17/$31.00 © 2017 IEEE

time system to parse vPROM applications into flow rule sets for both forwarding and monitoring, and extend OpenFlow to OpenFlow+ in order to allow applications to set up monitoring rules. A client is also built to facilitate the communication between the controller and the run-time system.

A salient feature of vRPOM is the extensive use of Open vSwitches (OVS) as measurement vantage points. OVS often 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. 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 migrated from the central administration point to the edge. The instrumented Open vSwitch, called UMON [7], supports the explicit measurement function that decouples the measurement function from the packet forwarding function. The decoupling is achieved via the introduction of the *monitoring flow table*, which separates the monitoring rules from the forwarding rules. Users can thus freely install monitoring rules without worrying about the possible interference with the forwarding rules.

vPROM extends the Pyretic run-time system to Pyretic+ so that a vPROM measurement application programmed in Pyretic can seamlessly utilize the UMON capabilities. The run-time system is modified to automatically identify the measurement capability of a SDN switch, and use the monitoring flow table if the switch is instrumented. A Ryu SDN controller is used in vPROM. A Ryu client is built so that the Pyretic+ run-time system can communicate with the Ryu controller to configure SDN switches and retrieve states from SDN switches. To demonstrate the capability of vPROM, we implemented a prototype and several vPROM applications. The performance of vPROM is evaluated and the comparison results with other existing programmable measurement approaches are also presented.

The paper is organized as follows. Related work is summarized in Section II. The vPROM architecture are described in Section III. A vPROM application is presented in Section IV. Evaluation results are presented in Section V. Concluding remarks are in Section VI.

II. Related work

Network programmability has been studied extensively and several network programming languages, e.g., [8], [1], [9], [10], among others, have been developed. The programming languages offer high level abstractions that make the programming of complex network functions/applications [11], [12], [13] possible. Sophisticated SDN applications can be programmed and run simultaneously without worrying about the intricate interactions among them. The study in [4], however, shows that it may take minutes to compile policies of different applications and generate millions of forwarding rules that need to be installed in the data plane for a realistic large Internet exchange point. In vPROM, measurement points, the vSwitches, are instrumented with the explicit measurement function. The network measurement function is thus decoupled from other functions, eliminating the interactions.

Network measurement has been programmed as SDN applications or query policies [1], [12], [2]. These network measurement applications run on top of the run-time system and the controller and often require repeated involvement of both elements, e.g., when conducting sub-flow monitoring. vPROM addresses this issue by decoupling the monitoring from the forwarding. We further extend the OpenFlow API to allow the measurement applications to directly control measuring switches through the controller. Trumpet [14] takes a different approach and designs its own distributed packet monitors and centralized event monitoring system. Trumpet packet monitors collect stats associated with pre-defined 5-tuple flows. vPROM favors customized monitoring that can dynamically change monitoring resolutions demanded by users/applications. In addition, vPROM leverages the existing open source software and latest research advancement on network programming languages.

III. vPROM design

Fig. 1 depicts the architecture of vPROM. As shown in the figure, vPROM consists of five major components: (1) UMON vSwitches, the instrumented Open vSwitches that provide user-defined monitoring capability and some local application functions being pushed from the central controller to the edge; (2) OpenFlow+, the augmented OpenFlow API that allows the applications to set the monitoring rules at UMON and to control the application threads running at the vSwitches; (3) the Ryu client, which serves as the "interpreter" between the run-time system and the Ryu controller; (4) Pyretic+, the extended Pyretic run-time system that can parse a vPROM app into the flow rule sets for traditional SDN switches and the monitoring rule sets for UMON vSwitches; and (5) vPROM applications programmed using extended Pyretic language. vPROM applications obtain measurement stats from both traditional SDN switches and UMON switches. Below we present the Pyretic+ and OpenFlow+, after an overview of UMON, the instrumented OVS.

A. Background on UMON

The UMON design [7] strives to achieve three goals: (1) decoupling monitoring from forwarding; (2) supporting subflow monitoring and monitoring based on non-routing fields; and (3) supporting application threads. To achieve these goals, the major challenge lies in how to implement the decoupling in the existing vSwitch architecture. The packet forwarding pipeline is defined in the Openflow specification [15] and implemented in the Open vSwitch's user space (see top part of Fig. 2). In UMON, a new table, *monitoring flow table*, is designed and implemented to separate monitoring rules from forwarding rules, as shown in Fig. 2. Users can thus freely install monitoring rules without worrying about the possible interferences with forwarding rules. The subflow monitoring is also supported by a newly defined *subflow monitoring action*, which acts as a local controller. The subflows subjected to

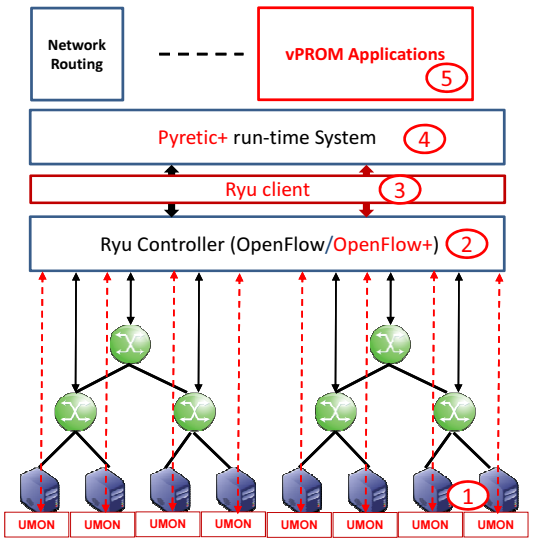


Fig. 1. vPROM framework architecture and key elements.

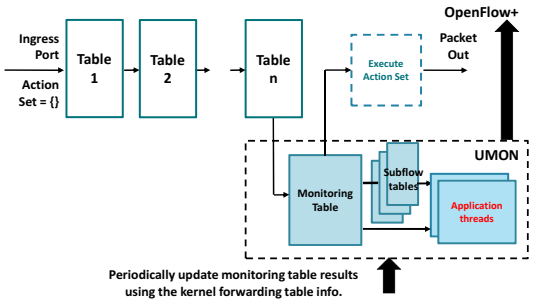


Fig. 2. Packet forwarding pipeline in the UMON, an instrumented Open vSwitch.

the monitoring are inserted into the subflow table where the monitoring results are gathered and stored. The measurement results can be actively collected by vPROM applications through the OpenFlow+ API. Application threads, such as the port-scan detection threads and DDoS attack detection threads, run in UMON using the locally collected stats. These application threads, running at UMONs distributed across the network, scale up the central vPROM application, and reduce the measurement traffic from the switches to the central controller.

The architecture of the Open vSwitch is more complex than the pipeline as depicted in Fig. 2. It includes a kernel module which caches the flow rules to speed up the packet forwarding. In Section IV-A, we instrumented UMON to support quick large flow detection using coincidence counting scheme [16]. We address the challenge of dividing the tasks between kernel module and user-space modules. Finally, in order to support subflow monitoring and monitoring on non-routing fields, it is infeasible to employ a dedicated flow table in the OpenFlow pipeline to replace the monitoring table.

B. Pyretic+ and its run-time system

Pyretic is a Python style network programming language that offers high-level abstractions for users to write compact programs to define what the network switches should do with incoming packets. Pyretic has a corresponding run-time system that takes multiple Pyretic programs as input, compiles them together and generates flow rule sets to be installed at the underlying SDN switches. These flow rule sets satisfy the collective Pyretic programs' requirements. We call the extended Pyretic Pyretic+. Below we first describe how the Pyretic+ language supports UMON switches semantically. We then describe how the Pyretic+ run-time system generates the forwarding rules and monitoring rules separately.

1) Pyretic+ language: Pyretic defines polices and operators [17]. The basic polices includes `match`, `drop`, `identity`, `forward`, `flood`, `if_`, etc., and the operators include `+` (parallel composition), `>>` (sequential composition), etc. Pyretic further defines three query polices:

- `packets(limit=n,group_by=[f1,f2,...])`, which callbacks on every packet received for up to `n` packets identical on fields `f1,f2,...`;
- `count_packets(interval=t,group_by=[f1,f2,...])`, which counts every packet received. Callback every `t` seconds to provide count for each group;
- `count_bytes(interval=t,group_by=[f1,f2,...])`, which counts every byte received. Callback every `t` seconds to provide count for each group.

For instances, in the following example, all TCP traffic incoming from `inport=1` are sub-flow monitored based on their `'srcip'` and `'dstip'`. The traffic is then forwarded to `outport=2`.

```
Q = count_packets(interval=t, group_by=['srcip','dstip'])
match(inport=1) >> if_(match(protocol=6), Q, identity) >> fwd(2)
```

To support UMON TCP flagged packets monitoring, Pyretic+ adds the `'tcpflag'` option in the query policies' `group_by` parameter. Using `'tcpflag'` option alone, namely `group_by=['tcpflag']` indicates that the action *OFPAT_MONITOR* as defined in OpenFlow+ is active. In contrast, if the `'tcpflag'` option is used along with other options such as `srcip` and `dstip`, the subflow monitoring will be executed.

New policies are introduced to control individual edge management threads. For instance, the new policy `prtscan_detection` can activate/deactivate local port-scan detector. The parameter options are defined the same as in the action *OFPAT_PRTSCAN_DETECTION* in OpenFlow+. The callback function can also be defined and registered to react to the received alert messages.

2) Pyretic+ run-time system: The Pyretic run-time system compiles the programs and generates an abstract syntax tree (AST) that represents the policies and their inter-relationship as defined by the operators. For example, the abstract syntax tree (AST) as shown in Fig. 3 is derived from the `count_packets` example in Section III-B1. In this figure, all the operator nodes are marked in green and the polices are in yellow. The tree is built by parsing the application programs.

The run-time system then generate the flow rule sets for individual SDN switches based on this AST.

In Pyretic+, the run-time system needs to generate both the forwarding rules and monitoring rules for a UMON switch. This is achieved by deriving separate forwarding AST and monitoring AST using the general AST as in Pyretic run-time system. The forwarding rules and monitoring rules are created thereafter.

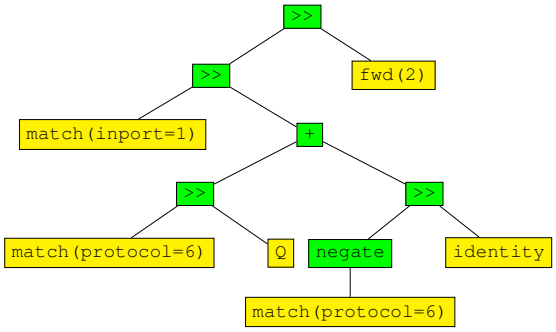
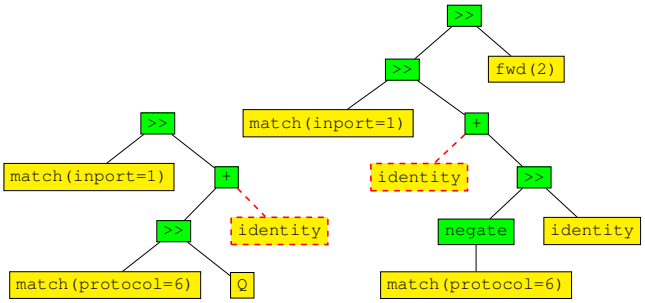


Fig. 3. Abstract syntax tree (AST) of a measurement application example

• **Deriving monitoring AST**. The Algorithm MON-AST-GEN describes how to generate monitoring AST and flow rules. The algorithm starts with finding the query policy nodes and UMON specific policy nodes as defined by the set $\mathbb{C}$. For each identified such node, e.g., policy `Q` in Fig. 3, the while-loop between line 9 and 15 collects all the operator nodes from the identified node up to the top-left node. The nodes posterior to the identified node are ignored since they have no effect on the monitoring policy. The nodes are further processed to remove the nodes operated in parallel with the identified nodes, as shown between line 14 and 18 in the algorithm. As a result, the sub-trees of any of the operators `intersection`, `sequential` and `difference` are preserved to build monitoring policy. The generated monitoring AST is shown in Fig. 4(a).



(a) Monitoring AST (b) Forwarding AST

Fig. 4. Derived forwarding and monitoring ASTs

The function BUILDPOLICY further compiles the monitoring AST into policy, i.e., flow rules, similar to a stack machine compiler. A stack machine uses a last-in, first-out(LIFO) stack to hold the temporary values. Most of its instructions assume the operands are popped from the stack and the operation results are pushed back to the stack. In MON-AST-GEN, BUILDPOLICY creates an empty stack and continuously reads

```
1: function MON-AST-GEN(rt_ast)
2:     init policies = LIST()
3:     ℂ=SET([count_packets, count_bytes,
   counts, packets, prtscan_detection])
4:     ℚ=SET([intersection, sequential,
   difference])
5:     𝕃 ← all leave nodes of rt_ast
6:     for l ∈ 𝕃 do
7:         if ISINSTANCE(TYPEOF(l)) ∈ ℂ then
8:             set r_nds = LIST()
9:             while p_node ≠ the top-left node do
10:                r_nds.append(p_node)
11:                p_node ← p_node.GETPARENT()
12:            end while
13:            set nd_lst = SET()
14:            for op ∈ r_nds do
15:                if ISINSTANCE(TYPEOF(op)) ∈ ℚ then
16:                    nd_lst.add(relevant nodes from sub-
   tree of op)
17:                end if
18:            end for
19:            policies.append(BUILDPOLICY(nd_lst))
20:        end if
21:    end for
22:    return policies
23: end function
```

nodes from *nd_lst*. If it reads an operand, the node will be pushed into the stack. Otherwise, operands will be popped from the stack based on the operation types.

The algorithm is run once for each SDN switch since the policies may be different for different switches. For example, monitoring policy `match(protocol=6)>>if_(match(switch=1,srcip='10.0.0.1'),Q,Q)` will generate different rules on switch 1 and other switches.

• **Deriving forwarding AST**. The gist of deriving forwarding AST is to remove the nodes relevant to the monitoring functions. Notice that the forwarding AST is not complementary to the monitoring AST entirely as shown in Figure 4. The node `match(protocol=6)` is unrelated to the forwarding policy. However, node `match(inport=1)` is shared by both ASTs. As a result, algorithm FORWARD-AST-GEN cannot simply remove all the nodes in monitoring AST. Function FORWARD-AST-GEN starts from the query policy nodes and UMON specific policy nodes in the AST. For each such node, the algorithm iterates upward until it hits the first `parallel` operator node. This process will remove all the nodes that are exclusive to the monitoring AST as identified between line 9 and 15 in the algorithm. Finally, function BUILDPOLICY is called to build forwarding rules/policies based on the nodes in the forwarding AST.

C. OpenFlow+ Protocol

OpenFlow+ extends the OpenFlow protocol to enable the SDN controller to manage the UMON monitoring table, collect the measurement stats, and start/stop the application threads at UMON vSwitches. The OpenFlow protocol contains

```
function FORWARD-AST-GEN(rt_ast)
    init policies = LIST()
    ℂ=SET([count_packets, count_bytes,
counts, packets, prtscan_detection])
    𝕃 ← all leave nodes of rt_ast
    for l ∈ 𝕃 do
        if ISINSTANCE(TYPEOF(l)) ∈ ℂ then
            set nd_lst = SET()
            set r_nds = LIST()
            while p_node ≠ the top-left node do
                if ISINSTANCE(TYPEOF(p_node)) ==
parallel then
                    break
                end if
                r_nds.append(p_node)
                p_node = p_node.GETPARENT()
            end while
            prune subtree of p_node
            nd_lst ← all the relevant nodes
            policies.append(BUILDPOLICY(nd_lst))
        end if
    end for
    return policies
end function
```

three types of messages: *Controller-to-Switch messages*, *Asynchronous messages*, and *Symmetric messages*. The controller-to-switch messages are initiated by the controller and may or may not require a response from the switch. Asynchronous messages are sent by switches to the controller without solicitation. Switches send asynchronous messages to the controllers to signal a packet arrival, change of switch state, or an error. Symmetric messages, such as Hello and Echo, are sent without solicitation in either direction. We next describe the additional messages added in OpenFlow+ and their implementation.

• **Monitoring Table Management**. Each OpenFlow message begins with the OpenFlow header, which includes a `type` field indicating the type of a message. We introduce a new type *OFPT_MONITOR_MOD* to indicate that the message is related to the monitoring table. Five commands, *OFPMMC_ADD*, *OFPMMC_MODIFY*, *OFPMMC_DELETE*, *OFPMMC_MODIFY_STRICT*, and *OFPMMC_DELETE_STRICT*, are similar to the forwarding flow table modification commands. The last one, *OFPMMC_DELETE_SUBFLOWS*, enables the controller to delete the subflow tables to save the storage space.

Besides the new commands, we add two types of new monitor actions: *OFPAT_MONITOR* for monitoring non-routing fields and subflow monitoring, and actions to control application threads. The *OFPAT_MONITOR* action structure is as follows:

```
struct ofp_action_monitor {
    ovs_be16 type;
    ovs_be32 monitor_flag;
    uint8_t subflow_flag;
    struct ofp_match_header subflow;
    ...
};
```

The field *monitor_flag* allows users to define the monitoring of non-routing fields. For instance, *monitor_flag* values of *OFPMT_SYN*, *OFPMT_SYNACK*, *OFPMT_FIN*, etc., instruct to collect packet/byte counts of TCP SYN, SYN/ACK, and FIN. The two parameters, *subflow_flag* and *subflow_mask*, are for subflow monitoring purpose. The first parameter is a boolean value indicating if subflow monitoring is turned on. If it is on, *struct ofp_match_header subflow* contains the wildcard mask for subflow monitoring. The action for application thread control is described later.

• **Stats collection**. The stats request from the controller to the switch is a new multipart message defined as *OFPMP_MONITOR_STATS*. This stats request allows the controller to collect the stats of the entire monitoring table, or the stats of a specific monitoring rule. The subflow tables associated with the monitoring rules can also be reported when available. We use the following data structure for *OFPMP_MONITOR_STATS*:

```
struct ofp_monitor_stats_request {
    uint8_t type;
    uint8_t with_subflows;
    uint8_t threshold_type;
    ovs_be32 threshold_value;
    /* Followed by an ofp_monitor_match
       structure for exact match rule request. */
    ...
};
```

We define two new *types*: *OFPMR_ALL* and *OFPMR_EXACT*. *OFPMR_ALL* requests the stats of the entire monitoring table, while *OFPMR_EXACT* requests the stats of a specific rule or rules matching the *ofp_monitor_match* field. The field *with_subflows* indicates if the subflow tables should be reported. If *with_subflows* is on, we also control the granularity at which the subflow tables are reported. The field *threshold_value* allows to set up a threshold and only the subflow entries whose byte count or packet count surpasses the threshold will be reported to the controller. The field *threshold_type* defines whether byte count (*OFPMRT_BYTE*) or the packet count (*OFPMRT_PKT*) is chosen in the threshold comparison. After receiving the stats request, the switch generates a reply message including information concerning the matching monitor rules, the related statistics, and the subflows, if any.

• **Application thread management**. For each application thread, we introduce an action to control this thread. Application threads are implemented as UMON threads that use the measurement stats for various purposes. For instance, we implement the vertical port-scan detection thread, the horizontal port-scan detection thread, and quick large flow detection thread (see Section IV-A). Using the port-scan thread as an example, we introduce the action *OFPAT_PRTSCAN_DETECTION* for its control. The action structure is defined as follows:

```
struct ofp_action_prtscan_detection {
   ovs_be16 type;
   uint8_t detector_switch;
   uint8_t detection_type;
   ovs_be64 interval;
   ovs_be16 vthresh;
   ovs_be16 hthresh;
   struct ofp_match_header submatch;
   ...
};
```

The parameter *detector_switch* is the knob to enable or disable the local detection thread. The detection is achieved by periodic analysis of the subflow stats. The parameter `interval` defines the period at which the port-scan detector runs to analyze the subflow stats. Moreover, we enable two types of scanning behavior detection, i.e., vertical scan and horizontal scan. The parameter `detection_type` dictates which scan is running. For the purpose of detection, this action accepts threshold for each type. Parameters `vthresh` and `hthresh` are thresholds used by vertical and horizontal detection, respectively. During local port-scan detection, whenever suspicious activities are detected by the application thread, we use the Asynchronous message for the application thread to send alert messages to the controller.

All the new commands introduced in OpenFlow+ are compatible with the early versions of OpenFlow. Extra data structures are necessary on both the controller and the switch to support the implementation of OpenFlow+.

D. Ryu Client

The controller client serves as an interface for the run-time system to communicate with the SDN controller. Its main function is to translate the Pyretic messages (of the run-time system) to the OpenFlow messages (used by the SDN controller), and vice versa. We choose to use the Ryu controller in the vPROM framework over the POX controller used by the original Pyretic run-time system. Ryu is a long-term supported project. The Ryu controller continuously upgrades itself to support newer versions of OpenFlow releases, which will allow vPROM to support newer version OpenFlow in the future with minor change to the controller client. In addition, the implementation of OpenFlow+ in Ryu is quite manageable.

In the Ryu client, an OpenFlow+ interface conducts the message translation. Furthermore, the Ryu client allows the Ryu controller to inform the run-time system if a SDN switch is instrumented, i.e., if a switch is a UMON switch, and if so, what edge management threads it supports. Such information will be stored in the run-time system and be used in meeting vPROM app requirement.

The Ryu client also provides the stats collection service for run-time system. The stats collected in UMON switches can be pulled by the Ryu controller. A stats collection module in the Ryu client periodically instructs the Ryu controller to pull the stats. The collected stats are then forwarded to the run-time system and the vPROM applications.

IV. vPROM-GUARD: A vPROM USE CASE

To demonstrate vPROM's effectiveness, we build vPROM-GUARD, a vPROM application that detects DDoS and port-scan attacks automatically. Distributed Denial of Service (DDoS) attacks and port-scan attacks are significant threats to the Internet. The challenge in DDoS and port-scan defense is the ability to detect patterns of abusive behaviors amongst a vast sea of benign individual network exchanges. Security monitoring systems often utilize the signature-based and/or the behavior-based approach to detect DDoS attacks. Fine grained packet-level or microflow-level measurement at line rate is often required. Such fine grained real-time measurement is extremely demanding on the hardware and requires sophisticated technologies, resulting in expensive network security middle-boxes.

In contrast, vPROM is a distributed measurement framework that can be programmed and reconfigured in real time to respond to ever changing attack vectors. The key idea of vPROM-GUARD is to *employ efficient attack detectors and monitor the attack cues at a coarse measurement granularity when the network is not under attack, and switch to the fine-grained network monitoring and attack detection/validation when suspicious activities are detected.* The benefits of such an approach are multifold: (1) the distributed edge measurement and coarse grained measurement level reduce the overall measurement burden on the network; (2) when under attack, only the alerted hosts need to conduct fine granularity measurement and local detection; (3) local detection at edge mitigates the burden of the central detector; and (4) false alarms are more tolerable because the detection is controlled by a program and a false alarm merely triggers the extra fine-grained measurement at vSwitches rather than frequent human interventions. If proven to be effective, vPROM-GUARD has the potential to replace the middle-box solution in a data center with a pure low cost software solution. Next, we present the detection methods used in vRPOM-GUARD.

A. Coincidence counting based large flow detection

Quick detection of large flows at the incipient of DDoS attack is vital for DDoS detection. The authors in [16] developed the Coincidence Base Traffic Estimator (CATE) that can estimate flow rates quickly with provable bounds on estimation error. CATE maintains a predecessor table and a coincidence count table, as shown in Fig. 5. The predecessor table includes the most recently received k packet headers. A flow is defined as $f = r\&m$ with r being the packet header and m the flow mask. Upon the arrival of a new packet, its corresponding flow id f is compared with every flow id in the predecessor table. The number of coincidences for the flow f, l_f, is the number of times the flow f occurs in the predecessor table. If $l_f > 0$ and flow f is not in the Coincidence count table, f is added into the coincidence count table with count of l_f. If f is already in the coincidence count table, then the count for f is incremented by l_f. Let $M(N, f)$ be the number of coincidences for flow f after N arrivals with k comparisons for each arrival. The estimated proportion of traffic from flow

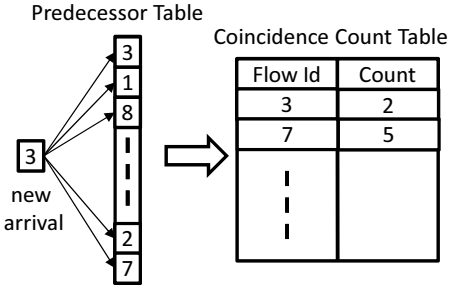


Fig. 5. CATE scheme.

f, $\hat{p}_f$, is $\hat{p}_f = \sqrt{\frac{M(N,f)}{Nk}}$.

While the CATE scheme is reasonably simple, instrumenting UMON to support CATE is not trivial. The coincidence counting is conducted for every new arrival. If CATE is implemented in the kernel module of OVS, it can slow down the data path forwarding speed. We adopt a strategy that offloads the CATE from the critical data forwarding path. Specifically, we implement the CATE scheme as a user-level thread in the OVS. A copy of the incoming packet header is made in the kernel module, and a batch of packet headers is periodically delivered to the user-level CATE. User-level CATE executes the coincidence counting upon receiving the newly arrived packet headers. The strategy minimizes the overhead imposed on the UMON data path.

B. Change-point monitoring for attack cues

The authors in [18] developed the *change-point monitoring* for TCP based attack detection. The technique is based on the observation that TCP {SYN, SYN/ACK} and {SYN, FIN} are *request-response pairs* that should be balanced in a normal network environment, and they deviate from the balanced state when under attacks. The Cumulative Sum Method [19], [20] is employed to detect the deviation. Specifically, let q_i and p_i be the number of requests and responses, respectively, in the i-th measurement epoch. The difference, δ_i, is $\delta_i \triangleq q_i - p_i$. Define $\tilde{\delta}_i$ to be the normalized difference,

$$\tilde{\delta}_i = \delta_i / P_i, \tag{1}$$

with $P_i = \alpha P_{i-1} + (1-\alpha)p_i$ and α being a positive constant less than one. To detect the deviation of $\tilde{\delta}_i$ from its mean, which should be close to zero, the *Cumulative Sum* method is used. Define S_i to be the cumulative sum:

$$S_i = (S_{i-1} + \tilde{\delta}_i - t)^+, \tag{2}$$

where t is a constant threshold and $(\cdot)^+$ takes the positive value or zero. The value of t is chosen such that $\tilde{\delta}_i > t$ indicates a potential attack. When the cumulative sum S_i becomes greater than the threshold T, $S_i > T$, a potential TCP based attack is detected. The value of t and T are design parameters that affect the *attack detectability*, *false alarm interval* and *detection delay*.

In order for the change-point monitoring to detect an attack, $\tilde{\delta}_i$ needs to be greater than t, $\tilde{\delta}_i > t$ when the attack is on. Otherwise $\tilde{\delta}_i - t$ is negative in Eqn (2), and does not contribute to the cumulative sum S_i. Therefore the average number of on-going TCP connections, i.e., the value of P_i, needs to be *comparable* to that of δ_i (see Eqn (1)). Otherwise the attack becomes either not detectable, or the variation in the normal TCP connections is greater than the value of t, which leads to a large number of false alarms (see Eqn (2)). For example, if the goal is to detect if one machine inside a large organization is under attack, conducting the change-point monitoring at the gateway for the entire organization likely does not work since the number of on-going TCP sessions is much larger than the attacking sessions. vPROM-GUARD addresses the issue by conducting the monitoring at individual machines hosting a small number of VMs.

C. Attack detection in vPROM-GUARD

The attack detection in vPROM-GUARD is accomplished in two phases: big flow and coarse-grained indicator/cue monitoring and fine-grained attack detection/validation. In the first phase, vPROM-GUARD periodically detects big flows (via CATE), and collects packet counts of TCP SYN, SYN-ACK, FIN, and RST and runs Cumulative Sum (CUSUM) algorithm. Assume that there are J hosts in total. The change-point monitoring at host j is:

$$\delta_i^j = q_i^j - p_i^j, \tag{3}$$

$$\tilde{\delta}_i^j = \delta_i^j / P_i^j, \tag{4}$$

$$S_i^j = (S_{i-1}^j + \tilde{\delta}_i^j - t)^+, \tag{5}$$

for $j = 1, 2, \ldots, J$. Comparing to the centralized change-point monitoring [18], the distributed change-point monitoring at individual hosts shortens the detection delay and localizes the attacks. For instance, if only host j is under SYN flood attack, then $\delta_i^j = \delta_i$ but $P_i^j \leq P_i$. Thus $\tilde{\delta}_i^j \geq \tilde{\delta}_i$ and $S_i^j \geq S_i$, leading to early detection. Furthermore, the distributed change-point monitoring localizes the detection. Only the hosts whose cumulative sum S_i^j is greater than threshold T need to be further examined.

vPROM-GUARD starts the detection process by turning on CATE monitoring, and installing monitoring rules for TCP SYN, SYN/ACK, FIN, and RST packet counts into UMON at each hosting machine. Then the big flow info and the packets counts are collected periodically by vPROM-GUARD using OpenFlow+ stats collection commands. The change-point monitoring is conducted for individual hosts using the collected stats. If a big flow is detected and the change-point monitoring detects the deviation, a TCP SYN flood attack is likely to be detected. We further install rules to collect TCP flag packet counts associate with this big flow to validate the type of attack. If a big flow is detected but the TCP flag packet counts do not deviate from the balance, it still indicates a potential DDoS attack. The vPROM-GUARD controller can implement whatever policy the users prefer to further classify this flow. Finally, if no big blow is detected but the change-point monitoring issues potential attack alerts to/from a host, the vPROM-GUARD starts the subflow monitoring and port-scan detection threads on that host. The vPROM-GUARD,

running at a central location, also periodically collects the subflow stats from the hosts under alert, and runs DDoS detection and port-scan detection across the subflow stats collected from these suspected hosts. This allows the detection of attacks that may spread across multiple hosts.

V. Evaluation

We instrument the Open vSwitch (version 2.3.2) and run it on a four-core, 3.2GHz CPU machine with 10GB memory. The machine is equipped with an Intel NIC with two 10GHz ports. The Ryu controller (version 3.25), Ryu client, Pyretic+, and vPROM apps run on another machine of the same configuration. Both machines use the Ubuntu 14.04.3 LTS kernel. We use a third machine as both the packet generator and the packet sink so as to avoid clock synchronization problem. The packet generator and sink are connected to the vSwitch via two 10GHz ports. The packet generator uses Tcpreplay [21] to replay a data center traffic trace collected by Benson et al. [22]. The trace lasts for a period of about 65 minutes.

A. Comparison of subflow monitoring: vPROM vs Pyretic

Subflow monitoring is an important monitoring capability for applications such as heavy-hitter flow detection and port scanning attack detection. Subflow monitoring is supported in Pyretic by so-called *query policies* [17], which can be conjoined to any of the other policies, e.g., routing policies. It is straightforward to program for the subflow monitoring in Pyretic:

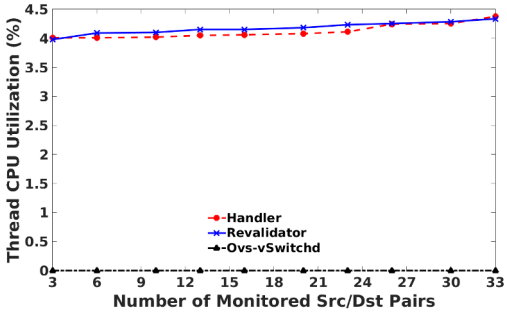
```
m=['srcmac','dstmac','srcport','dstport']
Q=count_packets(interval=t,group_by=m)
match(srcip=A, dstip=B) >> Q
```

The first line defines the subflow mask that is based on source MAC address, destination MAC address, source port Id, and destination port Id. The function *count_packets()* returns the packet counts every t seconds for each subflow. The megaflow is defined using *match()*. *match(srcip=A, dstip=B)* captures all packets from A to B and hands them to subflow monitoring policy Q.

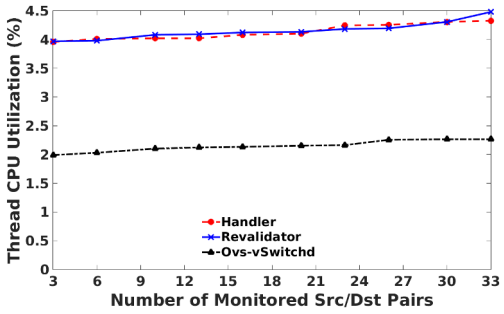
Below we compare the performance of vPROM and Pyretic in supporting subflow monitoring. In the Pyretic experiment, we employ a simple routing that forwards all packets from the input port *in_port* to the output port *out_port* that connects to the sink. The Pyretic routing policy is:

```
match(inport=in_port) >> fwd(out_port)
```

This routing policy runs in parallel with the subflow monitoring policy. The same routing and subflow monitoring are conducted using UMON in vPROM. In addition, since the subflow monitoring is a built-in capability of UMON, vRPOM can directly insert the monitoring rules for each monitored source-destination pair into the monitoring table with the subflow mask of srcmac, dstmac, srcport, and dstport on. We first evaluate the subflow monitoring overhead imposed on the switches, an UMON vSwtich in vPROM and an non instrumented Open vSwitch in Pyretic. Overhead is measured using the CPU utilization of three types of threads: *handler*, *revalidator*, and *ovs-vswitchd* [6], [7]. *ovs-vswitchd* is a user space daemon that handles the communication with the SDN controller, among other things. Figure 6 depicts the CPU utilization of different threads for vPROM and Pyretic. We vary the number of monitored source-destination pairs, from three to thirty-three, to change the monitoring workload. In all cases, the CPU utilization of all threads increases with the number of monitored pairs. The CPU utilization of threads *handler* and *revalidator* is similar for vPROM and Pyretic, but differs for *ovs-vswitchd* thread. For vPROM, no CPU resources are consumed by thread *ovs-vswitchd* since all packet processing decisions are made locally and there is no need to communicate with the controller. In contrast, the subflow monitoring in Pyretic requires the visibility of every matching subflows. Thread *ovs-vswitchd* needs to forward the matching packets to the controller, resulting in CPU consumption.



(a) vPROM



(b) Pyretic

Fig. 6. Overhead in Data Plane

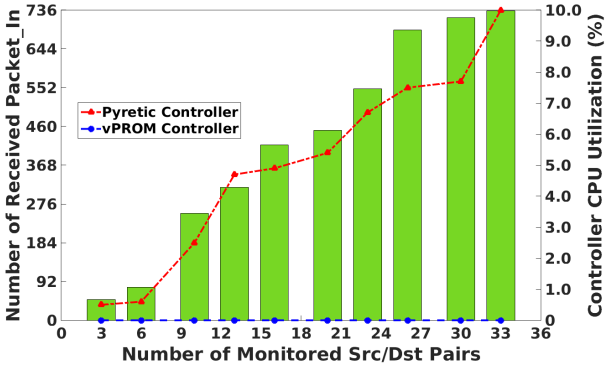


Fig. 7. Overhead in Control Plane

Next we measure the control plane overhead. Figure 7 depicts the number of Packet_In messages received at the controller (curves with the left Y-axis) and CPU utilization of the controller (bars with the right Y-axis) against the number

of monitored src/dst pairs. Since UMON has no interactions with the controller, the CPU utilization and Packet_In count remain at zero throughout the experiment. For Pyretic, the number of Packet_In messages increases when more pairs are monitored. The CPU utilization of the controller also increases proportionally to the number of received Packet_In messages.

To further compare the scalability of both solutions, we increase the number of monitored pairs to stress both the vSwitch and the controller. The test results are shown in Figure 8. In this figure, we present two sets of results. First,

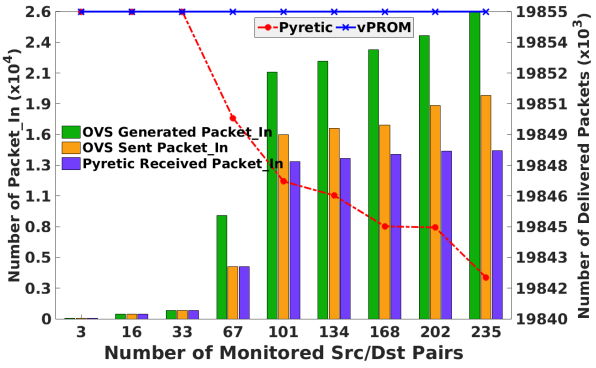


Fig. 8. Stress Test Results

we plot the number of delivered data packets by vPROM and Pyretic (curves with Y-axis on the right). The trace used for the evaluation contains 19,855,388 packets in total. The curves of delivered packets show that vPROM is able to deliver all packets as more and more pairs are monitored, while Pyretic starts to suffer from the packet losses when the number of monitored pairs is greater than 67 pairs.

We investigate the cause of the data path packet losses in Pyrectic. For that we examine the Pyretic's control plane overhead. We plot the number of Packet_In messages generated by Open vSwitch, sent by Open vSwitch, and received by the Pyretic (see bar charts with Y-axis on the left in Figure 8). We observe that Pyretic starts to lose Packet_In messages at both vSwitch and Pyretic (which runs on top of the SDN controller) when the number of monitored pairs surpasses 67 pairs. The difference between the number of OVS generated Packet_In messages and the number of Packet_In messages being sent out indicates the packet loss inside the vSwitch, which is due to the overflow of the Packet_In queue inside the vSwitch. Meanwhile, the difference between the number of sent-out Packet_In and the number of Packet_In received by the Pyretic application indicates the packet loss at the controller. The controller employs the event queue for dispatching various events, such as Packet_In event, to the applications. The losses are due to the overflow of event queues maintained by Pyretic [1]. The results show that the frequent communications between the vSwitch and the controller greatly degrade the performance of both the vSwitch and the controller. Due to the Packet_In message loss, the Pyretic monitor application can not offer accurate subflow monitoring results. vPROM addresses the problem by instrumenting the vSwitch and localizing the subflow monitoring task.

B. *vPROM-GUARD attack detection*

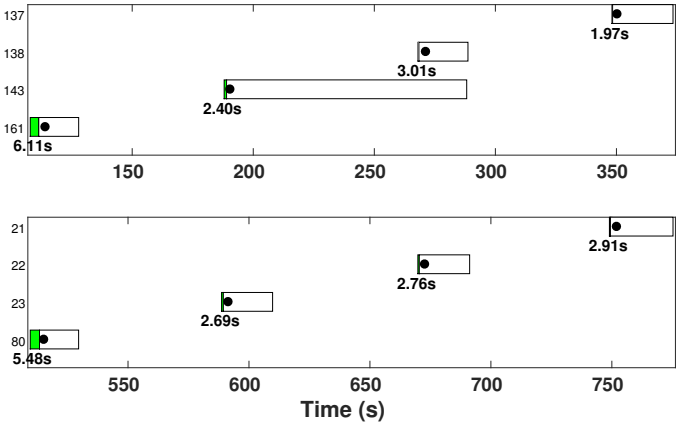
We use two data traces containing verified attacks to evaluate the effectiveness of vPROM-GUARD. For the SYN Flood attack, we use *Endpoint Traffic* collected from three different deployment points in NUST SEECS labs [23]. In this data-set, eight ports on two hosts are under known SYN Flood attacks. The attacking rate varies from 10 pkts/second to 1000 pkts/second and the average background traffic rate varies between 200 to 650 pkts/second. There are in total 2325 hosts in this data trace. For the port scanning attack, we use the trace collected by Mawilab [24]. In this data-set, one horizontal scanning attack and three vertical scanning attacks are known. We use Emulab in our lab to conduct the experiments. vRPOM-GUARD runs on the vPROM framework at one machine, and two UMON switches and a CATE capable switch controlled by vPROM-GUARD are running on another machine. The CATE capable switch emulates the gateway switch; While the two UMON machines emulate the vSwitches at two host machines in a data center, each hosting about 20 IPs with some of IPs being under attack. We set the polling intervals for all the detection to be one second, and t to be 0.4 and T to be 1. For the CATE scheme, we set the threshold for the large flow detection at 0.05, i.e., a flow is deemed to be large if it is more than 5% of overall traffic rate. The detected big flows are reported to vPROM-GUARD every one second.

vPROM-GUARD manages to detect all attacks in the data traces. Fig. 9(a) shows the eight SYN flood attack detection time. Attacks target the two hosts, with IP address of $87.51.34.132$ (top of Fig. 9(a)) and $69.63.178.11$ (bottom of Fig. 9(a)) with different port ids, as shown in the Y-axis. The horizontal bar indicates the starting and finishing time of the attack, with the vertical line indicating the moment at which the CATE issues a big-flow warning. Once a big flow is detected, a monitoring rule collecting TCP flag packet counts is installed to validate if the large flow is a SYN flood attack. The dot indicates the moment at which the vPROM-GUARD actually validates the attack as SYN flood attack. The lightweight design and implementation of CATE enables vPROM-GUARD to detect such attacks quite efficiently. The average detection time is about 3 seconds, including the attack validation time.

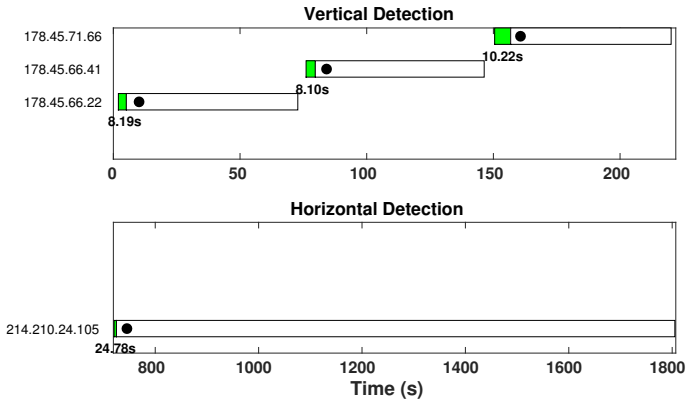
For the port-scan attacks in our trace, they do not generate big enough traffic flows to be detected by CATE. As a result, change-point monitoring and subflow collections are required for the detection. Fig. 9(b) shows the detection time of the vertical and horizontal port-scan attacks. The detection time for vertical port-scan attack is about 10 seconds. The horizontal port scan attack detection takes about 25 seconds. The horizontal port-scan spreads the attack traffic among multiple IPs, hence a smaller attacking rate for one IP and takes longer time to detect.

VI. Conclusions and Future Work

In this paper, we present the design and implementation of vPROM, a vSwitch enhanced programmable measurement framework that allows users to program the network mea-



(a) SYN Flood Attack



(b) Port scanning Attack

Fig. 9. Attack detection in vPROM-GUARD

surement and control applications. vPROM uses instrumented Open vSwitches (UMON) as the measurement points, and augments the OpenFlow API to OpenFlow+ so that the UMONes can be directly controlled by the applications via the SDN controller. In vPROM, we also extend the Pyretic programming language and run-time system to Pyretic+ and build a controller client in order to support and automate the programmable measurement. To demonstrate its usefulness, we also build the vPROM-GUARD, a DDoS and port-scan attack detection application that demonstrates the major features of vPROM. Performance evaluations and comparisons with other approaches show the advantages of vPROM. Moving forward, we are building more vPROM applications and investigating how to use both UMONes and physical SDN switches as the monitoring points simultaneously. In addition, we are studying to employ the behavior based anomaly detection as the coarse granularity monitoring cues.

VII. Acknowledgement

We appreciate constructive comments from anonymous referees. This work is partially supported by a NIST grant 70NANB16H166, an ARO grant W911NF-15-1-0262, and a NSF grant CNS-1524462. Yang Guo wishes to thank Nien-Fan Zhang (NIST) for helpful discussions on Cumulative Sum Chart Method, and thank Kevin Mills (NIST) and Vladimir Marbukh (NIST) for reviewing the paper and offering helpful comments.

References

[1] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," *USENIX ;login*, 2013.

[2] S. Donovan and N. Feamster, "Intentional network monitoring: Finding the needle without capturing the haystack," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, 2014.

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ICFP*, 2011.

[4] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever, "An industrial-scale software defined internet exchange point," in *NSDI*, 2016.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, 2014.

[6] "Open vSwitch," http://openvswitch.org/.

[7] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen, "Umon: Flexible and fine grained traffic monitoring in open vswitch," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, 2015.

[8] N. Foster, M. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, and D. Walker, "Languages for software-defined networks," *IEEE Communications Magazine*, 2013.

[9] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *SIGPLAN Not.*

[10] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *SIGCOMM*, 2016.

[11] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[12] S. Narayana, J. Rexford, and D. Walker, "Compiling path queries in software-defined networks," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, 2014.

[13] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A Software Defined Internet Exchange," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[14] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *SIGCOMM*, 2016.

[15] "OpenFlow Switch Specification," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[16] F. Hao, M. Kodialam, T. Lakshman, and H. Zhang, "Fast, memory-efficient traffic estimation by coincidence counting," in *INFOCOM*, 2005.

[17] "Pyretic Tutorial," https://github.com/frenetic-lang/pyretic/wiki/Query-Policies.

[18] H. Wang, D. Zhang, and K. G. Shin, "Change-point monitoring for the detection of dos attacks," *IEEE Trans. Dependable Secur. Comput.*, 2004.

[19] B. Brodsky and B. Darkhovsky, "Nonparametric methods in change-point problems," *Kluwer Academic Publishers*, 1993.

[20] P. Winkel and N. Zhang, "Statistical development of quality in medicine," *Wiley Publishers*, 2007.

[21] AppNeta, "Tcpreplay," http://tcpreplay.synfin.net/wiki/tcpreplay.

[22] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010.

[23] S. Ali, I. U. Haq, S. Rizvi, N. Rasheed, U. Sarfraz, S. A. Khayam, and F. Mirza, "On mitigating sampling-induced accuracy loss in traffic anomaly detection systems," *ACM SIGCOMM Computer Communication Review*, 2010.

[24] C. Sony, "Traffic data repository at the wide project," in *Proceedings of USENIX 2000 Annual Technical Conference: FREENIX Track*, 2000.