A SDN-based Sampling System for Cloud P2P Bots Detection

JIANG-YONG SHI¹, JIE HE²* AND YUE-XIANG YANG¹

¹School of Computer
National University of Defense Technology
Changsha, China
²Department of Information Engineering
²Officers College of PAP
Chengdu, China

E-mail: {shijiangyong; yyy}@nudt.edu.cn; jack.237@163.com

Cloud network monitoring is a crucial problem in protecting cloud security. As the traffic is huge and the network structure is dynamically changing, it is hard to monitor collaborative attacks such as P2P botnets. This paper presents a two-stage sampling system based on SDN, which is able to extract security related packets from the vast cloud traffic thus reducing the performance cost. We implement a prototype of the sampling method to detect P2P bots in cloud. The prototype is evaluated with real-world P2P botnet traffics. The experimental results demonstrate that our method can identify potential P2P bots quickly and accurately with few false positives and high detection accuracy at an acceptable performance cost.

**Keywords:** sampling, SDN, cloud, P2P botnet, security

1. INTRODUCTION

As the development of cloud computing, more and more enterprises and government organizations are moving their IT infrastructure into cloud. There is an urgent demand for network monitoring to protect the security of digital assets and enterprise network from being attacked. However, due to the inability of efficiently inner-cloud network traffic monitoring, it’s very likely that one insider attacker could attack or infect the others on the same cloud or even the same host machine where their virtual machines co-exist. Besides, the homogeneity of virtual machines facilitates the spread of P2P bots on cloud servers, which might cause deny of service (DoS) to the server, and the bots could hide them through the cover of the cloud.

The dynamic nature of cloud computing makes it hard to statically monitoring the network. Virtual machine migration and dynamic resource scheduling make the network structure an unstable state. Software Defined Network (SDN), which is proposed to manage the dynamic cloud network, is also being used to implement security-related applications, such as DDoS prevention and IDS. However, one of the main problems of SDN-related security applications is the performance bottleneck of the controller. As a centric model, SDN controller is crucial in collecting security related information of the network, which would get overload when the traffic is large, especially in a cloud environment. So sampling is required to reduce the performance cost to the controller.

*Corresponding author, Email: jack.237@163.com*
Based on these demands, we proposed a two-stage sampling system to monitoring network status in cloud. In the first stage, we capture the package header information in a set of time window, which can be easily accomplished by SDN flow query commands. The header information consists of five-tuples, including source IP address, destination IP address, source port, destination port, and IP protocol type. Based on this simple information, we can use a set of algorithms to roughly judge the network status and get a small set of suspicious flow and host. For example, we can use Sample&Hold [1] for detecting elephant flows and Lightweight [2] for detecting DDoS traffic. In the second stage, we can use the SDN controller to push static flows targeted to these suspicious traffics, to redirect them to deep packet inspection (DPI) engine for further analysis, including P2P bots detecting.

Compared with mirror dispatching method, which needs expensive equipment and typically has delays in detecting threats, our method is more economical and efficient considered with performance and cost. What’s more, we use a modular design to enable the future expansion for security applications. For example, we can use different modules to detect different threats, including P2P, IDS, DDoS and so on. In addition, the Cloud Service Provider (CSP) can benefit in using the centric controller model of SDN to detecting collaborative attacks such as DDoS and P2P botnet.

To test the performance of our system, we implement a P2P botnet detection module to effectively detect P2P bots in cloud. As our module is working online, it is very timely compared with other offline methods. Besides, with the help of SDN controller, we can respond quickly based on the detection results. For example, we can block the P2P botnet traffic by updating the switch flows as long as we detect it.

Overall, our main contributions include:

1. We propose a cloud traffic sampling system based on SDN, which is able to monitor real-time network traffic of cloud and detect collaborative network attacks.
2. We introduce a two-stage sampling rate calculate algorithm and implement it with OpenFlow protocol, which aims to use limited resources to sample as many security-related packets as possible.
3. We implement a prototype system, and evaluate it using real-world P2P botnets traffic datasets. The experimental results show that our system is able to identify potential P2P bots in cloud quickly with high accuracy and few false positives and greatly increase the proportion of botnet-related packets in all sampled packets.

2. RELATED WORK

SDN is flexible for its detachment of control plane and data plane. Users can write their applications easily based on the data collected by controller. However, as the network is more and more busy, sampling is required to effectively obtain useful SDN information. OpenNetMon [3] uses an adaptive rate to poll the switch so as to measure the throughput, packet loss and delay. The adaptive policy is based on the rate of new flow generation, which can reduce unnecessary flow samplings when there are stable traffic flows. However, it cannot overcome the controller limitation when there is
outburst traffic. OpenSample [4] uses a math model to prove that there exist two distinct flow samples over the identical flow. Based on the information carried by the two samples, mainly sequence numbers and time stamps, it can estimate the packet rate and detecting elephant flows. OpenSample is limited in packet rate estimation while security related network monitoring needs more information than that. Other methods such as Sample&Pick, FleXam and Wildcard Samper [5-7] mainly concerns on heavy flow detection and traffic scheduling, instead of security monitoring. What’s more, they need to modify either OpenFlow protocol or the switch hardware, which are complex to implement.

Some prior works have being done in exploring the application of SDN-based network security monitoring, including malware detection, DDoS, IDS and so on [8-10]. However, these works mainly focus on anomaly detection and signature-based detection, which has significant features of traffic bursting over specific host or specific port, or features matching existing malicious behaviors signature, including urls, traffic bytes and so on. Different from elephant flows detection and anomaly detection, P2P control traffic is much stealthier with its decentralized architecture. However, there are still some features which can be used to identify P2P control traffic and P2P botnet, including multiple transport protocols, package size, in-degree, out-degree and so on.

BotHunter [11] and BotMiner [12] identify bots based on detectable malicious activities, including scanning, spamming, exploiting, DDoS, etc. Unfortunately, the malicious activities of P2P botnets are becoming stealthier and cannot be easily detected any more, thereby limiting the applicability of these approaches. TDG-based (traffic dispersion graphs) approaches like BotGrep [13] and BotTrack [14] detect P2P botnets by analyzing the communication graphs extracted from network flows collected over multiple large networks (e.g., ISP networks). Although they do not rely on the malicious activities for bots identification, they need information about infected hosts collected from additional systems such as honeypots to bootstrap the detection and a global view of Internet traffic, which is hard to acquire. PeerSorter [15] and PeerDigger [16] mainly detect P2P botnet traffic based on cluster flow classification and analyzing, which is easy to implement and efficient. However, due to the large traffic in cloud data centers, it is hard to work online and detect P2P botnets in real time.

Over system use PeerDigger as the DPI engine for P2P bots detecting. We combine it with SDN-based network monitoring approaches, to make it usable online. Besides, we implement an effective two-stage sampling method based on SDN, which can overcome its performance limitation and make it work in real time in cloud environment. Our work can be extended to build a general online cloud network monitoring system, which focuses on security application developments over SDN.

3. ARCHITECTURE AND ALGORITHMS

3.1 Sampling Architecture

As the Fig. 1 shows, the architecture of our sampling system mainly consists of three modules: temp flow capture, sampling rate decision and packet sampling.
The temp flow capture module is based on Floodlight controller and has two features. Firstly, it default forwards network traffic which is active and accessible, and generates temp flows for the traffic. These flows survive only a short period of time, thus the controller must timely query the switch to get the status of the network.

Fig. 1. Architecture of our sampling system

The sampling rate decision module is made up of two steps, namely the suspicious IP detector and sampling rate calculator. The detector detects suspicious IPs based on the flows information, which is five-tuples of <src.ip, dst.ip, src.port, dst.port and proto> in our special purpose to detect P2P hosts. Then the calculator decides how to assign the preset sampling rate to different IPs based on the suspicious degree of the IPs. The more suspicious the IP is, the larger sampling rate will be assigned to the flows related with the IP. The sampling rate decision module is a standalone application, which makes it extensible and can be customized to better detect other kinds of threats, including DDoS, port scanning, application protocols detection and so on.

Lastly, the packet sampling module is used to capture the full packet whose IP is suspicious by installing static flows on the switch and adaptively send the packets to the Deep Packet Inspection (DPI) module. The static flows are installed by the controller according to the suspicious detection results. A timeout is set to control the survival time of the flows based on their sampling rate calculated in the last stage.

The architecture is simple, but a lot of challenges need to be overcome in order to make it work. We will first describe the sampling decision module and then discuss the details in solving these challenges. The sampling decision module is customized based on the security needs. For different threats, we might implement different sampling
strategies. Here, we propose a two-stage algorithm for P2P detecting, which consists of suspicious IP detecting algorithm and adaptive sampling algorithm.

3.2 Suspicious IP detecting Algorithm

This phase aims to monitor the outbound traffic at the edge of cloud networks and narrow down suspicious internal IP addresses that are likely P2P bots as soon as possible. It consists of two components: Raw Flow Counting Table and Suspicious IPs Detector.

**Raw Flow Counting Table.** In order to reduce the time and memory consumption of our system, we only monitor the TCP and UDP packets sent from internal IP addresses in this phase. As shown in Table 1, each entry in the Raw Flow Counting Table records a raw flow and its packets count and total bytes during a statistic time window. A raw flow is defined as a set of TCP/UDP packets sent from the same internal IP address and port number to the same external IP address and port number. In this table, $P$ represents the transport-layer protocol of the raw flow. $IP_{in}$, $PORT_{in}$, $IP_{ex}$ and $PORT_{ex}$ respectively represent the internal/external IP address and port number. These five items can get from the OpenFlow matching fields. $PKT$ and $BYTE$ represent its packets count and total bytes. The $SYN$ item represents the count of packets that contain $SYN$ flag in a TCP raw flow. These three items can get from OpenFlow counters. After each time window, the data in this table is transferred to the Suspicious IPs Detector for further analysis, and then are reset to 0. The end of each time window also triggers the Suspicious IPs Detector to identify IP addresses that are related to potential P2P bots, and the Sampling Rate Calculator to recalculate the instant sampling rate for each internal IP address.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$IP_{in}$</th>
<th>$PORT_{in}$</th>
<th>$IP_{ex}$</th>
<th>$PORT_{ex}$</th>
<th>$PKT$</th>
<th>$BYTE$</th>
<th>$SYN$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>192.168.0.2</td>
<td>80</td>
<td>225.12.14.54</td>
<td>80</td>
<td>3</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>UDP</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Suspicious IPs Detector.** After each time window of T, this module analyzes the information in the Raw Flow Counting Table and tries to identify a small number of suspicious IP addresses that are likely P2P bots. In our previous work [16], we have demonstrated that P2P bots have two significant characteristics compared to normal hosts. First, according to their P2P network nature, there are number of aggregation flows (flows with same packets count and total bytes) in the traffic generated by them, and the destination IP addresses of the aggregation flows in the same cluster always spread across a large amount of networks. In other words, the external IP addresses of the aggregation flows of P2P bots have a large number of distinct BGP prefixes. Second, according to their botnet nature, they are more likely to repeatedly contact the same external IP addresses than benign hosts. Specifically, the aggregation flows generated by P2P bots tend to re-connect the same external hosts to exchange packets.

To identify suspicious IP addresses, we design a suspicious host identification algorithm, which is described in Algorithm 1. This algorithm operates in the following steps. First, we consider the entries (E) whose $SYN$ is greater than 1 as unexpected raw flows and discard them from the Raw Flow Counting Table. Then, for every internal IP address ($IP_{i}$) in the Raw Flow Counting Table, we aggregate all entries related to $IP_{i}$ to
different clusters, i.e. $C_j(ip_i)$. The entries in the same cluster $C_j(ip_i)$ have the same value of PKT and BYTE. Third, we first translate $C_j(ip_i)$ to $\Gamma_j$, where $\Gamma_j$ is a set of $IP_{ex}$ extracted from entries in $C_j(ip_i)$, i.e., $\Gamma_j = \{ip_{ex}^1, ip_{ex}^2, ..., ip_{ex}^q\}$, and then define the re-connection number (RCN) of $C_j(ip_i)$ as the number of repeated elements in $\Gamma_j$, denoted as,

$$RCN_j = \sum_{x=1}^{q} (\text{count}(ip_{ex}^x) - 1)$$  \hspace{1cm} (1)

After that, we define re-connection ratio (RCR) of $ip_i$ as the maximum re-connection number of each $C_j(ip_i)$, i.e.,

$$RCR(ip_i) = \max\{RCN_1, ..., RCN_q\}$$  \hspace{1cm} (2)

Forth, we consider of the $\Gamma_j$ which has the maximum $RCN$, and count the number of distinct $BGP$ prefixes among its elements, denoted as $BGP(ip_i)$. At last, we label $ip_i$ as a suspicious IP address if $BGP(ip_i)$ and $RCR(ip_i)$ are greater than thresholds $\theta_{BGP}$ and $\theta_{RCR}$, respectively.

Algorithm 1: Suspicous Host Identification Algorithm

Input: Raw Flow Counting Table (RFCT)
Output: a set of suspicious IP addresses $Set_{sus}$

begin
  for each entry $E$ in RFCT
    if $E.SYN > 1$ then
      Delete $E$;
    end if
  end for
  for each $IP_{in} ip_i$ in RFCT
    Split $\{E | E.IP_{in} = ip_i\}$ to clusters: $\{C_1(ip_i), ..., C_n(ip_i)\}$
    for each cluster $C_j(ip_i)$
      Extract its $IP_{ex}$ to set $\Gamma_j = \{ip_{ex}^1, ip_{ex}^2, ..., ip_{ex}^q\}$;
      $RCN_j = \sum_{x=1}^{q} (\text{count}(ip_{ex}^x) - 1)$;
      $RCR(ip_i) = \max\{RCN_1, ..., RCN_q\}$;
      for each set $\Gamma_j$
        if $RCN_j = RCR(ip_i)$ then
          $BGP(ip_i) = BGP(\Gamma_j)$;
          if $BGP(ip_i) > \theta_{BGP}$ and $RCR(ip_i) > \theta_{RCR}$ then
            Add $ip_i$ to $Set_{sus}$;
            return $Set_{sus}$;
          end if
        end if
      end for
    end for
  end for
end.

3.3 Adaptive Sampling Algorithm
In this phase, the instant sampling rate of every internal IP address is adaptively tuned according to the given target sampling rate. Arrived packets are sampled and delivered to the DPI for further accurate detection. It has two components: Sampling Table and Sampling Rate Calculator.

**Sampling Table.** Each entry in the Sampling Table records a suspicious IP address ($IP_{sus}$) and its related information, including RCR, BGP, packets count (CNT) and instant sampling rate ($SR_{ins}$). At the end of each time window, Sampling Table is updated according to the set of suspicious IP addresses identified by the Suspicious IPs Detector. Moreover, we discard entries which have not been updated for two or more time windows. In other words, the entries whose $IP_{sus}$ have not been identified by the Suspicious IPs Detector for two or more continuous time windows will be deleted from the Sampling Table. The instant sampling rate of every entry will be updated by the Sampling Rate Calculator after each time window. All non-suspicious internal IP addresses share a same instant sampling rate which is also updated by the Sampling Rate Calculator. On arrival of a packet, Sampling Table queries its instant sampling rate $SR_{ins}$ based on its internal IP address, and samples this packet with probability $SR_{ins}$. All of the sampled packets and corresponding $SR_{ins}$ are sent to the DPI for further in-depth analysis.

**Sampling Rate Calculator.** This module dynamically calculates the instant sampling rate for every internal IP address to fulfill a target: to sample as many packets as possible that are related to the identified suspicious IP addresses while keeping the actual sampling rate close to the target sampling rate.

To accomplish this purpose, we design a two-step sampling rate calculate algorithm, which we call $T$-Sampling. This algorithm consists of two major steps as follows. In the first step, we allocate as many available resources as possible to the group of the identified suspicious IP addresses. The remaining available resources, if there is still anything left, will be used to sample packets of other non-suspicious internal IP addresses. In the second step, the resources that have been allocated to the group of suspicious IP addresses in the previous step are further subtly allocated to each single suspicious IP address. The algorithm is described in Algorithm 2. $SR_{tar}$ is the pre-defined target sampling rate. $p_{sus}$ is the proportion of packets related to all suspicious IP addresses in all observed packets. $\{p_1, p_2, ..., p_m\}$ is the proportion of packets related to each $IP_{sus}$ in all suspicious packets. $m$ is the number of entries in the Sampling Table. $\{SR_{ins}^1, SR_{ins}^2, ..., SR_{ins}^m\}$ is the instant sampling rate of each $IP_{sus}$. $SR_{non\_sus}$ is the instant sampling rate of all non-suspicious internal IP addresses. $SR_{sus}$ is the overall sampling rate of the group of suspicious IP addresses.

**Algorithm 2:** Two-Step Sampling Rate Calculate Algorithm

```
Input: $SR_{tar}$, $p_{sus}$, $p_1$, $p_2$, ..., $p_m$
Output: $SR_{ins}^1$, $SR_{ins}^2$, ..., $SR_{ins}^m$, $SR_{non\_sus}$
begin
    if $m == 0$ then
        $SR_{non\_sus} = SR_{tar}$ ;
        return $SR_{non\_sus}$ ;
```
end.
\[ SR_{sus} = SR_{tar} \times p_{sus} ; \]
if \( SR_{sus} > 1 \) then
\[ SR_{non} = SR_{sus}^{1} = \ldots = SR_{sus}^{n} = 1 ; \]
\[ SR_{non \_sus} = (1 - \frac{SR_{sus} \times p_{sus}}{SR_{tar}}) \times \frac{SR_{tar}}{1 - p_{sus}} ; \]
return \( \{ SR_{sus}^{1}, SR_{sus}^{2}, \ldots, SR_{sus}^{n}, SR_{non \_sus} \} \); end.
else
\[ SR_{non \_sus} = 0; \]
for each \( IP_{sus}^{i} \)
\[ V_{i} = \omega_{i} \times BGP_{i} + \omega_{j} \times RCR_{i} ; \]
\[ V_{sus} = \sum_{i=1}^{n} V_{i} ; \]
for each \( IP_{sus}^{i} \)
\[ K_{i} = V_{i} - \frac{V_{sus} \times P_{i}}{SR_{sus}} ; \]
Reorder Sampling Table in inverted order by \( K_{i} \);
for each \( i = 1, 2, \ldots, m \)
\[ SR_{sus}^{i} = SR_{sus} \times \frac{V_{i}/V_{sus}}{P_{i}} ; \]
if \( SR_{sus}^{i} > 1 \) then
\[ SR_{sus}^{j} = 1 ; \]
for each \( j = i+1, \ldots, m \)
\[ V_{j \_+} = K_{i}/(m - i) ; \]
return \( \{ SR_{sus}^{1}, SR_{sus}^{2}, \ldots, SR_{sus}^{n}, SR_{non \_sus} \} \);
end.

We introduce a budget scheme to allocate instant sampling rates for different subsets [17]. Suppose there are \( n \) subsets and subset \( i \) has \( p_{i} \) proportion of packets in total packets. We set budget \( b=1 \) and divide it into \( n \) parts. Each subset \( i \) gets a budget \( b_{i} \), i.e. \( \sum_{i=1}^{n} b_{i} = 1 \). Then, we can calculate the instant sampling rate for subset \( i \) as \( SR_{sus} = SR_{tar} \times \frac{b_{i}}{p_{i}} \). In the first step of T-Sampling, we divide all internal IP addresses into two subsets: the group of suspicious IP addresses and the group of non-suspicious IP addresses. We set the budget of the group of suspicious IP addresses to 1, since we want to allocate as many budget as possible to them. Then, if anything is left, we give the remaining budget to the group of non-suspicious IP addresses. In the second step, we further allocate \( SR_{sus} \) to every single suspicious IP address. In this case, each \( IP_{sus} \) in the Sampling Table is a subset, namely, we get \( m \) subsets. The budget \( b_{i} \) of each suspicious IP address \( IP_{sus}^{i} \) is determined by its values of BGP and RCR. The greater its
BGP and RCR are, the larger its budget should be, since BGP and RCR are two significant indicators of the existing of P2P bots. Thus, we allocate budget for each $IP_{sus}^i$ as $b_i = \frac{V_i}{V_{sum}}$, where $V_{sum} = \sum_{i=1}^{m} V_i$ and $V_i = \omega_1 \times BGP_i + \omega_2 \times RCR_i$. We assign $\omega_1 = 0.2$ and $\omega_2 = 0.8$, since we consider RCR is more important than BGP in the context of P2P botnet detection. In this way, some $IP_{sus}$ may be allocated with more budget than its need. Therefore, we reorder the entries of the Sampling Table in inverted order by the value of $K_i$ before calculating their instant sampling rates, where $K_i = V_i - \frac{V_{sum} \times p_i}{SR_{sus}}$. $K_i$ indicates the difference between the budget which is allocated to $IP_{sus}^i$ minus the maximum budget that $IP_{sus}^i$ needs. If $K_i$ is greater than 0, we equally split the extra budget and allocate them to other $IP_{sus}$ who have smaller $K_i$. Since $p_{sus}$ and $\{p_1, p_2, ..., p_m\}$ cannot be obtained precisely in advance, we dynamically estimate them using weighted moving average (WMA), i.e., $p_i = \lambda_1 p_i^{pre} + \lambda_2 p_i^{cur}$, where $p_i^{pre}$ and $p_i^{cur}$ are the observed value in the previous and current time windows. We set $\lambda_1 = 0.2$ and $\lambda_2 = 0.8$ in this algorithm.

4. IMPLEMENTATION DETAILS

4.1 Raw Flow Counting

We use Openvswitch (OVS) as the SDN switch and Floodlight as the SDN controller [18]. In the raw flow counting phase, we need to count the SYN flag. However, Floodlight doesn’t support TCP flags matching by the time we write this paper, so we must manually add the matching field to it. OVS 2.1 and above already supports TCP flags matching, but we need to modify the controller to be able to detect and set the new matching field. In floodlight, we need to define the OXM (OpenFlow Extendable Module) for TCP flags. Corresponding to OXM speculation, the Type-Length-Value (TLV) structure for TCP flags is shown in Fig. 2.

![Fig. 2 TLV structure of TCP Flags](image)

The length is only two bytes considering the limited type of TCP flags, including FIN, SYN, RST, PSH, ACK, URG, ECE, CWR, NS. Even though we only use SYN, other flags can be used to determine the state of the TCP session, which are useful in other traffic engineering applications. As floodlight uses oxigen as its backend, we need to create a matching module under openflow_output using the TLV header and add corresponding structures in other related source files. As we use the forwarding module
to generate temp flows, we also need to modify the forwarding module to match TCP flags. The patching files will be open-sourced at github.

The statistic time window for raw flow counting is set up as 5 minutes to collect sufficient information for suspicious IP detecting. However, this time greatly exceeds the default survive time (5s) of the temp flows, which means we need to query the switches every 5s to get the statistics before the temp flows are removed. We call the survival time of each flow a slice, and add a temp list to store the flows in last slice. Each time a flow of the slice is added to the counting table of the window, it will check if it is counted in the last slice, thus avoiding the duplicated flows when we merge the 60 slices into one window.

4.2 Packet Sampling

After sampling rate is calculated, implementing the sampling rate to the switches needs careful handling. Even though traditional sampling technologies such as Netflow and sFlow provide sampling function, they use uniform sampling rate for all traffics. Dynamical sampling rate adjustment is provided by sFlow, but it only adjusts the sampling rate based on the overall traffic throughput, rather than based on the importance of each flow.

To implement per-flow sampling rate to SDN switches, we covert the sampling rate to a time concept of OpenFlow protocol. For those monitoring jobs which only need packet header information (such as elephant flow detection), we just install suspicious IP flows and periodically query the flows to get statistics. The query interval is determined by the sampling rate and statistic time window as,

\[
\text{interval} = \text{rate} \times \text{window}
\]  

For those monitoring jobs which require deep packet information (such as P2P bots detection), we can install static flows with suspicious IP and set the flow actions to forwarding packets to controller using packet-in message. However, to reduce the cost introduced by frequently packet-in messages, we set the static flows to forward suspicious packets to a separate DPI system. The survival time (hard timeout) of the flows is calculated based on the sampling rate as,

\[
\text{timeout} = \text{rate} \times \text{window}
\]  

As an example, if we calculated the sampling rate to be 0.5 in the second stage, and the statistic time window is 5 minutes, we can get the hard timeout for the forwarding flows to be 2.5 minutes. That means the statistic flows will expire when it’s idle more than 5 seconds (idle timeout) or after 2.5 minutes (hard timeout) regardless of its activity. In a short window, it may seem like a continuous sampling of 2.5 minutes. But over the longtime, it is a dynamical changing sampling rate at no fix interval as the idle time is at random. This is not only efficient in performance, but also resistant to attacks which
detect periodic sampling using network delay probing and bypass the sampling by attacking during the interval of samplings.

### 4.3 Performance Cost

The performance cost of our sampling system mainly consists of three parts: the storage cost of switches to store the temporal flows, the controller overhead of handling packet-in messages and the communication overhead of flow queries. The first one mainly happens in the first stage of sampling, and the second one happens at the second stage of packet sampling. The communication overhead happens at both stages.

In the first stage, we need to set up a time window long enough to collect general flows which can represent the statistics of the traffic during that period. In our practice, we found that 5 minutes is a proper time windows for that. However, this time window is much larger than the default timeout of 10 seconds, which means a lot of temp flows would remain in the flow table of the switches. As flow storage is expensive for hardware switches, we choose software switches Openvswitch (OVS) as the replacement. As OVS uses disk as the storage of flows, it is cheap and almost unlimited. However, the software switches are slower than the hardware switches, which is a tradeoff between storage and speed. Besides, as more and more flows are installed, the packet forwarding process would be slower as the flow matching time increases. We will test the performance in next section to evaluate this influence.

In the second stage, after installing static flows that forward packets to controller using packet-in messages, the handling of packet-in messages would increase the cost of the controller. Especially when elephant flows is identified as suspicious in the first stage, there would be too much packet-in messages for the controller to evaluate, and thus the response time would be delayed. However, we can reduce the overhead of the controller by redirect the suspicious packets to a separate DPI system. What’s more, we can control the number of packet-in messages by implementing proper timeout for the installed flows. The timeout is calculated based on the first stage’s analysis of suspicious, the more suspicious a IP is, the timeout is bigger, thus maintain the representativeness of the samples while reduce the packet-in messages.

Both two stages would increase the overhead of communication between the switches and the controller. In the first stage the flow query operations would transmit a lot of flow information, while in the second stage the packet-in messages carry a lot of payloads. Thus, we use a 10Gbit interface to communicate between the controller and switches to ensure the network availability.

### 4. EVALUATION

This section verifies the accuracy and reliability of the proposed scheme through simulation and comparison of the performance with several well-known schemes.

#### 4.1 Experiment Preparation
The experiment layout is shown in Fig. 3. The Host1 is a Dell PowerEdge R730xd server, with one Intel Xeon CPU E5-2603 v3 and 6 cores, and installed with Xen 4.6.0 and OVS 2.5.0. The network cards eth1 is BRCM 10GbE 2P 57810S adapter. The physical switch is 10GbE convergence switch. Each virtual machine is assigned with 1 core CPU and 2G memory. VM1 and VM2 are infected by botnet, VM3 is normal. All of the VMs are installed with Ubuntu 1404 64bit system and tcpreplay for traffic replay. They separately replay the traffic listed in Table 2 to simulate the real-world cloud traffic. The Floodlight controller and P2P detector are installed on standalone physical machines to better test the performance impact. They are both installed with Windows 7 SP1 64bit, and configured with 3.3GHz CPU and 4G memory.

![Fig. 3 Experiment Layout](image)

The background dataset came from a span port mirroring a backbone router at the campus network. We collected all traffic crossing the router for 12 hours. This dataset contains a large number of general traffic from a variety of applications, including web-browsing, email, online-games, P2P file-sharing systems, P2P-TV platforms, etc. Overall, we observed 655 internal IP addresses. We also obtained datasets of two popular P2P botnets, Storm and Waledac, from third parties. The dataset of Storm included 13 individual bots, while the dataset of Waledac included 3 individual bots. Table 2 summarizes brief information of the datasets.

<table>
<thead>
<tr>
<th>VM</th>
<th>Dataset</th>
<th>Duration</th>
<th>Hosts</th>
<th>Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM1</td>
<td>Storm</td>
<td>12 hour</td>
<td>13</td>
<td>27.18M</td>
</tr>
<tr>
<td>VM2</td>
<td>Waledac</td>
<td>12 hour</td>
<td>3</td>
<td>6.51M</td>
</tr>
<tr>
<td>VM3</td>
<td>Background</td>
<td>12 hour</td>
<td>655</td>
<td>1812.29M</td>
</tr>
</tbody>
</table>

Floodlight forwarding module default drop packets which cannot connect and create no flows on OVS, which makes the replayed traffic hard to be captured by our sampling system. To solve that, we use tcprewrite to map the IP addresses into our internal IP addresses and modify tcpreplay to exchange the source IP and destination IP so that outward flows shown to be flows whose destination IP is internal IPs. After that, the
packets sent from internal IP addresses would generate temp flows in the OVS. During the later handling of the captured flows, we need to exchange back the source IP and destination IP.

### 4.2 Suspicious Host Detection

The values of threshold $\theta_{BGP}$ and $\theta_{RCR}$ are crucial during the identification of suspicious hosts. Some true P2P bots may be mistakenly discarded in the phase if these thresholds are set too high. Inversely, if they are set too low, the range of suspicious hosts may be expanded, which may impact the accuracy of the sampling phase. In order to find out their optimal values, we separately set $\theta_{BGP}$ from 90 to 160 and $\theta_{RCR}$ from 5 to 40, and repeatedly test the performance of the Suspicious IPs Detector using the experimental traffic dataset. The corresponding true positive rate (TPR) and false positive rate (FPR) of P2P bot identification are presented in Fig. 4.

As can be seen clearly, smaller thresholds result in higher TPR, but also lead to higher FPR. When $\theta_{BGP} = 100$ and $\theta_{RCR} = 10$, we obtained an acceptable tradeoff between TPR and FPR where the average TPR = 99% and the average FPR = 6.9%. Although the average FPR is relatively high in this phase, it can be further lowered by DPI in the next phase.

![Fig. 4. P2P bot identification results for Suspicious IPs Detectors, over different thresholds.](image)

### 4.3 Sampling Algorithm Evaluation

We evaluate $T$-Sampling algorithm using the experimental dataset with different pre-defined target sampling rates, and compared $T$-Sampling to $B$-Sampling [17], which is a botnet-aware adaptive sampling algorithm. Fig. 5(a) presents the actual sampling rates $SR_{actual}$ for different target sampling rates achieved by $T$-Sampling and $B$-Sampling [18]. $B$-Sampling is a sampling method specially designed to detect P2P botnet. As we can see, both of them always keep the actual sampling rate close to the target sampling rate.

Fig. 5(b) and Fig. 5(c) present the overall sampling rate for packets related to Storm and Waledac, respectively. The results show that $T$-Sampling achieves higher sampling rate for both Storm and Waledac packets, compared to $B$-Sampling. For example, when the target sampling rate is 0.05, $T$-Sampling captures 92.6% of Waledac packets while $B$-Sampling only captures 5% of Waledac packets.
Fig. 5. Sampling performance comparison between T-Sampling and B-Sampling.

Table 3 presents the time comparison between T-Sampling and B-Sampling. B-Sampling needs at least one hour to boot the sampling process, i.e., to obtain the first set of instant sampling rates, while T-Sampling only needs 5 minutes. Moreover, the time window of T-Sampling is much smaller than B-Sampling, which means that T-Sampling is much more adaptive than B-Sampling.

Table 3. Time comparison between T-sampling and B-sampling.

<table>
<thead>
<tr>
<th></th>
<th>T-Sampling</th>
<th>B-Sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot time</td>
<td>5 minutes</td>
<td>60 minutes</td>
</tr>
<tr>
<td>Time window</td>
<td>5 minutes</td>
<td>15 minutes</td>
</tr>
</tbody>
</table>

4.4 P2P Bot Detection (DPI)

After sampling, all of the sampled packets, together with their corresponding instant sampling rates, are delivered to the DPI deployed on Host3 for further accurate detection. We use PeerDigger as our DPI engine to detect P2P botnet, which is derived from our previous work [16]. Other existing network-based P2P botnet detectors can also be plugged into our framework with little modification. The detection TPR and FPR of the DPI for different target sampling rate are presented in Table 4. The last column of the table presents the detection results of the DPI engine using the original experimental dataset without sampling.

As we can see, the DPI still can achieve a high TPR when the target sampling rate is less than 0.1. For example, even when the DPI only focuses on 10% percentage of the traffic volume, it still can detect P2P bots with an average TPR of 98.76%. The reason for this is that our adaptive sampling technique can precisely capture most of the packets...
generated by P2P botnet. Moreover, there is no longer false positive after using traffic sampled by \textit{T-Sampling}. It is probably because that the sampled traffic, which is delivered to the DPI, contains too few legitimate packets to generate any false positive.

Table 4. P2P bot detection results for different target sampling rate.

<table>
<thead>
<tr>
<th>SR</th>
<th>0.01</th>
<th>0.025</th>
<th>0.05</th>
<th>0.075</th>
<th>0.1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPR</td>
<td>26.45%</td>
<td>63.83%</td>
<td>93.08%</td>
<td>96.27%</td>
<td>98.76%</td>
<td>99.65%</td>
</tr>
<tr>
<td>FPR</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>1.2%</td>
</tr>
</tbody>
</table>

4.5 Performance Evaluation

We use iperf [19] to test the impact of our sampling method on the overall throughput of SDN network. We test it every one minute during the traffic replaying. The bandwidths under different conditions are shown in Fig. 6. We can see that when the OVS is connected to SDN controller, the bandwidth gradually decreases as the time in the first five minutes. This is due to the number of flows in the switch increases as the time grow, which increases the matching time for traffic. After the five minutes of the first sampling window, the flows generation and expiring come to a relative balance. So the bandwidth is relatively stable after the first five minutes.

Fig. 6 Bandwidth comparision using iperf

When the SDN controller is not connected, the traffic just directly goes through the switch. When SDN is enabled, the switch must first check if the traffic match any flow entries in the flow table, thus decreases the throughput. The bandwidth degradation caused by SDN is 12% in average.

However, the sampling function actually has little impact on the overall throughput. Compared with SDN network when sampling is disabled, the overall throughput just decreases 2% in average. This is due to the fact that sampling process only queries the switches to get the flows in every 5 minutes. So its impact on bandwidth is small and overall performance cost by SDN and sampling is about 14%.
5. CONCLUSION AND FUTURE WORK

In this paper, we presented a traffic sampling system based on SDN to detect security threats in cloud computing environment. We implemented a prototype to detect P2P bots, which can effectively reduce the volume of traffic that P2P botnet detectors need to process while keeping their detection accuracy, thus allowing them to operate on high-speed and high-volume networks. Our system mainly includes a suspicious host identification algorithm and a two-step sampling rate calculate algorithm. By exploiting the inherent nature of P2P botnet, the suspicious host identification algorithm identifies a small number of suspicious IP addresses that are likely P2P bots as soon as possible. The two-step sampling rate calculate algorithm dynamically tunes the instant sampling rate of every internal IP address to capture as many botnet-related packets as possible while keeping the actual sampling rate close to the target sampling rate. The experimental results show that our system achieves good performance. It is able to identify potential P2P bots in 5 minutes with TPR of 99% and FPR of 6.9% and capture much more botnet-related packets than the state-of-the-art adaptive sampling techniques. By carefully integrate the sampling algorithm to open-source SDN software, the overall performance cost of our system is about 14% and can be reduced by using hardware SDN switches, which means our system can work on-line and response timely when threats are detected. As a modular design, our system can be easily extended to detect other cloud security threats such as DDoS, which will be our future work.

REFERENCES