CounterMap: Towards Generic Traffic Statistics Collection and Query in Software Defined Network

Jiamin Liu, Peng Zhang, Huanzhao Wang, and Chengchen Hu
Department of Computer Science and Technology
Xi'an Jiaotong University, Xi'an, China

***Abstract*—Traffic statistics are fundamental for many network measurement tasks like heavy hitter identification, traffic matrix estimator, anomaly detection, etc. However, traditional techniques like NetFlow and sFlow only provide coarse-grained statistics due to packet or flow sampling. Even Software Defined Networking (SDN) offers fine-grained traffic statistics collection, most of existing methods focus on specific applications and thus lack generality. To this end, we propose CounterMap, a generic traffic statistics collection and query platform. CounterMap maintains a full map of flow counters by actively polling switches and passively monitoring flow timeouts. For efficient storage and query, CounterMap stores the counters in fast off-the-shelf in-memory data store, and offers a generic SQL-like query language. With the CounterMap language, applications can gain visibility into both existing and historical flows, without querying the dataplane devices themselves. We show how network applications benefit from CounterMap, with higher measurement accuracy and lower dataplane overhead.**

Keywords—Software Defined Network; traffic statistics; query language

I. INTRODUCTION

Traffic statistics collection is important for various network applications. There is a fundamental tradeoff between overhead and accuracy for traffic statistics collection. Tools like NetFlow [1] and sFlow [2] use packet or flow sampling to reduce collection overhead, while only provide coarse-grained statistics.

The recent advance of Software Defined Network (SDN) provides new opportunities for fine-grained traffic statistics collection. For example, OpenFlow [5] allows applications to actively query switches for per-flow statistics like byte/packet count. In addition, the controller can also passively sense flow arrivals and leaves by monitoring OpenFlow messages sent by switches. Some studies leverage the fine-grained flow statistics collection feature of OpenFlow to achieve lightweight security functions [7-8]. However, we observe that the traffic statistics collection mechanism in OpenFlow has three limitations:

(1) Lack of visibility. Currently, the OpenFlow flow table size is strictly constrained by the TCAM capacity. For example, CENTEC v350 can only hold 2k flow entries. Therefore, switches need to frequently time out flows, whose counters will be permanently lost and become invisible to applications. We will show in Section II that the counters for expired flows are also important for some applications.

(2) Low abstraction level. The traffic statistics collection interface provided by OpenFlow has a low abstraction level [16]. As a result, programmers must tune the spatial and temporal granularity by setting proper wildcard values and querying period, and then filter out the results. This makes the programs complicated and error-prone. Even SDN languages like Frenetic [16] and Pyretic [17] support flow statistics queries at a higher level, they cannot provide statistics of historical flows which are important for traffic analysis and prediction.

(3) High data redundancy. Currently, when multiple applications need traffic statistics simultaneously, they have to query the data plane for counters individually [16]. This will result in redundant counters. Moreover, multiple switches on a flow's path may have the same entries, and querying all these switches will also result in redundant counters. The above cross-application and cross-switch redundancy can impose large overhead on switches and OpenFlow channel. Previous studies [13-14] only consider the cross-switch redundancy, without addressing the cross-application redundancy.

To this end, we propose CounterMap, a generic traffic statistics collection and query platform for SDN applications. CounterMap addresses the above three limitations as follows. First, CounterMap maintains counters for both existing and expired flows, such that applications always have access to statistics of all flows that ever exist during a specific period, and reduces cross-application redundancy as well. Secondly, CounterMap stores the counters in in-memory key-value store, and offers a SQL-like query language for statistics query. Third, CounterMap uses a heuristic algorithm to poll switches, in order to reduce counter redundancy, and at the same time ensuring load balance among switches.

The key challenges faced by us include: (1) how to make the query language expressive enough to support a large spectrum of applications, and to the most extent reduce lines of codes for applications; (2) how to make the request, storage, and retrieval of flow counters efficient enough, so as to scale to a large networks. We will show how CounterMap addresses these challenges in Section III.

In summary, the contribution of this work is three-fold:

- We demonstrate the importance of expired flows for network applications, motivating the need to maintain a full map of counters.
- We design and implement CounterMap, a new platform for flow statistics collection and query in SDN.
- We propose a heuristic switch polling algorithm to reduce counter redundancy among switches, and test it with various datacenter and campus network topologies.

II. MOTIVATION

A. The Importance of Expired Flows

Due to the limited flow table size, flow entry expiration is frequent for SDN switches. Thus, applications can only retrieve

978-1-5386-2704-4/17/$31.00 ©2017 IEEE

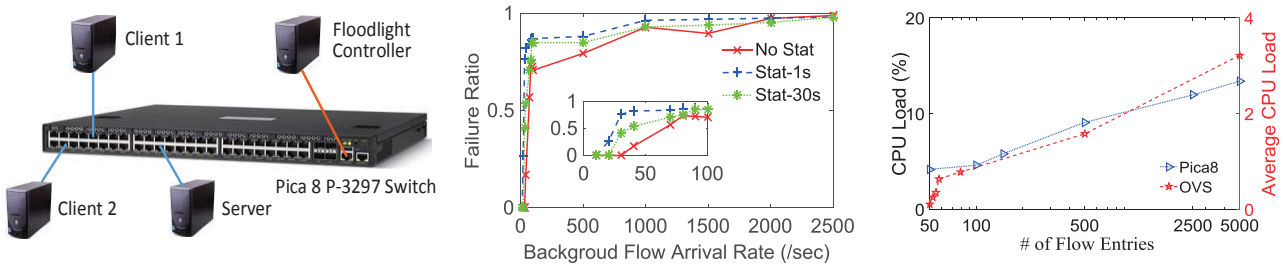

Fig. 1. Experiments showing the counter polling overhead. (a) Topology; (b) The packet loss rate trend under different flow arrival rate and request rate ; (c) the CPU load of pica8 and OVS as the increase of flow entry.

counters for flows that exist in the flow table. However, missing expired flows can result in suboptimal performance for applications. To demonstrate this, we implement a simple entropy-based DDoS detection algorithm, and compare the detection accuracy with and without expired flow entries, respectively.

We use Mininet [6] to emulate a topology with one switch and four hosts, and generate background traffic and DDoS traffic with Scapy. We let the DDoS attack begin at the 30th second, which lasts for 30 seconds. At the same time, we let the controller send a flow_statistics_request message every 2 seconds. After retrieving flow counters, we calculate the entropy for destination IP (Dst_IP) and destination port (Dst_Port). The idle_timeout of flow entries is set to 10 seconds, and we collect expired flow entries by monitoring the flow removal messages sent by switches. The experiment lasts for 2 minutes. Fig. 2 reports the entropy of Dst_IP and Dst_Port when no expired entries, 5 seconds of expired entries, and 10 seconds of expired entries are used, respectively. Note the average entropy value of Dst_Port for the three cases is 1.633, 0.844 and 0.753, respectively. We can see that the detection accuracy increases when more expired flow entries are added, demonstrating the importance of expired flow entries.

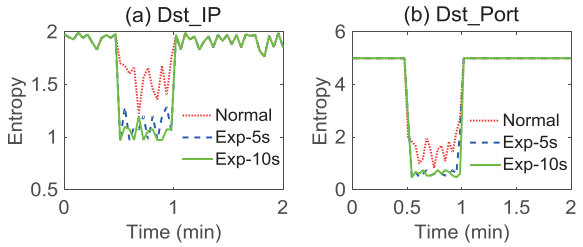

Fig. 2. The entropy for Dst_IP and Dst_Port, when using (1) no expired entries, (2) 5 seconds of expired entries, and (3) 10 seconds of expired entries.

B. The Overhead of Statistics Query

In OpenFlow, when applications request flow statistics, switches should return the counters of all queried flows. This will inevitably impose overhead on switches. In the following, we will use both emulated network and real testbed to numerically study how flow statistics queries affect the performance of switches.

Fig. 1(a) shows the testbed, which consists of a Pica8 P-3297 switch and three hosts [15]. We let client 1 send packets with random source IP addresses to the server, in order to simulate background flow arrivals. At the same time, we let the client 2 send 100 packets with random source IP addresses to the server, and let the server count the number of packets received from the client 2 with tcpdump, denoted by *#Rev*. Then, we vary the background flow arrival rate, and calculate the flow failure rate as 1 - *#Rev*/100. Fig. 1(b) reports the results, showing that the failure ratio increases with background flow arrival rate, especially when the statistics query frequency is high. This means that statistics query indeed affects the performance of switches in setting up new flows. We do the same experiment on an emulated network using Mininet, running on a Linux server with an Intel Core i5-4590 CPU@3.30GHz and 16 GB memory. The trend is similar with that of the testbed.

Then, we vary the number of queried flow entries and measure the CPU load for both Pica8 and OVS using the Pica8's built-in monitoring utility and the Linux *top* command, respectively. Fig. 1(c) reports the relationship between the switch's CPU load and the number of flow entries, for Pica8 and OVS. Not surprisingly, the CPU load increases when more flow entries are queried for both Pica8 and OVS.

III. Design

In this section, we first give an overview of CounterMap architecture, then present the syntax of the flow statistics query language, and finally show how to use CounterMap with examples.

A. System Architecture

As shown in Fig. 3, CounterMap adds four modules into the vanilla controller. The *Counter Polling Scheduler* strategically queries counters from switches periodically or on demand, and puts the counters into the *Counter Info Base (CIB)*. The *Counter Query Compiler* parses the queries from APPs, and then either retrieves flow statistics from the CIB, or requests the counters directly from switches by invoking the Counter Polling Scheduler. The *Expired Flow Tracker* monitors expired flows and stores their counters into CIB for later queries.

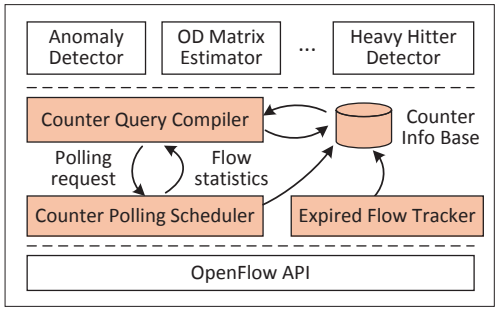

Fig. 3. System Architecture. Shaded parts belong to CounterMap.

B. CounterMap Query Language

The design of CounterMap query language is driven by two goals: first, it should be expressive enough to support various SDN applications that require flow statistics; second, it should be intuitive enough for programmers to use. At a high level, the query language lets programmers write SQL-like clauses to select counters from the CIB. The queries support features like filtering, grouping, and computing, which will be discussed in details below. Table I shows the syntax of the query language.

A SELECT (* | *m* | *c* | *f*(*m* | *c*)) clause specifies the returned results, including the matching fields (e.g., TCP-five tuple) and counters (i.e., packet count and byte count) of flow entries. In addition, *SELECT* also supports simple functions like *MAX*, *MIN*, *SUM*, *AVG*, and *COUNT* over the returned results.

A FROM (*normal* | *expired* | *full* | *dp*) clause specifies the data sources to query from. Currently, four data sources are supported: *normal* stands for all active flows, *expired* stands for expired flows, and *full* stands for their combination. These three data sources are all maintained by the CIB. Finally, *dp* means querying the data plane directly, rather than from CIB.

A WHERE(*pred*) clause filters the results according to the predicates *pred*. Predicates can be `src-ip = "10.0.0.1",dst-port!=8080, packetCount >= 80,` etc., or can be constructed using logical operators like conjunction (*and*), disjunction (*or*), and negation (*not*). For example, `where not (src-ip = "10.0.0.1" and dst-ip = "10.1.0.0/24")` filters out entries of flows sent from IP address 10.0.0.1 to IP addresses 10.1.0.0/24.

A GROUPBY (m_1[,m_2,...,m_n] | *time*) clause splits the queried results into subsets, according to matching fields or retrieval time. For example, suppose there are two source IP addresses 10.0.0.1 and 10.0.0.2, and two destination IP addresses 10.0.0.3 and 10.0.0.4. Then, the query `SELECT SUM(packetCount) FROM normal GROUPBY src-ip, dst-ip` divides the selected packet counters into four subsets (each for a source-destination IP pair), and returns the sum of packet count for each subset.

A TIMERANGE (*time1*,*time2* [,*step*]) clause specifies the time range within which counters should be retrieved, with querying period specified by *step*. Note that *time2* can be wildcard "*" to continuously retrieve the counters until the query is stopped. An application can specify the querying interval by setting *step* in unit of second. For example, TIMERANGE (10:00:00, 10:10:00, 30) returns the results at 10:00:00, 10:00:30, 10:01:00, etc.

A SAMPLE(*ratio)* clause can be used to sample entries from the CIB, in order to save query cost. For example, if the parameter *ratio* is set to 0.6, then 60% of all matched flows will be returned.

A LIMIT(*begin*[*:end* | *-1*]) clause returns flow entries from *begin* to *end* rows, ranked by storage time and collection order. Similarly, LIMIT (*begin*) returns flow entries from the 1st to *begin* row and LIMIT (*begin, -1*) returns the *begin* to the last row.

CounterMap also provides several commands to execute queries. An APP can execute a query using Execute (*query*,*frequency*,*times*). For example, Execute (*query*,10*s*,5) will query the database every 10s and execute 5 times in total. An APP can also use the Adjust(*statsCycle*) to flexibly modify the statistics query period.

C. Using CounterMap

To show how CounterMap works, we use the above query language to implement three representative applications, including DDoS attack detection, heavy hitter identification, and traffic matrix estimation. We assume that the time range of counters stored in the current CIB is [08:00:00,12:00:00], and the statistics request period is set to 5 seconds.

TABLE I. COUNTERMAP QUERY SYNTAX

Queries	*q*	:: = SELECT (* | *m* | *c* | *f*(*m* | *c*)) FROM (normal | expired | full | dp) WHERE (*pred*) GROUPBY (m_1[,m_2,…,m_n] | *time*) TIMERANGE (*time1*,*time2* [,*step*]) SAMPLE (*ratio*) LIMIT (*begin*[:*end* | -1])
Matches	m	:: = src-ip | dst-ip | src-port | dst-port | proto
Counters	*c*	:: = packetCount | byteCount
Functions	*f*	:: = MAX | MIN | SUM | AVG | COUNT
Predicates	*pred*	:: = field > value | field < value | field >= value | field <= value | field = value | field != value | *pred* and *pred* | *pred* or *pred* | not *pred* | *pred* (*pred*) | (*pred*) *pred* | field / mask

Distributed Denial of Service (DDoS) Detection. We consider a DDoS attack where a large number of packets with different source IP addresses and port numbers arrive at the same destination. For detection, we choose the entropy-based method which utilizes the Dst_IP and Dst_Port information of the flow entry [10]. The detection method proceeds as follows.

First, we query all flows, grouped by their storage time, and count the total number of items in each group. The results will be a vector, denoted by *Total*, where *Total*[*t*] corresponds to the time instance *t*.

```
SELECT time,COUNT(time)
FROM full
GROUPBY time
TIMERANGE (11:30:00,12:00:00)
```

Second, we query all flows, grouped by their time and Dst_IP, and count the total number of items in each group. The results will be a matrix *DstIP*, where *DstIP*[*t*][*i*] is the number of occurrence of the i^{th} destination IP at time *t*.

```
SELECT time, dst-ip, COUNT (dst-ip)
FROM full
GROUPBY time, dst-ip
TIMERANGE(11:30:00,12:00:00)
```

Third, we compute the probability *p*[*t*][*i*] for the i^{th} Dst_IP at time *t* as *p*[*t*][*i*]=*DstIP*[*t*][*i*]/*Total*[*i*]. Then, we can compute the entropy of destination IP as:

$$Entropy[t] = \sum_{i=1}^{n(t)} p[t][i] \log_2 p[t][i] \quad (1)$$

Similarly, we can compute the entropy of destination port. If there is significant decrease in either DstIP entropy or DstPort entropy, we should suspect that there may be DDos

attack. Then, we can query more fine-grained flow statistics by adjusting the *statsCycle* to smaller values with the command Adjust(*statsCycle*), and examine information from 12:00:00 to 12:10:00 by calling Execute (*query*,1s,600), where *query* will fetch data from *full* with TIMERANGE (12:00:00,*).

Comparatively, if we implement the above detection method with OpenFlow, we need to classify the returned flows into groups, and count the number of items in each group ourselves. This requires more lines of codes, and is thus more error-prone. In addition, the programmer cannot look back into a previous period, and retrieve the counters.

Heavy Hitter Identification. Heavy hitter refers to a small number of flows that constitute a majority of bytes or packets. For example, suppose we are interested in which (dst-ip,dst-port) pair contributes most to the total number of packets between 11am to 12am, we can make the following query:

```
SELECT time,dst-ip,dst-port,SUM(packetCount)
FROM normal
GROUPBY time,dst-ip,dst-port
TIMERANGE (11:00:00,12:00:00)
```

After obtaining the results, we can compare them with a given threshold, in order to identify the heavy hitter. Suppose the set of flow with destination IP 10.0.0.1 and port 21 is identified as a heavy hitter, we can look at its detailed information by making the following query:

```
SELECT *
FROM full
WHERE dst-ip=10.0.0.1 and dst-port=21
TIMERANGE (11:00:00,12:00:00)
```

Traffic Matrix Estimation. A traffic matrix can reveal the volume of traffic between different origin-destination pairs in a network. To build the traffic matrix from 8am to 9am, we can make the following query:

```
SELECT time,src-ip,dst-ip,SUM(byteCount)
FROM full
GROUPBY time, src-ip,dst-ip
TIMERANGE (8:00:00,9:00:00)
```

D. Optimization

According to the experiment results in section II, the counter polling process imposes a non-negligible overhead on switches. Thus, we should optimize the counter polling strategy in order to reduce such overhead. In the following, we will present a preliminary step, and leave full optimization as future work.

Let F and S be the set of flows and switches, respectively, with $|F|=m$ and $|S|=n$. Let $A_{m\times n}$ be a Boolean matrix: $A_{i,j}=1$ if flow i traverses switch j, and $A_{i,j}=0$ otherwise. Let C_j be the set of flows that traverse switch j. Let I be a vector defined as $I_j=1$ if switch j is selected for polling, and $I_j=0$ otherwise. Let L_j be the processing capacity of the switch j.

We assume that the general statistics querying cost (either communication or computation cost) of a switch j can be decomposed into two parts: a constant cost B_j, and a variable cost $R_j|C_j|$, where R_j is a coefficient. Take the communication cost for example, a flow_statistics_reply message has a minimum length of 82 bytes, and a variable length of 56 bytes per flow entry, according to OpenFlow 1.3. Then the counter polling problem can be formulated as the following mixed integer program:

$$\begin{aligned} \min\ & \sum_{j\in S}(B_j+R_j\,|\,C_j\,|)I_j \\ s.t.\ & \sum_{j\in S}A_{i,j}I_j\geq 1,\ \forall i\in F \\ & (B_j+R_j\,|\,C_j\,|)I_j\leq L_j,\ \forall j\in S \\ & I_j\in\{0,1\} \end{aligned} \tag{2}$$

This is a typical minimum set cover problem, which is NP-hard. In the following, we propose a simple heuristic algorithm to find an approximate solution. Algorithm 1 summarizes the process of Covered Flow Selection (CFS) algorithm always selects the switch which is the most cost-effective in each round. Specifically, we define the weight of switch j as:

$$w_j=\frac{|\,C'_j\,|}{B_j+R_j\,|\,C_j\,|} \tag{3}$$

, where $C'_j\subseteq C_j$ is the set of rules that have not been covered by any switch already selected.

Algorithm 1 Covered Flow Selection Algorithm

Input: F: the set of all flows, S: the set of all switches, C_j: the flows that traverse switch j.
Output: SW: the switches that have been selected for polling

```
SW ← {} // set of selected switches
CF ← {} // set of flows covered by selected switches
C'_j ← C_j, ∀j ∈ S // flows that traverse switch j but have not been covered by any selected switch
while CF != F do
    compute w_s using Eq.(3) for each switch s ∈ S \ SW
    find a switch s ∈ S with the maximal weight, while satisfying B_s + R_s | C_s | ≤ L_s
    SW ← SW ∪ {s}, CF ← CF ∪ C_s
    C'_j ← C'_j \ C_s, ∀j ∈ S \ SW
end while
return SW
```

IV. Implementation and Evaluation

A. Implementation

We prototype CounterMap based on Floodlight. For Counter Information Base, we use Redis [12], an in-memory key-value store that supports various data structures such as hashes, lists and set. We implement the Counter Query Compiler with a *parser* that parses queries and an *execution unit* that interacts with the Redis data store. The *execution unit* evaluates clauses in the order of FROM, SAMPLE, TIMERANGE, WHERE, GROUPBY, LIMIT, and SELECT. To speed up the execution, we use the LUA script language provided by Redis since version 2.6.0.

B. Evaluation

In the following, we verify the effectiveness of the CFS algorithm with experiments.

Setup. We use Mininet to emulate different network topologies, including FatTree, BCube, DCell, Stanford campus backbone network, and China Education and Research NETwork (CERNET), as shown in Table II. We run Mininet on a Linux server with an Intel Core i5-4590 CPU@3.30GHz and 16 GB

memory. In our experiment, we generate flows with pingall. For comparison, we use both the vanilla and the CounterMap-enabled Floodlight controller. The controller sends flow statistics requests every 2 seconds, and the idle_timeout of flow entries is set to 15 seconds.

Results. As shown in Table II, CFS reduces the *number of polled switches* by 29%-55%, and the *number of queried entries* by 55%-77%. Since cross-switch redundancy is not a main focus of this paper, we have not fully optimized CFS. While the reduction rate is already acceptable.

TABLE II. COMPARISON RESULTS OF CFS AND OPENFLOW FOR DIFFERENT TOPOLOGIES

topologies	size	#polled switches			#queried entries		
		OF	CFS	ratio	OF	CFS	ratio
FatTree(4)	20	20	9	55.0%	10226	3526	65.5%
FatTree(6)	45	17	12	29.4%	10934	2473	77.4
BCube(1,4)	24	19	10	47.4%	9835	3232	67.1%
BCube(1,6)	48	20	10	50.0%	9033	2366	73.8%
DCell(1,4)	25	19	10	47.4%	9557	2715	71.6%
DCell(1,6)	49	21	14	33.3%	9609	4289	55.4%
Stanford	26	20	12	40.0%	8566	3296	61.5%
CERNET	41	13	9	30.7%	8830	3747	57.6%

V. RELATED WORKS

There are many traditional flow collection tools like NetFlow [1], sFlow [2], Sniffer [3] and SNMP [4]. With the advance of SDN, some statistics collection schemes have been proposed. Here, we roughly classify them into two categories, flow collection methods and switch polling strategies.

There are some **flow collection methods** based on OpenFlow, For example, [8] presents a lightweight DDoS detection method whose flow collection module periodically requests flow entries from all flow tables of switches. Considering the scalability and cost of OpenFlow statistics collection, [10] combines OpenFlow and sFlow to detect and mitigate the anomaly detection, which separates the data collection process from the SDN control plane by the employment of sFlow. However, packet sampling can lower detection accuracy. In order to reduce flow monitoring cost, FlowSense [11] proposes a push-based passive flow monitoring approach, which listens for PacketIn and FlowRemoved messages to compute the link utilization between switches. PayLess [9] provides a flexible RESTful API at different aggregation levels for monitoring applications and introduces a flow statistics algorithm with low-overhead and high-accuracy. However, PayLess neglects the influence of expired flow entries on monitoring accuracy.

Switch scheduling strategy is essential to reduce the querying cost. OpenTM [13] proposes several algorithms for deciding which switches to query. However, the extra cost is unneglectable when there are large active flows and it is designed only for the traffic matrix application. FlowCover [14] formulates the communication cost of flow statistics request and reply as a weighted set cover problem and presents a heuristic switch polling scheme to reduce the cost. However, FlowCover only considers communication cost, while neglecting other important aspects like switch load.

Some SDN programming languages like Frenetic [16] and Pyretic [17] also support flow statistics queries. Different from CounterMap, they directly translate the queries into OpenFlow messages. Therefore, they can only return the counters of existing flows, while cannot provide historical statistics, which we have shown are also important for SDN applications.

VI. CONCLUSION AND FUTURE WORK

This paper presented CounterMap, a generic flow statistics collection and query platform for SDN applications. CounterMap actively polls counters of existing flows from OpenFlow switches, tracks counters of expired flows, and combines them to form a full map of counters across time. Built atop in-memory data store, CounterMap offers a SQL-Like query language that can be easily used by various applications. We designed a switch polling algorithm, and showed that it could reduce the overhead on the data plane. Our future work includes (1) improving the expressiveness of the query language; (2) further reducing the data plane overhead with more intelligent counter collection methods (e.g., wildcard-based).

Acknowledgement. This work is supported by the National Key Research and Development Program of China (2016YFB0800101), the National Natural Science Foundation of China (61402357, 61672425), and the Microsoft Research Asia Collaborative Research Program.

REFERENCES

[1] Cisco NetFlow site reference, http://www.cisco.com/en/US/products /ps6601/products_white _paper0900aecd80406232.shtml.

[2] sFlow.org Forum, 2012. [Online]. Available: http://www. sflow.org/

[3] Network Sniffer, [Online]. Available: http://www.network-sniffer.com/.

[4] SNMPv3 White Paper, http://www.snmp.com/snmpv3/v3white.shtml.

[5] Openflow switch specification 1.0.0, https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf.

[6] Mininet, [Online]. Available: http://mininet.org/.

[7] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in Proceedings of ACM CoNEXT, 2013, pp. 25–30.

[8] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in Proceedings of IEEE LCN, 2010.

[9] S. Chowdhury, M. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in Proceedings of IEEE NOMS, 2014, pp. 1–9.

[10] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments," Computer Networks, vol. 67, April 2014, pp. 122–136.

[11] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring network utilization with zero measurement cost," in Proceedings of PAM, 2013, pp. 31–41.

[12] Redis, [Online]. Available: https://redis.io/, 2007.

[13] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic matrix estimator for OpenFlow networks," in Proceedings of PAM, 2010.

[14] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "FlowCover: Low-cost flow monitoring scheme in software defined networks," in Proceedings of IEEE GLOBECOM, 2014, pp. 1956–1961.

[15] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen. "Scotch: elastically scaling up sdn control-plane using vswitch based overlay," in Proceedings of ACM CoNEXT, 2014, pp. 403–414.

[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, 2011, pp. 279–291.

[17] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN programming with pyretic, " USENIX Mag., vol. 38, no. 5, 2013.