Efficient Implementation of Security Applications in OpenFlow Controller with FleXam

Sajad Shirali-Shahreza, Yashar Ganjali
Department of Computer Science
University of Toronto
Toronto, Canada
{shirali, yganjali}@cs.toronto.edu

Abstract— **Current OpenFlow specifications provide limited access to packet-level information such as packet content, making it very inefficient, if not impossible, to deploy security and monitoring applications as controller applications. In this paper, we propose FleXam, a flexible sampling extension for OpenFlow designed to provide access to packet level information at the controller.**
Simplicity of FleXam makes it possible to implement it easily in OpenFlow switches and operate at line rate without requiring any additional memory. At the same time, its flexibility allows implementation of various monitoring and security applications in the controller, while maintaining balance between overhead and collected information details. FleXam realizes the advantages of both proactive and reactive routing schemes by providing a tunable trade-off between the visibility of individual flows, and the controller load. As an example, we demonstrate how FleXam can be used to implement a port scan detection application with an extremely low overhead.

Keywords-OpenFlow, Port Scan, Sampling, Software-Defined Network

I. INTRODUCTION

Network security applications, such as intrusion detection systems (IDS), usually require access to packet level information: port scan detection needs packet header information (to determine whether a connection is a scan or legitimate) and inter-arrival times (to detect scans), worm detection needs access to packet contents (to search for specific signatures), botnet detection needs connection patterns and packet contents (to distinguish between legitimate traffic such as sending email and sending spams).

Software-Defined Networking (SDN) aims at simplifying and enhancing network control and management, while making it easy to implement new applications. For some networks, it might be desirable to implement security services such as network monitoring and intrusion detection systems as controller applications in the SDN framework, due to simplicity and agility of such a solution. For instance, in small-sized networks, the cost of deploying and maintaining extra security boxes can be significantly reduced if we can implement such services as controller applications.

A preliminary version of this was presented at HotSDN 2013 (in conjunction with SIGCOMM 2013) as a short paper. This project was partly supported by NSERC SAVI Strategic Network.

However, the feasibility of such a design is not obvious, as packet level information might not be readily available in SDN controllers. OpenFlow, a widely adopted open standard for SDN, is primarily designed for routing applications [1], and mostly deals with flows rather than individual packets. There are three major information channels for the controller in the current OpenFlow specification:

1) Event-based Messages. Event-based messages are sent by the switches at events such as state change of a link or port, and usually deliver information about changes in network structure and topology.

2) Flow Statistics. Flow statistics (received packets, received bytes and duration in the current specification) are collected by the switches and pulled by the controller. This is the only way for the controller to collect information about active flows.

3) Packet-in Messages. A switch may send a packet-in message either because it did not know what to do with the packet – no matching entry found in the flow table – or as a result of a send-to-controller action in the matching flow entry. The switch may buffers the original packet and only includes part of the packet – usually the first 128 bytes – in the packet-in message.

These information channels are designed to provide flow-level (as opposed to packet-level) information to the controller. The first two do not provide any packet-level information, and the third one only provides limited access to such information. Even in the original OpenFlow proposal [1], it is suggested to direct flows that require further packet-level analysis to a separate machine dedicated to this purpose.

As a result, it is difficult and inefficient, if not impossible, to implement security applications that need packet-level information as OpenFlow controller applications.

There are two ways for an OpenFlow controller application to access packet-level information of a given flow. The first option is to not install any flow entries for the desired flow on one of the switches on the path. Every packet of the flow will be a table miss at that switch, triggering a packet-in message from the switch to the controller. The controller then needs to tell the switch to send out these packets on the correct port. The controller can continue this

process as long as it needs access to packet contents, and ends it by installing a rule to send out the packet on the appropriate port. This option was used in SDN-based port-scan detection system proposed by Mehdi et al. [2].

This approach has two major limitations. First, the controller effectively sits on the packet delivery path, potentially creating a bottleneck, and leading to increased packet delivery times. Second, the switch may and probably prefer to buffer the packet locally and only send part of it to the controller. This will limit the amount of packet content that controller can access. While this might not be a problem for some applications such as port scan detection [2] that only need packet header, it will create problems for applications such as worm detection that need to search the entire packet content.

The second option, which was suggested in [1], is to ask the switch to send a copy of each packet to another machine (probably a monitoring host other than controller, considering the load and scalability issues of the controller), during the forwarding process. While this option does not have the problems associated with the previous option, it might have a high overhead in certain scenarios. The entire flow will be sent to the monitor, regardless of whether it only needs the few first packets, only packet headers, or just a few samples every minute. The monitoring machine needs to be extremely powerful given the significant load resulting from applications such as IDS that need to process all connections. This can lead to major increases in cost and complexity of the network.

In this paper, we propose FleXam, a flexible sampling extension for OpenFlow that enables the controller to access packet-level information. Simply stated, the controller can define which packets should be sampled, what part of packet should be selected, and where they should be sent. Packets can be sampled stochastically (with a predetermined probability) or deterministically (based on a pattern), making it flexible for different applications. At the same time, it is simple enough to be done entirely in the data path. The controller can also request switches to only send part of packets that are needed (e.g. headers only, payload, etc.) and define where they should be sent, make it possible to easily manage and distribute the load.

FleXam allows developers to implement security applications that need packet level data with low overhead. As a result, the application could be run directly on the controller for small networks, eliminating the need for additional monitoring machines. More complex applications for larger networks can be implemented with the help of distributed monitors, at a fraction of the overhead of existing solutions, where all flow has to be directed to a monitor

FleXam can also be used to infer information about different flows that match a wild-carded rule (a flow entry that match more than one flow, e.g. all flows from host A to host B). This is a powerful complement to switch statistics that makes it easier to use proactive control schemes without losing visibility. As a result, we have the best of both worlds: flow setup times are eliminated, we can have a smaller flow table, and the control plane load will also be reduced [3], while we still have visibility over active flows.

II. OpenFlow Sampling Extension

Our goal is to provide a sampling feature that is general enough for different applications, simple to be implemented fully in the data-path and operate at line rate and completely tunable by the controller to balance the overhead and information detail.

FleXam includes two types of sampling: (1) select each packet of the flow with a probability of ρ, and (2) select m consecutive packets from each k consecutive packets, skipping the first δ packets of the flow.

The first case is the stochastic sampling. The second case is a generalized version of the deterministic sampling. For $m=1$, it is equivalent to the normal one out of k, or every k^{th} packet sampling. If an application needs more than one consecutive sample, it can set m to a value more than one. By choosing a very large k, an application can ensure it will only receive the first m consecutive packets. This is usually what security applications such as intrusion detection need. Finally, by changing the value of δ, the application can skip the first few packets of each flow. This can have a significant impact on the number of sampled packets, by excluding small and short flows (mice flows).

Considering that sending full packets could impose a significant load on the network, and not all applications need full packet contents, we let the controller decide what parts of sampled packets should be sent (e.g. IP header only).

Although there are previous works in the SDN domain that use sampling, they either use uniform sampling methods like sFlow, such as DevoFlow [3], or use statistics that are gathered by OpenFlow switches – e.g. received packet count – to replace the need for sampling techniques like NetFlow, such as the one proposed by Jose et al. [4].

FleXam is a per-flow sampling, so it has the advantages of per-flow sampling such as generating more accurate estimation of traffic statistics [5] in comparison to per-packet sampling methods like sFlow. It is also more useful for security applications (such as intrusion detection systems (IDS)) that need data about short-lived flows, which can be missed by uniform sampling [6] or flow-based sampling techniques that focus on heavy-hitters [7]. Note that we are not proposing a new sampling method. We are proposing an extension to OpenFlow that enables the controller to efficiently perform sampling. We believe such an extension can be extremely useful for some applications such as monitoring and security applications.

A. OpenFlow Specification Modification

In our OpenFlow implementation, we define sampling as a new action (OFPAT_SAMPLING) that could be assigned to each flow. This new action is similar to OFPAT_OUTPUT, which sends the sampled packet to the controller. The action has six parameters: *scheme*, ρ, m, k, δ and *destination*. The *scheme* parameter defines the sampling scheme that identifies which parts of the sampled packets will be sent. The m, k, and δ parameters define deterministic sampling, and the ρ parameter defines stochastic sampling. The *destination* parameter defines the host that sampled packets should be sent to. It could be the controller (e.g. for small networks that that the controller can run the application

that uses the sampled data) or another host that will process the sampled packets and communicate with the controller.

Representing sampling as an action has two main advantages. First, it can be easily added to current OpenFlow implementations without the need to modify the overall processing structure (such as matching with the flow table or performing different actions). Second, there is no overhead for flows that do not need sampling.

B. Switch Implementation

Implementing per-flow sampling in traditional networks and in solutions such as ProgME [8] requires significant changes to the hardware and packet processing flow in the switches. However, identifying the flow that the packet belongs to and keeping a record of the identified flows – a major task for per-flow sampling methods in traditional networks such as ProgME [8] – is an essential part of OpenFlow. As a result, our sampling extension could be easily implemented without requiring any significant hardware changes or software modifications.

The stochastic case is relatively straightforward to implement: for each packet, we generate a random number uniformly between 0 and 1. If it is less than ρ, the packet will be selected for sampling.

The deterministic case can also be implemented easily, without any additional memory or counter. OpenFlow switches maintain a number of counters for each flow. One of them is the *Received Packets* counter, which counts the number of packets received for each flow. Deterministic sampling of m first packets from each k packets after ignoring δ packets could be done using this counter: if

$$((Received\_Packet\_Counter-\delta)\ \%\ k)<m$$

then the packet will be sampled. This simple expression could be executed in the data-path at line rate without need to any new memory for sampling.

III. Port Scan Detection

To demonstrate how FleXam can be used to implement security applications, we show how we can implement the Threshold Random Walk (TRW) [9] port scan detection, which is one of the prominent port scan detection methods studied in different previous studies [7] [2] [10].

A. Threshold Random Walk

TRW is based on the assumption that during a port scan, the attacker tries to connect to different hosts and most of these connections fail, while the probability of a failed connection is relatively low for a benign host. This algorithm performs a sequential hypothesis testing by maintaining a likelihood ratio for each host. Each time the host initiates a connection to another host, this likelihood ratio will be increased if the connection failed, or decreased if the connection succeeded.

TRW assumes that we can mark each connection either as successful or failed when it starts. Considering that the monitoring host may not receive all packets of a connection (e.g. due to sampling [7] or as a result of unidirectional links in backbone [10]), there are slightly modified versions of it for such cases. We implemented a similar modification: a UDP connection is marked as successful if we see at least two packets from it, and a TCP connection is marked successful if we see at least one packet that is not TCPSYN.

B. Evaluation Setup

We use an OpenFlow switch simulator that we have developed for simulating packet processing. Our simulator provides an API similar to the API provided by NOX [11], so our port scan detection implementation can be easily changed to run on real hardware.

We use the benign and port scan attack data collected by Mehdi et al. [2]. The benign traffic is collected from 8 different hosts in a residential network over a period of one day and then merged together. The attack traffic is collected from three hosts performing a TCP SYN port scan attack to port 80 with 5 different rates (0.1, 1, 10, 100, and 1000 packets/second) for a period of two minutes. The attack and the benign data are provided separately. To better simulate the real world port scan attacks, in which a host might initiate an attack and at the same time generate other legitimate connections, we created 20 different trials from these data set by inserting the attack data at different times inside benign data, and averaged the results over all runs.

C. Resolving Flow-Shortening Problem

We start by describing how the FleXam deterministic sampling can be useful in port scan detection. One of the problems that per-packet uniform sampling introduces for port scan detection is flow-shortening [7]: we will only see a small fraction of flow samples. For small flows, it means that we probably see only one packet. If this is the SYN packet, then that connection will mark as a failed connection, decreasing the accuracy of port scan detection.

Deterministic sampling in FleXam can easily solve this problem. If we see the SYN packet from a flow, then we can ask the switch to send us the next packet from this flow by setting $m=1$ and $k=\infty$. As a result, the next packet of this connection (which will be the first packet that matches the flow entry rule) will be sampled. So we will definitely receive another packet from flows that are not a single SYN packet, and they will not be marked as failed connection, eliminating the flow-shortening problem.

D. Elephant Flow Exclusion

Performing uniform per-packet sampling with probability ρ is simple with FleXam: every installed flow entry rule will have a sampling action with stochastic sampling rate ρ and there is also one wild-carded rule[2] rule that matches all other packets with an action to perform sampling with probability ρ. However, uniform per-packet sampling is not suitable for port scan detection. The major problem of per-packet sampling (in addition to the flow-shortening that discussed previously) is missing most of the short flows – also known

[2] For simplicity, we assume that one single rule can describe how the rest of traffic should be routed. In reality, this one rule might be replaced with a set of rules (e.g. one rule per output port).

as flow-reduction [7] – which are important for security applications such as port scan detection [6].

To solve this problem, we identify and exclude large (i.e. elephant) flows, which enables us to focus on small (i.e. non-elephant) flows and spend our sampling budget only on them. This is based on the fact that a small number of flows carry most of the traffic [12]. Figure 1 shows the benign traffic statistics that we used. We can see that while less than 6% of flows have more than 50 packets, they carry 96% of all packets and 95% of total bytes. FleXam enables us to identify and exclude elephant flows with a very low overhead (both in terms of switch/controller and network load), without the need to complicated modifications to OpenFlow switches (e.g. DevoFlow [3] or Hedera [13]).

We use the same sampled packets that are sent by the switches for port scan detection to identify elephant flows: a wild-carded rule is installed with stochastic sampling rate ρ action that matches all packets that do not belong to any detected elephant flows. We keep track of how many packets we see from each observed flow (any flow that we are aware of since we have received at least one sampled packet from that flow in the past). If the number of packets we have received from a flow is greater than ε – the elephant flow detection threshold – then we mark this flow as an elephant flow and exclude it from sampling by installing a rule that describes how its packets should be routed but without any sampling action. This process will identify and exclude flows with size greater than ε/ρ.

As a result, for a fixed sampling budget (the acceptable network overhead), we can have a higher sampling rate for non-elephant flows. This could result in higher accuracy for port scan detection without increasing the network overhead.

E. Network Overhead

There are two tunable parameters in our port scan detection method: non-elephant flow sampling rate (ρ) and elephant flow detection threshold (ε). Increasing each of them will increase the overhead. However, they have different effects: increasing the non-elephant flow sampling rate (ρ) will increase the accuracy as we will see smaller flows with a higher probability. Increasing the elephant flow detection threshold (ε), on the other hand, will decrease the number of flows marked as elephant and reduce the flow table occupancy on switches.

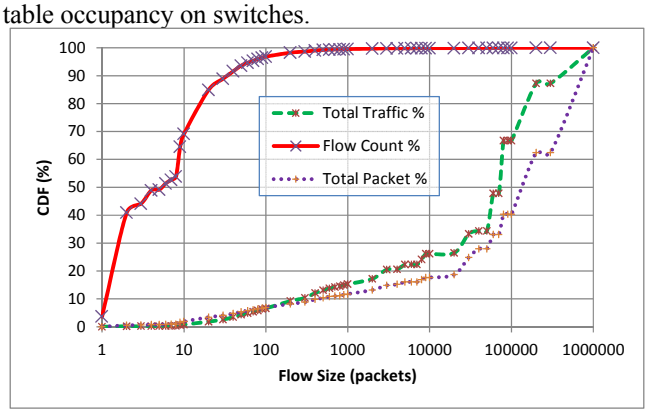

Figure 1. Benign traffic statistics.

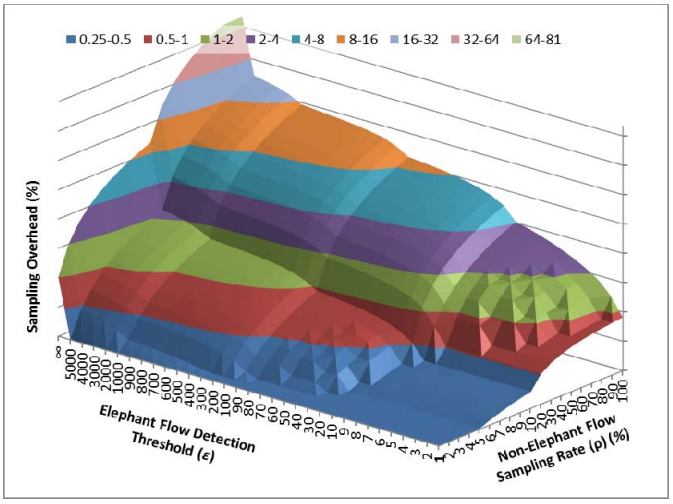

Figure 2. Sampling overhead for different (ρ, ε) values.

We tested a wide range of possible values for these parameters in our simulations: sampling rates of 1%, 2%, …, 100% and elephant flow detection threshold 1, 2, 3, …, 3000, 4000, 5000 and ∞. Figure 2 shows the overhead of different combinations of these parameters. As we see, the sampling overhead is low for a wide range of values, i.e. it is less than 0.4% for $\rho<10$ and $\varepsilon<10$.

F. Accuracy

For attack rates of 10, 100 and 1000 packets/seconds, all scanners were detected for any pairs of (ρ, ε), i.e. we have an accuracy of 100%. For these attack rates, we could select the pair $(\rho=1\%, \varepsilon=1)$ that has about 0.1% overhead for 10 and 100 pkts/s and 0.5% overhead for 1000 pkts/s.

In the case of lower attack rates (0.1 and 1 pkts/s), the accuracy depends on the values of ρ and ε. Figures 3 and 4 show the average accuracy for these attack rates as a function of ρ (averaged over all ε) and ε (averaged over all ρ). As we can see, the accuracy only depends on the sampling rate. The parameter ε only determines when an observed flow should be marked as an elephant and excluded from sampling, and has no impact on how many different flows are seen.

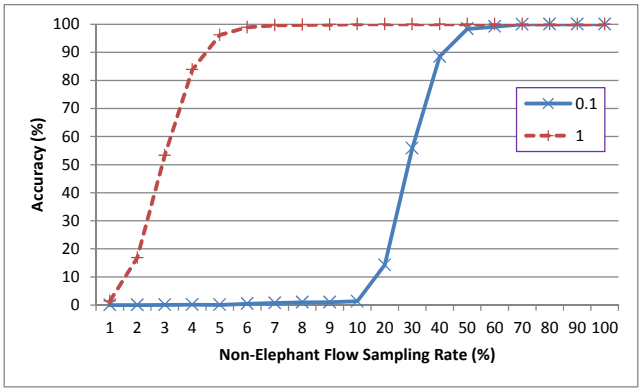

Figure 3. Average accuracy for different sampling rates.

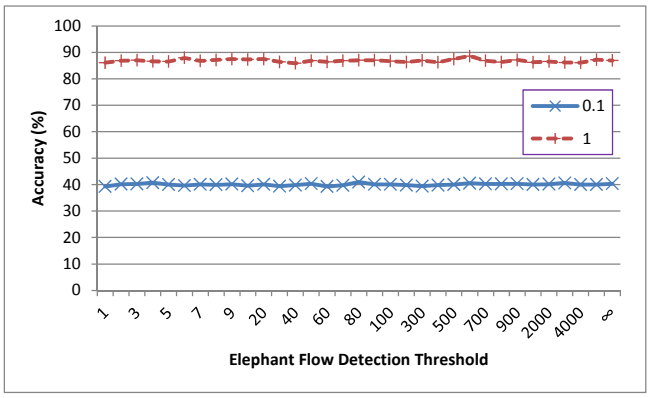

Figure 4. Average accuracy for different elephant flow detection threshold.

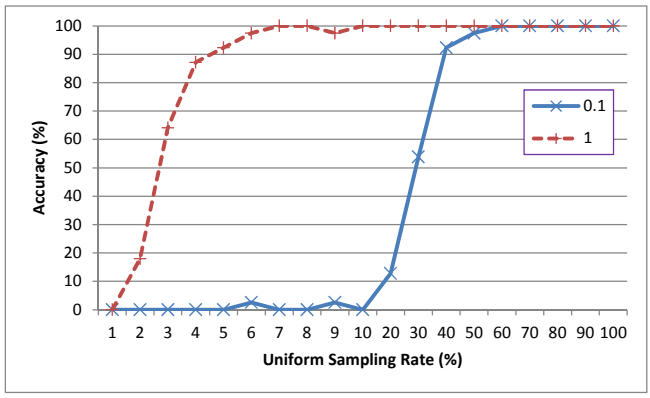

Figure 5. Uniform sampling accuracy.

The reason that accuracy for an attack rate of 0.1 pkts/s is zero for small values of ρ is that the attack duration is only 2 minutes, so there are only 12 attack flows, while TRW needs at least 7 consecutive failed connections to mark a host as a scanner. Figure 3 shows the average accuracy over all values of ε. Among pairs with 100% accuracy, the pair of *(ρ=50%, ε=3)* has the lowest overhead of 0.7% for an attack rate of 0.1 pkts/s, and the pair of *(ρ=5%, ε=4)* has the lowest overhead of 0.25% for an attack rate of 1 pkts/s.

False positives – benign hosts marked as scanners – are extremely rare with our method, mainly due to the deterministic sampling that enables us to solve the flow-shortening problem: only in 0.06% of more than 50,000 simulations one of benign hosts incorrectly marked as scanner, and there was no clear relation between the attack rate, the non-elephant flow sampling size, or the elephant flow detection threshold and these false detections.

G. Comparison

In this section, we compare the accuracy and overhead of our algorithm with other possible options to perform port scan detection. We start with uniform sampling. The special case of $\varepsilon = \infty$ represents the uniform sampling: no flow will be excluded from the sampling and all packets will be uniformly sampled with probability ρ. Figure 5 shows the accuracy of uniform sampling. The minimum required sampling rate for 100% accuracy is 60% for an attack rate of 0.1 pkts/s and 7% for an attack rate of 1 pkts/s. For higher attack rates (10, 100, and 1000 pkts/s), 1% sampling leads to 100% accuracy, so in these cases, the overhead is 1%.

Next, we compare the overhead of our method with the scheme proposed by Mehdi et al. [2] that uses current OpenFlow packet-in messages. In their implementation, they do not install any rule upon receiving the first packet of the flow, and wait for the second packet to determine whether the connection was successful or not, and then install the rule. So the overhead of their method is equal to the case of *(ρ=100%, ε=2)*. The overhead associated with this approach is shown in Table 1. These results show that FleXam significantly reduces the network overhead for port scan detection in comparison to uniform packet sampling or approaches such as Mehdi et al. [2] that use packet-in messages to access to packet contents.

TABLE I. OVERHEAD COMPARISON OF DIFFERENT METHODS

Method	Attack Rate				
	0.1	1	10	100	1000
Our Method	**0.7%**	**0.25%**	**0.1%**	**0.1%**	**0.5%**
Uniform Sampling	60%	7%	1%	1%	1%
Mehdi et al. [2]	1.1%	1.2%	2%	9%	47%
Reactive OpenFlow Routing	**0.7%**	0.8%	1.5%	8.7%	47%

Finally, we compare the overhead of our method with regular reactive routing in OpenFlow without any port scan detection. Here, the first packet of each flow is sent to the controller (via a packet-in message) and then the controller installs the flow entry in the switches. This is equal to the case of *(ρ=100%, ε=1)*. The overhead of these packet-in messages is shown in Table 1, which shows that our port scan detection overhead is even less than normal reactive routing overhead.

Figure 6 shows the flow visibility of controller (percentage of flows that the controller received at least one packet from them) for different non-elephant flow sampling rates. Flow visibility only depends on non-elephant flow sampling rate (ρ), similar to the accuracy (see section III.F). Although the reactive routing gives the controller 100% flow visibility, the controller may not need such a high flow visibility. On the other hand, simple proactive routing reduces the flow visibility to 0%, because all packets will be routed by switches without reaching the controller. FleXam enables the controller to achieve a balance between flow visibility and controller overhead depending on control applications requirements. Regular proactive or reactive routing schemes are special cases in this model.

IV. OTHER POSSIBLE APPLICATIONS

In the previous section, we demonstrated how we can use FleXam to implement port scan detection as an example of security applications. In addition to security applications, FleXam can be used for other types of applications:

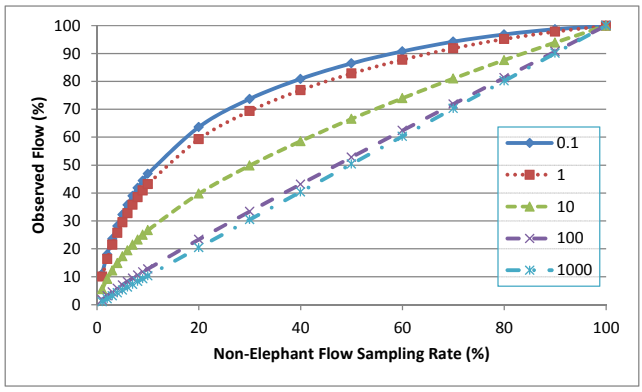

Figure 6. Flow visibility of different sampling rates.

Traffic Classification: The controller could analyze the data payload of sampled packets, e.g. the first few packets [14], to determine the traffic type of flow to provide Quality of Service or block certain type of traffics. For example, it could reserve some low-load routes for time-sensitive flows such as VOIP, then detect and reroute VOIP traffic through them. This is not possible in current specifications as the controller has no access to packet contents.

Quality of Service: The controller could use the arrival time of sampled packets to deduce the overall behavior of the flows for QoS purposes. For example, it could distinguish between constant rate flows and flows that send a burst of data and then wait. This information could be used to predict possible congestion or provide better QoS for delay-sensitive or loss-sensitive flows. In other words, the controller can infer fine-grained traffic properties that "have a significant impact on effectiveness of switching mechanisms and traffic engineering" [15].

Diagnostics: Sampling can help network administrators become aware of what is happening in the network. In the extreme case of sampling with a rate of 1/1, a copy of each packet would be sent to the controller. Although this would impose a huge overload, it would enable the operator to see exactly what is happening in a switch. In other words, the operator could remotely see every packet – or a subset of packets – that passes through the switch to determine possible causes of a problem. This will be similar to OFRewind [16]. A selective sampling approach can control the overhead, e.g. only sample packets on the switch that has problem, or packets from a specific application or machine that may cause the problem.

V. CONCLUSION

In this paper, we proposed FleXam, a flexible sampling extension for OpenFlow that enables the controller to access packet-level information such as packet contents. Our extension is flexible for different applications (supporting both deterministic and stochastic sampling), yet simple enough to be implemented entirely in switch data-path and operate at line rate without requiring any additional memory. Using simulations, we showed how FleXam can be used for efficient implementation of a port scan detection application, as an example of security applications that need access to packet-level information.

In addition to flexibility, FleXam is also scalable. For small networks, the samples can be sent to the controller and applications running on the controller. This can eliminate the need for deployment and maintenance of multiple devices for network management and monitoring, leading to significant reductions in cost. On the other hand, for larger networks where having dedicated middleboxes is unavoidable, FleXam can still be used to significantly reduce the load to such boxes. The downside here is that we need to update the box to work with sampled traffic, a requirement that sometimes can be challenging.

Security applications are not the only applications that can benefit from FleXam. For example, the elephant flow detection that we performed to eliminate elephant flows from sampling, could be used in routing applications to optimize the routing path of elephant flow while eliminating flow setup time for mice flows, similar to what DevoFlow [3] tries to do. We leave other applications of FleXam as a future work.

REFERENCES

[1] McKeown N., et al. 2008. OpenFlow: enabling innovation in campus networks. Comput. Commun. Rev. 38(2). 69-74.

[2] Mehdi, S.A., Khalid, J., and Khayam, S.A. 2011. Revisiting traffic anomaly detection using software defined networking. In RAID'11. 161-180.

[3] Curtis, A.R., et al. 2011. DevoFlow: scaling flow management for high-performance networks. In SIGCOMM'11. 254-265.

[4] Jose, L., Yu, M., and Rexford, J. 2011. Online measurement of large traffic aggregates on commodity switches. In Hot-ICE'11. Paper 13.

[5] Hohn, N., and Veitch, D. 2006. Inverting sampled traffic. IEEE/ACM Trans. Netw. 14(1). 68-80.

[6] Salem, O., et al. 2010. A scalable, efficient and informative approach for anomaly-based intrusion detection systems. Int. J. Netw. Manag. 20(5). 271-293.

[7] Mai, J., et al. 2006. Is sampled data sufficient for anomaly detection? In IMC '06. 165-176.

[8] Yuan , L., et al. 2011.ProgME:towards programmable network measurement. IEEE/ACM Trans. Netw. 19(1).115-128.

[9] Jung, J., et al. 2004. Fast portscan detection using sequential hypothesis testing. In S&P 2004. 211-225.

[10] Sridharan, A., et al. 2006. Connectionless port scan detection on the backbone. In IPCCC 2006. 10-576.

[11] Gude, N., et al. 2008. NOX: towards an operating system for networks. SIGCOMM Comput.Commun. Rev. 38(3).105-110.

[12] Estan, C., and Varghese, G. 2003. New directions in traffic measurement and accounting. ACM Trans. Comput. Syst., 21 (3), 270-313.

[13] Al-Fares, M., et al. 2010. Hedera: dynamic flow scheduling for data center networks. In NSDI'10.

[14] Hassas-Yeganeh, S., et al. 2012. CUTE: traffic Classification Using Terms. In ICCCN'12.

[15] Benson, T., et al. 2010. Understanding data center traffic characteristics. In SIGCOMM CCR. 40(1). 92-99.

[16] Wundsam, A, et al. 2011. OFRewind: enabling record and replay troubleshooting for networks. In USENIX ATC'11.