BotSifter: An SDN-based Online Bot Detection Framework in Data Centers

Zili Zha
George Mason University
zzha@gmu.edu

An Wang
Case Western Reserve University
axw474@case.edu

Yang Guo, Doug Montgomery
NIST
{yang.guo, dougm}@nist.gov

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

***Abstract*—Botnets continue to be one of the most severe security threats plaguing the Internet. Recent years have witnessed the emergence of cloud-hosted botnets along with the increasing popularity of cloud platforms, which attracted not only various applications/services, but also botnets. However, even the latest botnet detection mechanisms (e.g., machine learning based) fail to meet the requirement of accurate and expeditious detection in data centers, because they often demand intensive resources to support traffic monitoring and collection, which is hardly practical considering the traffic volume in data centers. Furthermore, they provide little understanding on different phases of the bot activities, which is essential for identifying the malicious intent of bots in their early stages.**

In this paper, we propose BotSifter, an SDN based scalable, accurate and runtime bot detection framework for data centers. To achieve detection scalability, BotSifter utilizes centralized learning with distributed detection by distributing detection tasks across the network edges in SDN. Furthermore, it employs a variety of novel mechanisms for parallel detection of C&C channels and botnet activities, which greatly enhance the detection robustness. Evaluations demonstrate that BotSifter can achieve highly accurate detection for a large variety of botnet variants with diverse C&C protocols.

I. Introduction

Driven by the great success of cloud computing, the last decade have seen the continuous migration of various services and applications to different cloud platforms. With the flexible pay-per-use model, more and more end users also rely more and more on the cloud platforms for their personal storage and computing need [1]. Under this trend, bots and botnets-as-a-service [2] are no exception: bots and botnets have been one of the most severe Internet threats underpinned by the economic motive, and recent years have witnessed the emergence of cloud-hosted botnets [3] largely due to its cost effectiveness and the long-term availability of the machines in the data centers. Compared to traditional bot machines which get switched on/off frequently by the end-users, data center hosted bot machines usually stay online for longer periods of time [3] and generate more attack traffic (and profit). It was previously reported [4] that cybercriminals have managed to install DDoS botnets in AWS by exploiting a vulnerability in Elasticsearch [5], an open source search engine that is often deployed in cloud environments such as Amazon EC2, Microsoft Azure, Google's Compute Engine. Likewise, botnet command and control software has previously been found to be hosted in Dropbox [6]. Apart from these particular cases, they show that cybercriminals are now breaching commodity data centers, seeking the values of cloud infrastructures to host the botnets.

Cloud-hosted bots (and thus truly botnets-as-a-service) are more harmful than their traditional counterpart, yet accurate and expeditious botnet detection schemes on cloud platforms are not on the horizon yet because of the following challenges. First, since the cloud is a multi-tenant environment, the detection of bots should be fast (e.g., at runtime to prevent further damage) and as non-intrusive as possible so that the detection would have no or trivial impact on the normal data center applications. Second, given the large traffic volume in a data center, such a detection scheme must be scalable, capable of handling tens or hundred gigbit line rate. Third, the detection must be very accurate, since compared to the end user environment, the cost of misclassifying a bot process or connection (and subsequently closing/blocking the connection) in a data center could be much larger or even disastrous.

Some of the early-day bot detection schemes [7] [8] rely on deep packet inspection, which suffer from high resource demands and ineffectiveness against encrypted botnet traffic. Some others [9], [10], [11], [12], [13] focused on host traffic and bot behavior analysis. These schemes became less and less effective since contemporary botnets are constantly evolving to circumvent the advanced detection mechanisms [14], [15], [16], [17], [18]. For example, most botnets abandoned the traditional IRC based Command and Control (C&C) channels and embrace HTTP or P2P for communications.

To deal with such challenges, lately more schemes have been built by taking advantage of the machine learning (ML) techniques for detecting bot activity and/or C&C channels [9], [10], [18], [12], [13], [19]. Such schemes often demand intensive resources to capture the incoming and outgoing traffic information, e.g., a centralized traffic monitoring facility at the network gateway or the firewall, and then run the detection after the traffic features are extracted. These schemes may work for a small network with medium or low traffic volume, but they are hardly effective in detecting the cloud-based bots because (1) they demand a lot of extra resources for effective traffic monitoring, which is hardly scalable considering the huge traffic volume in data centers, (2) their accuracy varies with the used ML model and the chosen features, and they are often incapable of identifying unseen bots without understanding bot invariant characteristics (e.g., C&C channels), and (3) they provide little understanding on different phases of bot

978-1-5386-7117-7/19/$31.00 ©2019 IEEE

activities, which is essential for identifying the malicious intent of bots in their early stages. This is however very desirable for data centers in order to prevent further damage.

More importantly, existing ML-based approaches focus on the detection of botnet flows [12], [13]. However, in the detection of cloud-hosted bots, accurate identification of the bots is more imperative compared to the detection of individual malicious flows. Only after the bots are identified, the infected VMs could be shut down to prevent future attacks. To our best knowledge, no prior works focused on the detection of VM-based bots in the cloud.

To this end, we propose to build BotSifter towards a scalable and accurate runtime bot detection framework in data centers. To be scalable, BotSifter integrates centralized learning (thus to have a global view of the traffic and centralized intelligence) with distributed edge-assisted detection by leveraging software switches in data centers. Due to their widespread deployment in data centers, software switches (e.g., Open vSwitch[1]) are being increasingly employed as monitoring devices as they typically reside in commodity servers with abundant hardware resources. Since they are usually deployed at the edge of the monitored network and located within close proximity to the end hosts, only relatively a small amount of traffic flows traverse the switch, rendering it a more scalable solution for anomaly detection in data centers. Implementing detection at the edge also enables our system to observe both directions of the traffic, which is an essential prerequisite for connection based anomaly detection. Moreover, our edge-based detection framework has the inherent capability of observing the internal attack traffic and C&C traffic within the data center network.

To achieve high accuracy, BotSifter not only conducts neural network (NN) based bot activity detection, but also conducts the detection of C&C channels in parallel. These detections are further enhanced with local and network wide correlations to minimize false alarms. Not only detecting the existence of C&C channels, BotSifter is also able to differentiate different communication protocols (i.e., IRC, HTTP, P2P) utilized by the bots so that custom mitigation mechanisms could be quickly deployed to defend against specific types of bots. Since the majority of the detection operations in BotSifter are conducted within software switches that are instrumented to collect connection features and states on the fly, BotSifter is highly efficient in identifying bots without interrupting other network functions, thus making it a practical solution for the cloud and data center systems. A prototype of BotSifter is implemented, and evaluations based on real-world traces show BotSifter is highly accurate and efficient in detecting bots utilizing known protocols, but also bots with customized protocols.

The highlights of BotSifter lies in

- The design naturally utilizes the SDN's centralized structure to have a global visibility while distributing most of the detecting load to the network edge to be scalable.
- It builds the monitoring capability in the OVS via an off-path traffic collection design to minimize the intrusion of traffic monitoring to normal applications.
- It utilizes neural network to detect bot activities, and employs newly designed protocol specific mechanisms (e.g., self-correlation for P2P) to detect and differentiate different C&C protocols. The local and network wide correlations further enhance the accuracy and robustness of the detection.

The remainder of the paper is as follows. Section II describes the BotSifter design and section III sketches its implementation. The evaluation is presented in section IV. We discuss the related work in section V and make concluding remarks in section VI.

II. BotSifter Design

The design of a cloud botnet detection framework that is capable of monitoring thousands of servers, tens of thousands of Virtual Machines (VMs) running on these servers, and terabit per second communication among these entities and between the data center and the outside Internet is extremely challenging. In this paper, we explore a ML based distributed online cloud bot detection framework, called BotSifter, that monitors and detects the bots within a data center. BotSifter takes advantage of Software Defined Networking (SDN) architecture widely adopted by the data center, and integrates the centralized ML training with distributed monitoring and detection. The ML based bot detector is trained at the central controller, and runs distributedly at the cloud servers with the ML configurations acquired from the central controller. SDN software switches, e.g., OVS, are instrumented with traffic monitoring capability. Both bot activity detector and bot C&C communication detector are implemented at servers using locally collected traffic stats. If necessary, the central detector is invoked to detect data-center wide botnets.

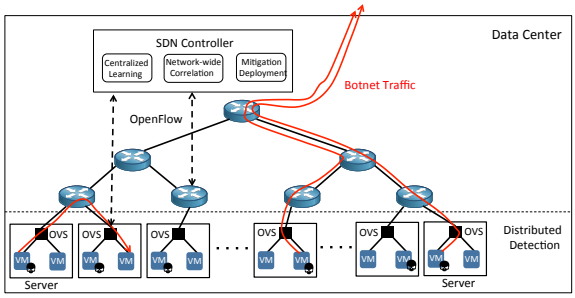

Fig. 1: Overall architecture of BotSifter

The key feature of our design is to leverage the data-center servers and the software switches running on these servers. The contemporary servers have abundant computational and memory resources, and the software switches have the full access to the traffic originated from and destined to the end hosts residing on the servers. Our design philosophy is to

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

place as many monitoring and detection functionality at the edge servers as possible, while judiciously utilize the central controller for data-center wide tasks when necessary.

The BotSifter architecture is depicted in Fig. 1. The central controller is responsible for ML training, network wide bot detection, and bot mitigation. At individual servers, traffic features are extracted and recorded by the instrumented software switches, which are then used by C&C channel detection modules and ML based bot activity detection module. Below we describe local traffic monitoring, C&C bot detection, ML based bot detection, and network wide bot detection and mitigation, respectively. The major components to accomplish these tasks are sketched in Figure 2.

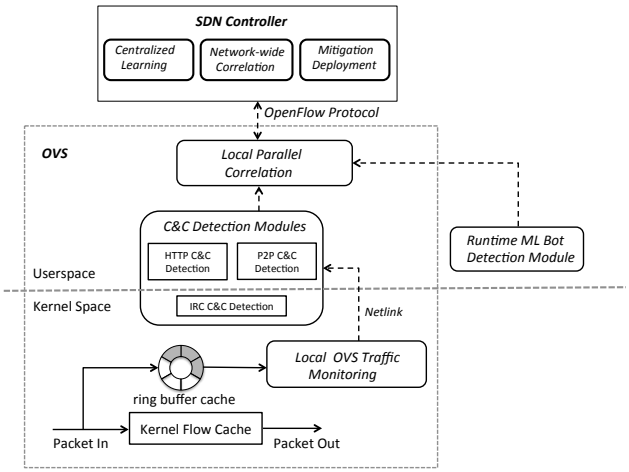

Fig. 2: BotSifter design: major components

A. Local OVS Traffic Monitoring

To lessen the impact on software switch's forwarding speed while efficiently capturing the traffic stats required by ML bot detection module and C&C detection modules is the main design challenge for local OVS based traffic monitoring. OVS maintains a user-space forwarding pipeline and a kernel space forwarding cache. The majority of the incoming packets are forwarded by the kernel module, with few packets that do not have the matches in the kernel being redirected to the user-space. To decouple the monitoring from the forwarding, a ring buffer cache, as shown in the bottom of Fig. 2, is introduced. Such a design significantly reduces the monitoring interference to the forwarding.

ML detection module and different C&C detection modules require different traffic stats. For instance, ML module uses a traffic feature vector, while P2P C&C detection keeps track of DNS transactions for each source/destination pair. Multiple hash tables are employed for efficient lookup and update.

B. Parallel C&C and Bot Activity Detection

At edge server, BotSifter implements one ML based bot detection module and three C&C detection modules: HTTP C&C detection module, P2P C&C detection module, and IRC C&C detection module, respectively. We have the design choice of implementing them inside the software switch OVS, or on the server but outside the OVS. In addition, if a module is placed inside the OVS, we need to decide whether the module resides in the kernel or at user-space. Since P2P and HTTP C&C modules use the connection stats rather than individual packet info, placing them in the user-space at OVS is the right choice. In contrast, IRC based C&C module conducts keyword search over a packet, thus is impleneted in OVS kernel. ML based bot detector uses TensorFlow ML libraries, thus is implemented in the user-space outside OVS. We also place the local parallel correlation module in the OVS, as shown in Fig. 2.

1) C&C Detection: **P2P C&C detection:** Distinguishing P2P traffic used by bots from normal P2P traffic is not trivial. The study in [11] made the discovery that the sets of peers contacted by two different bots within the same botnet typically have a much larger overlap compared to the peer sets contacted by two legitimate P2P clients within the same network. In practice, there is the possibility that only one bot resides in the data center.

We develop a client self-correlation approach to detect P2P bots. Specifically, the peer set of a P2P bot is more likely to remain stable over the time, while that of a normal P2P client is more likely to change due to the changing user behavior (e.g., downloading of disparate resources over time). We conduct extensive empirical experiments, and the results show that P2P bots do exhibit strong self-overlap patterns while normal P2P hosts do not. The new approach is more flexible and robust than the approach in [11].

HTTP C&C detection: Bots tend to communicate with the server periodically over the HTTP channel [20]. HTTP C&C detection module makes the detection using such periodic traffic pattern. The timing information of HTTP connections between a pair of end hosts is collected by the monitoring module. The number of HTTP connections within each time slot is counted. The time series of connection counts is fed into a Discrete Fourier Transform (DFT). If periodic pattern is discovered, the host becomes a bot candidate.

IRC based C&C detection. Although IRC based botnets are diminishing nowadays, we include the IRC C&C detection for the sake of completeness. To minimize system overhead, a combination of keyword matching and port numbers is leveraged to identify IRC connections. Furthermore, to determine whether they are IRC C&C channels, our detection relies on inspection of packet payloads by searching for attack relevant commands in IRC response messages. We implement the detection module at the kernel space of OVS.

Note that our system design is extensible and new C&C detection schemes can be easily integrated. In addition, the design enables network wide correlation for agile C&C detection. For example, a P2P C&C host may not exhibit sufficient self-overlap. In such a case, flow stats of all P2P hosts inside a network can be exported to the central controller, where the clustering based detection can be performed.

2) Runtime ML based bot detection: In parallel to the C&C channel detection, BotSifter also conducts bot activity detection using a deep learning Neural Network (NN). For

each connection, NN is able to detect if it is potentially a bot activity connection.

In order to detect if a host is compromised and becomes a bot, BotSifter keeps track of the percentage of connections initiated by this host that are identified by the NN as the bot activity connections. If the percentage surpasses a preset threshold, the host is identified as a bot.

3) Local parallel correlation: While C&C based detection detects bots via monitoring C&C communication, ML based detection detects the bots via monitoring bot activity traffic. BotSifter introduces a local detection correlation module that combines the results from these two types of detection modules. The consistent detection by both C&C modules and ML based module is a strong indication that a host has been compromised and will be placed on a blacklist. On the other hand, a single positive identification may not be conclusive. For example, the P2P C&C detection module detects that a host may potentially be a bot. However, the bot may still be dormant and have not launched any attacks yet. As another example, if a host is identified by the online ML module as an attacker, it is not clear if the identified host is a bot with a customized C&C protocol that is not recognized by our C&C detection modules, or an ordinary attacker without any C&C channels. In any case, such bots are placed on a so-called local grey list and will be under persistent monitoring/scrutiny. The blacklists and grey lists are sent to the central controller for network-wide correlation and identification.

C. Network-wide Correlation and Mitigation

While local detection is beneficial for the system scalability, the lack of global view can deter the bot detection. Hence BotSifter introduces a centralized correlation module that examines the blacklists and greylists received from the edge servers, and performs network-wide correlation to detect bots. In addition, a mitigation module is implemented to mitigate bots' damages. For each bot on the blacklist, the controller installs flow rules into the OVS to block C&C traffic and/or attacking traffic.

For hosts on a local C&C greylist, the controller performs network-wide grouping analysis to determine if it is a bot. Each group consists of the hosts on blacklists or C&C greylists communicating with the same destination host. If any host in the same group has already been positively identified as a bot, the greylist hosts in the same group are marked as a bot and will be placed on the blacklist. Otherwise, the controller places the host on the global C&C greylist, and continues to monitor the host until a timeout occurs. For hosts on the local ML greylist, the controller looks up the global C&C greylist to check if there is a match. If so, this host is included in the global blacklist. Otherwise, it is placed onto the global ML greylist and continues to be monitored for future signs of C&C communication.

III. BotSifter Implementation

We implement a BotSifter prototype following the design as laid out in the Section II. Fig. 3 highlights the major parts implemented for BotSifter. More implementation details are described as below.

A. Local OVS Traffic Collection Implementation

The local traffic monitoring function is implemented in a kernel thread called *kernel_collector*. We also modify the kernel forwarding thread so that the packet headers are pushed into the ring buffer cache upon arrival. The *kernel_collector* thread takes the packet headers off the ring buffer, and process them to generate the required stats that are stored in hash tables. The hash tables with connection stats are periodically pulled by the user-level stats collection thread *stats_collector*, as shown in Fig. 3.

A more involved task is to detect P2P connections required by the P2P C&C detection module. To identify a P2P connection, the kernel thread intercepts the DNS requests, parses them, and records the hosts who have conducted look-up for a specific IP address. For each new connection between any two hosts *src_ip* and *dst_ip*, we check whether the *src_ip* host has conducted a DNS lookup for the destination *dst_ip*. If yes, it is a normal connection. If not, this new connection is marked as a P2P connection, which will be further examined by the P2P C&C detection module.

B. Parallel C&C and Bot Activity Detection Module Implementation

Three threads, *irc_c&c_detector*, *p2p_c&c_detector* and *http_c&c_detector* are implemented to realize the IRC C&C detection, P2P C&C detection, and HTTP C&C detection, respectively. Thread *irc_c&c_detector* runs in the kernel, as discussed in the Section II, while *p2p_c&c_detector* and *http_c&c_detector* run in the user space, as in Fig. 3.

To identify IRC connections, *irc_c&c_detector* performs lightweight payload inspection against connections with standard IRC ports (6660-6669). The first few packets of an IRC connection typically contain certain keywords, such as "JOIN","USER", "NICK" and "PRIVMSG". To further identify IRC C&C channels, *irc_c&c_detector* examines IRC messages by searching for attack relevant commands (e.g., "scan", "flood"). If such commands are recognized, the *irc_flag* for the connections will be set and exported to userspace.

As mentioned in Section II, the differentiation of P2P C&C traffic and normal P2P traffic relies on self-correlation behavior of peer sets. For the self-correlation, we conduct overlap analysis over multiple time windows of each P2P connection. Time window is a fixed length period of time during the connection. For each time window, we calculate the per-host peer sets in the current and N subsequent time windows. Then, we calculate the number of re-appearing peers in both the current time window and the $i^{th}(i \leq N)$ time window. We use the average of the N overlap values to represent how the peer sets evolve over time. For runtime detection, we calculate the moving average overlaps to differentiate the P2P bots from normal hosts. The used parameters are discussed in Section IV.

Thread *online_ml_detector* implements a NN based bot detection module. The NN model is implemented using TensorFlow 1.8.0 [21], an open source NN library. Since the thread

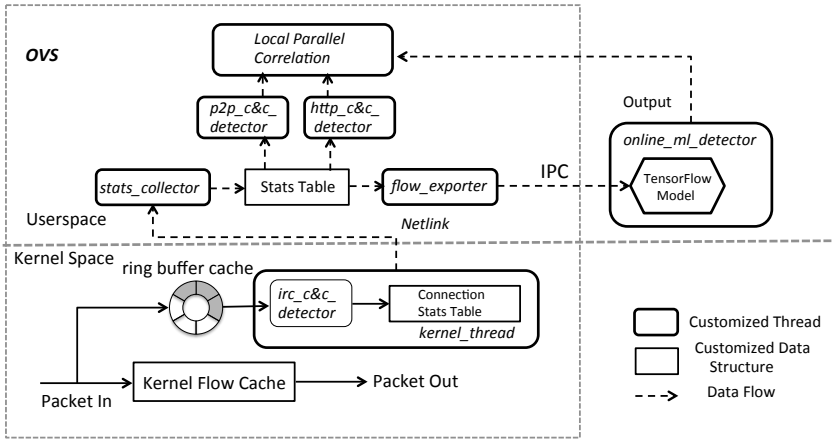

Fig. 3: BotSifter system implementation

runs outside of the OVS, a user-space thread inside OVS, called *flow_exporter*, is created to allow *online_ml_detector* to retrieve the connection [2] feature vectors collected by OVS using Linux IPC calls, as depicted in Fig. 3.

Table I summarizes the features used in our NN model. Notably, different from previous flow based ML models, *src/dst* IP addresses are excluded from our feature set. We believe they are unique to particular data centers and thus should be excluded. More importantly, the exclusion can avoid over-fitting the NN model. In general, choosing the right feature set, or feature engineering, is a challenging research task. We experimented with different features and choose the most discriminative ones based on the experiment results. For example, the forward/backward average packet size of normal flows tends to be larger than botnet traffic since they contain realistic payloads. Meanwhile, we introduce a novel feature, i.e., *conn_stat* to capture whether the connection has been successfully established. Since bots usually maintain a large number of half open connections, this feature is effective in identifying malicious botnet traffic such as direct DoS flooding attack and the amplification attack traffic.

The *online_ml_detector* thread fetches NN configuration parameters from the controller, and initiates the NN model. The *flow-exporter* thread in OVS continuously exports connection features to the NN model, as shown in Fig. 3. To facilitate communication between *flow_exporter* and *online_ml_detector*, a shared memory pool is created. Linux semaphores are used to synchronize access to the shared memory.

C. Network-wide Correlation and Mitigation Implementation

Bot detection applications are programmed at the central controller to perform network-wide correlation analysis. Once a bot is detected, the controller installs customized flow rules into the corresponding OVS switch to prevent future attacks. To facilitate the communication between the threads at servers and controller, traditional OpenFlow protocol is extended to facilitate the collection of detection results from edge servers.

[2]A UDP "virtual" connection is considered successful if at least one packet is received from the destination host.

TABLE I: Description of connection features.

Feature	Description
Source Port	Source port.
Destination Port	Destination port.
Protocol	IP protocol.
Forward packet count	Total number of packets in the forward direction.
Backward packet count	Total number of packets in the backward direction.
Forward byte count	Total number of bytes in the forward direction.
Backward byte count	Total number of bytes in the backward direction.
Duration	Connection duration.
Forward packet rate	Packet rate in the forward direction.
Backward packet rate	Packet rate in the backward direction.
Connection State	whether the connection is successfully established.
Forward inter-arrival time	Packet inter-arrival time in the forward direction .
Backward inter-arrival time	Packet inter-arrival time in the backward direction.
Forward average packet size	Average packet size in the forward direction.
Backward average packet size	Average packet size in the backward direction.
Forward byte rate	Byte rate in the forward direction.
Backward byte rate	Byte rate in the backward direction.

IV. Evaluation

Our testbed consists of three Lenovo ThinkServer machines running Ubuntu 14.04. Each machine is equipped with Intel Xeon 4-Core 3.20GHz CPU and 4GB RAM. One machine is installed with Open vSwitch 2.3.90. Another machine runs the Ryu SDN controller. A third machine serves as both packet generator and data sink, which is connected to the OVS machine via two 10Gbps Ethernet cables. Packet traces are replayed using TCPReplay at the original speed.

A. Datasets

For our experiments, we are able to obtain several third-party traces, including a variety of botnet traces composed of different botnet types and normal network traces. CTU-13 dataset [22] contains traffic of botnet samples captured in the CTU University, Czech Republic, in 2011. In each botnet sample, bots use specific protocols for C&C communication and perform diverse malicious tasks, including SPAM, port scanning, DDoS, click fraud, etc. Due to the diversity of botnet samples, this dataset is appropriate for evaluating the performance of our framework. For the normal traces, we use the UNB ISCX IDS 2012 dataset [23] containing normal traces with full packet payloads. This dataset captures typical user daily activities (e.g., HTTP, DNS, SSH, FTP) at UNB university and was made public for scientific research.

Our runtime bot activity detection relies on a pre-trained model. Table II summarizes the composition of the botnets in

Datasets	*Underlying C&C Protocol*			
	HTTP	***P2P***	***IRC***	***Custom***
Training Dataset	Neris, Virut	Storm, Waledac	Rbot	-
Testing Dataset	Neris, Virut, Sogou	Storm, Waledac, NSIS	Rbot	Menti

TABLE II: Description of botnet datasets for training and testing.

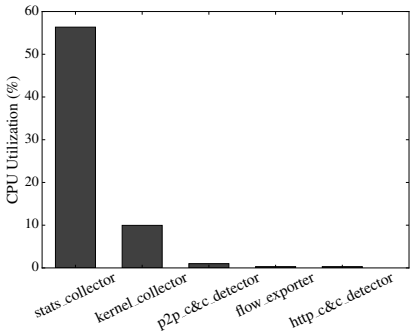

Fig. 4: CPU utilization of detection related threads.

the training and testing datasets. For the training dataset, we merged several traces, including P2P botnets, IRC Botnets and HTTP botnets. For runtime detection, we include not only the same types of botnets as the training dataset, but also novel botnet variants, such as NSIS (P2P C&C), Menti (Proprietary C&C) and Sogou (HTTP C&C). Among them, Menti uses a custom protocol for C&C communication. By evaluating how BotSifter performs when facing novel botnet variants, this experimental strategy aims to demonstrate the effectiveness of our design in real-world bot detection.

B. Impact on Normal Applications

Since the botnet detection accuracy of our system strongly relies on the accurate timing information of the packet sequences, the botnet traces are replayed at the original speed. To evaluate the overall system performance, we perform a stress test using a CAIDA trace and replay the trace towards our monitoring system at the highest achievable rate.

Considering that no modification is made for native OVS threads (e.g., handlers and revalidators), their CPU usage is not shown here. We only report the CPU utilizations of the customized threads in Fig. 4, which shows the peak CPU usage of each detection-related thread under various detection tasks.

We can see that the *stats_collector* in the userspace incurs the highest CPU utilization while the threads regarding C&C detection and flow exportation to the online ML detector introduce negligible overhead. Since *stats_collector* mainly manages the collection of connection stats from the kernel space through Netlink and the maintenance of the connection hash table in the userspace, we infer that its CPU utilization is mainly attributed to the Netlink communications. Further optimization is possible by utilizing shared memory between the user and kernel space instead of relying on Netlink sockets.

In realistic scenarios, these overheads are acceptable as OVS typically resides in commodity servers with abundant CPU resources. The detector threads could be pinned to different CPUs to avoid interfering with CPUs dedicated to the forwarding functions in OVS.

To estimate the network overhead, we use DPDK based packet generator MoonGen to generate high speed traffic and measure the maximally achievable throughput such that no packet loss occurs on the forwarding path. The experiment is repeated 10 times. Compared to the throughput of the native OVS (1.44Mpps), BotSifter could achieve 1.15Mpps throughput. This overhead is acceptable and further analysis shows this overhead is mainly incurred by the *memcpy* operations on the ring buffer cache.

C. Detection Performance

1) Evaluation metrics: To evaluate the performance of Botsifter, we use three metrics. *C&C Accuracy* is calculated as the ratio between the number of hosts which have triggered alarms of C&C detectors and the total number of hosts. *ML Accuracy* represents the detection rate of the bots based on suspicious ratios predicted by the runtime NN model. Since our final detection result relies on a parallel correlation between C&C detection and runtime ML detection, we define a novel metric *Local Parallel Accuracy* to represent the overall accuracy.

As discussed in Section II, the hosts triggering alarms from both *http_c&c_detector* and *irc_c&c_detector* will be put onto a greylist instead of blacklist since they need further network-wide correlation. In contrast, alarms from *p2p_c&c_detector* indicate the associated hosts must be bots and thus they are included in a blacklist. Therefore, we claim that if the host appears on either the C&C blacklist or the ML greylist (e.g., is an attacker, but needs further monitoring to find out C&C channels), it is regarded as a true positive, which also implies that it has been successfully detected by BotSifter. This design principle further demonstrates the robustness of BotSifter compared to methods based on single stage detection.

2) Detection Accuracy and Latency: The measurement results are shown in the C&C accuracy column in Table III. Note that, in ML Accuracy, the percentage values in the brackets represent the suspicious ratios for the bots in the current trace. The parallel detection scheme in BotSifter successfully detects all bots with zero false positives, although some bots trigger alerts from only one stage (either C&C or NN model), such as NSIS and Menti. In the normal trace, there are no false positives due to the following two reasons. First, the hosts are included in HTTP greylist instead of the blacklist, due to the periodicity of software updates. Besides, the suspicious ratios of all hosts are far below the threshold. In a nutshell, BotSifter can detect all bots which may get missed by any single stage of detection. Since local detection accomplishes the tasks, no computation is needed from the central controller, which demonstrates that BotSifter is a distributed scalable solution. In the following, we analyze the results in more detail.

TABLE III: Detection results for 8 botnet variants and normal trace.

Botnet Variant	C&C Protocol	#Bots or #Hosts	Duration	P2P Overlap	HTTP Periodicity	IRC C&C	C&C Accuracy	ML Detection Accuracy	Local Parallel Accuracy
Neris	*HTTP*	1	4.8*h*	-	*Strong*	-	1/1	1/1(99.2%)	1/1
Virut	*HTTP*	1	16.36*h*	-	*Strong*	-	1/1	1/1(99.7%)	1/1
Sogou	*HTTP*	1	0.38*h*	-	*Weak*	-	1/1	1/1(95.5%)	1/1
Storm	*P2P*	13	3.1*h*	*Yes*	-	-	13/13	13/13(90.1%)	13/13
Waledac	*P2P*	3	3.45*h*	*Yes*	-	-	3/3	0/3(50.5%)	3/3
NSIS	*P2P*	3	1.21*h*	-	-	-	0/3	1/1(93.7%)	3/3
Rbot	*IRC*	1	5*h*	-	-	Yes	1/1	1/1(99%)	1/1
Menti	*Custom*	1	2.18*h*	-	-	-	0/1	1/1(99.4%)	1/1
Normal	*n/a*	36	10*h*	-	*Yes(4/36)*	-	4/36 on greylist	FPR:0/36(See Fig. 6)	FPR: 0/36

HTTP C&C detection. Based on our empirical experiments, we find that the C&C connections of certain HTTP botnets may not exhibit periodicity patterns as strong as other botnets. This may be due to unknown factors such as network delay and congestion which introduce noises to the connection timing. We use *strong* and *weak* to represent whether strong periodicity is observed in each botnet trace. Among the used HTTP botnet samples, Neris and Virut exhibit strong periodicity since the bots connect to the HTTP C&C server at regular intervals with negligible timing variations. Since periodicity patterns are observed for each 3-tuple *(src_ip, dst_ip, dst_port)*, the suspicious client and server associated with the 3-tuple could be accurately pinpointed and placed onto a greylist for further examination. Aside from this, we also observed that certain botnets contain more than one HTTP C&C channels, some of which exhibit no periodical pattern. These C&C communications are not typical and cannot represent the botnet C&C behaviors. For those C&C channels exhibiting periodical connection patterns, our online *http_c&c_detector* can accurately identify the C&C channels.

Apart from accuracy, the detection latency is dominated by the prevalence of bots' activities. As previously discussed, HTTP C&C detection relies on disclosing the periodicity inherent in bot C&C communication. Therefore, the latency for identifying the C&C channels is closely correlated with the inter-connection duration. In our experiment, BotSifter can detect periodic C&C communication after 4∼5 connections on average. The actual latency depends on how frequent bots connect to C&C servers. For example, in one of our Neris botnets, the bot periodically communicates with the C&C server on a minute basis, in which BotSifter can detect this channel after around 4∼5 minutes since the first connection.

During the test for the normal trace, the *http_c&c_detector* triggers an alert unexpectedly, which implies that periodic http connections are observed between two normal hosts. Further examination reveals that those hosts are running legitimate applications with periodic HTTP connections.

P2P C&C detection. The detection results for the P2P botnets are also quite promising. Storm and Waledac both show strong self-overlap in their C&C communication. One exceptional case is NSIS, for which no P2P overlap is observed. This is reasonable as the trace is reported to be incomplete with only one C&C channel that is insufficient for analysis. Below we demonstrate that our scheme allows to accurately distinguish P2P C&C from normal P2P traffic.

TABLE IV: Description of P2P traces.

Trace	*Normal*			*Botnet*	
	Emule	*FrostWire*	*uTorrent*	*Storm*	*Waledac*
Number of Packets	3.6M	1.35M	6.5M	6M	2.5M
Duration	3.25*h*	10*h*	8*h*	3.16*h*	8.23*h*

As previously discussed, the key characteristic differentiating P2P C&C from normal P2P traffic is that the peer sets of a P2P bot have large overlaps across consecutive time windows; while the overlaps for a normal P2P host are noticeably smaller. To verify this, we use several third-party traces [24] of normal P2P applications and P2P botnets, as summarized in Table IV. The Storm and Waledec contain 13 and 3 P2P bots respectively. Our experiments show that the *p2p_c&c_detector* could successfully detect all bots.

In our experiment, the average overlap values are calculated across 5 consecutive time windows with respect to the current time window. The length of the time window is set to be 10 minutes. We have tested with various window parameters and acquired similar detection results. However, choosing a relatively small window and short window sequence results in more prompt detection. Otherwise, it causes longer detection delay if more subsequent time windows are used to calculate the average overlaps. In our parameter setting, the P2P C&C channels can be detected within 5 consecutive time windows. To demonstrate the effectiveness of the self-correlation scheme, the moving average overlaps for 10 initial time windows for each P2P host in the trace are shown in Figure 5. For each trace, the result for only one P2P host is depicted in the figure. Other hosts in the same trace all demonstrate similar patterns as the one reported here.

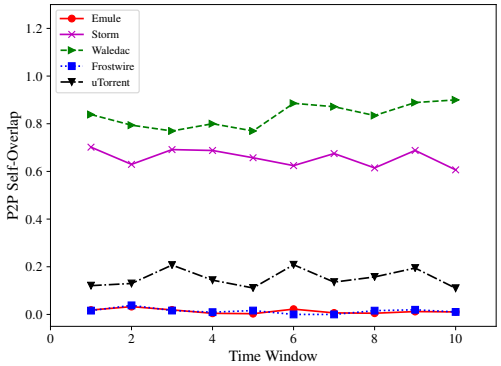

Fig. 5: Peer overlaps for botnet/normal P2P applications.

From Figure 5, there is significant difference between the

peer overlap for P2P bots and normal P2P hosts. In real world detection, a simple threshold based measure (e.g., 0.6) would suffice to distinguish between them. Based on the reported overlaps, our *p2p_c&c_detector* manages to identify all P2P bots in both traces. This demonstrates the effectiveness of the self-correlation scheme for detecting P2P C&C.

IRC C&C detection. In the Rbot botnet, the bot communicates with the C&C server through an established IRC channel. Via this channel the server commands the bot to perform port scanning against certain IPs in the network and the bot continuously reports scanning results to the C&C server. Our *irc_c&c_detector* can accurately detect the IRC C&C channel using keyword matching with negligible delay since the detection is performed entirely in kernel space.

Different from other botnets, Menti adopts a custom protocol and performs port scans. Since its C&C communication does not rely on the three well-known C&C protocols, *c&c_detector* discovers no suspicious C&C pattern. However, our experiment shows that *online_ml_detector* raises an alert since suspicious activities are discovered in the trace, which will be further explained in the following analysis.

Runtime ML based bot detection. As shown in Table III, the suspicious ratios for the majority bots fall between 95% to 99%. For the remaining bots, the ratios still exceed 90%. The result for Waledac is relatively low since the majority of its traffic is C&C related. Since the goal of ML detection module is to detect suspicious bots instead of individual connections, *online_ml_detector* raises alerts based on the suspicious ratio for each individual host. As defined in Section II, it represents the ratio of the number of suspicious connections with respect to the total number of connections for each host. If the ratio exceeds a pre-specified threshold, the host will be included in a greylist. Obviously, the choice of this threshold has a direct impact on the detection accuracy. A higher threshold leads to more false negatives while a lower threshold incurs more false positives. With a threshold of 10%, in the normal trace, 4 out of 36 are false positives. Instead, with a higher threshold, for example, 85%, all bots are accurately identified, with zero false positives. By excluding hosts with a limited number of connections, the ratios of all other hosts are shown in Figure 6.

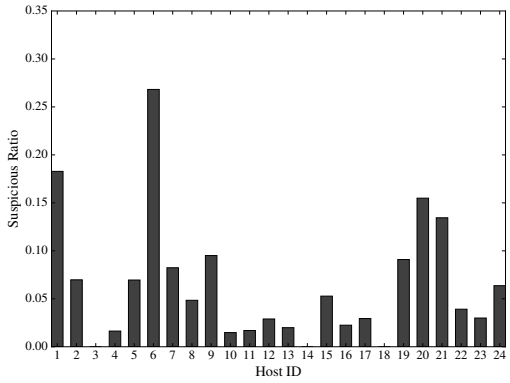

Fig. 6: Suspicious ratios for hosts in the normal trace.

For novel botnets, such as Menti, no C&C patterns are discovered since it uses a custom C&C protocol. However, as shown in Table III the *online_ml_detector* reports high suspicious ratio for the bot in this trace and raises alarms for further correlation. This parallel design is extremely effective in detecting bots in real-world, since it increases the difficulty of novel botnet variants evading both stages of detection.

Performance comparisons. We compare our approach to previous works by evaluating them using the same datasets in our experiment. Since our detection relies on parallel correlation of C&C detection and bot activity detection, the comparison is two-fold. First, we compare our ML detection accuracy to a recent ML based approach [13]. Different from ours, their ML model is based on per-flow feature vectors and only achieves an overall accuracy of 75% on a testing dataset containing multiple novel botnet variants not embraced in the training dataset. By contrast, in our evaluation for Menti (a variant with a custom C&C protocol), the ML model reports a fairly high accuracy ($\sim$99.4%). Indeed our detection method achieves satisfactory accuracy for a majority of the test traces.

Moreover, another similar work [11] focuses on the detection of stealthy P2P botnets through cross-bot overlap analysis, which shows that P2P bots could be identified based on cross-bot correlation. Our experiments evaluate the detection accuracy on the same P2P botnets. As shown in Table III, all P2P bots are detected without any false negatives, which indicates that Botsifter can achieve comparable detection performance by solely using single-bot patterns. However, our approach is more robust in the scenario of a small number of bots.

V. Related Work

SDN based detection The emerging SDN techniques offer new opportunities for enhancing network security. The visibility and programmability provided by SDN have often been leveraged to perform various security detection and attack mitigation schemes.

FRESCO [25] represents one of the initial attempts to compose security services for SDN network systems. It is a framework to enable rapid development of OpenFlow-based security applications on the control plane. Shin *et al.* implemented a FRESCO version of the BotMiner [9] to perform clustering and correlations over network traffic to identify bot hosts. Later, Sonchack *et al.* proposed OFX [26], a system that enables practical deployment of security functions within an existing OpenFlow infrastructure. It allows control applications to dynamically load security modules directly into unmodified SDN compatible switches. OFX integrates botnet detection as a sample security application. It loads pre-processing functions into the switches so that the switches could send batch updates containing the processed feature vectors to the controller for further processing. However, neither solution utilizes computational power of the SDN data plane to perform more fine-grained and accurate detection.

Machine learning based detection In the early stage of bot detection, numerous efforts focused on identifying unique behavioral patterns of bots. For this purpose, various techniques are employed, such as matching the IDS dialog [8] and statistical algorithms to detect organized network activities [10].

However, a big challenge faced by these solutions is the prior knowledge of the malicious behaviors. Alternatively, the advancement of machine learning techniques and frameworks facilitate the explorations of the feature space of malicious traffic to address the challenge.

BotFinder [27] creates a model based on statistical features of flows to detect C&C communications. The model is built on a clustering algorithm to capture similar malicious behaviors of flows. Later, Zhao *et al.* proposed a detection method that employs a decision tree based algorithm with feature vectors extracted from the flows using a sliding window technique. In this model, the feature space is expanded to 12 features of flows [12]. Nonetheless, both algorithms are created towards specific botnet families or certain types of botnet, e.g., P2P botnet. Furthermore, most existing approaches are offline since real-time detection with a rich feature set is expensive and may impact network performance significantly.

Targeting the newly emerged cloud hosted bots, BotSifter addresses all these limitations by utilizing distributed detection and centralized learning in SDN, enhanced with a scalable traffic monitoring facility, and thus offering better detection efficiency.

VI. Conclusion

The cloud-based bots pose an imperative threat to the applications and services running on various cloud platforms, yet highly accurate and scalable detection solutions are not available. In this study, we have designed and implemented BotSifter, an SDN based scalable and accurate runtime bot detection framework in data centers. BotSifter utilizes a centralized learning and distributed detection model that is effectively supported by SDN. Furthermore, BotSifter adopts parallel detection of both the botnet C&C communication patterns and the machine learning based attack activities. The evaluations show the effectiveness of BotSifter based on real-world traces.

VII. Acknowledgement

We appreciate the constructive comments from the reviewers. This work is supported by NIST grant 70NANB18H272 and NSF grant CNS-1524462. This work was also supported in part by the Institute for Smart, Secure and Connected Systems at Case Western Reserve University through a grant provided by the Cleveland Foundation.

References

[1] L. Columbus, "83% Of Enterprise Workloads Will Be In The Cloud By 2020," 2018. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/#32e7847e6261

[2] P. McDougall, "Microsoft: Kelihos Ring Sold 'Botnet-As-A-Service'," 2011-09-30. [Online]. Available: https://www.darkreading.com/risk-management/microsoft-kelihos-ring-sold-botnet-as-a-service/d/d-id/1100470?piddl_msgorder=thrd

[3] K. Clark, M. Warnier, and F. M. Brazier, "Botclouds-the future of cloud-based botnets," in *in CLOSER*. Citeseer, 2011.

[4] [Online]. Available: https://securelist.com/elasticsearch-vuln-abuse-on-amazon-cloud-and-more-for-ddos-and-profit/65192/

[5] [Online]. Available: https://www.elastic.co/

[6] B. Butler, "Hackers found controlling malware and botnets from the cloud," 2014-06-26. [Online]. Available: https://www.networkworld.com/article/2369887/cloud-security/hackers-found-controlling-malware-and-botnets-from-the-cloud.html

[7] J. Goebel and T. Holz, "Rishi: Identify bot contaminated hosts by irc nickname evaluation." *HotBots*, vol. 7, pp. 8–8, 2007.

[8] G. Gu, P. A. Porras, V. Yegneswaran, M. W. Fong, and W. Lee, "Bothunter: Detecting malware infection through ids-driven dialog correlation." in *USENIX Security Symposium*, vol. 7, 2007, pp. 1–16.

[9] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection," 2008.

[10] G. Gu, J. Zhang, and W. Lee, "Botsniffer: Detecting botnet command and control channels in network traffic," 2008.

[11] J. Zhang, R. Perdisci, W. Lee, U. Sarfraz, and X. Luo, "Detecting stealthy p2p botnets using statistical traffic fingerprints," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*. IEEE, 2011, pp. 121–132.

[12] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant, "Botnet detection based on traffic behavior analysis and flow intervals," *Computers & Security*, vol. 39, pp. 2–16, 2013.

[13] E. B. Beigi, H. H. Jazi, N. Stakhanova, and A. A. Ghorbani, "Towards effective feature selection in machine learning-based botnet detection approaches," in *Communications and Network Security (CNS), 2014 IEEE Conference on*. IEEE, 2014, pp. 247–255.

[14] T. Holz, M. Steiner, F. Dahl, E. Biersack, F. C. Freiling *et al.*, "Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm." *LEET*, vol. 8, no. 1, pp. 1–9, 2008.

[15] B. B. Kang, E. Chan-Tin, C. P. Lee, J. Tyra, H. J. Kang, C. Nunnery, Z. Wadler, G. Sinclair, N. Hopper, D. Dagon *et al.*, "Towards complete node enumeration in a peer-to-peer botnet," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ACM, 2009, pp. 23–34.

[16] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich, "Analysis of the storm and nugache trojans: P2p is here," *USENIX; login*, vol. 32, no. 6, pp. 18–27, 2007.

[17] G. K. Venkatesh and R. A. Nadarajan, "Http botnet detection using adaptive learning rate multilayer feed-forward neural network," in *IFIP International Workshop on Information Security Theory and Practice*. Springer, 2012, pp. 38–48.

[18] T. Cai and F. Zou, "Detecting http botnet with clustering network traffic," in *Wireless Communications, Networking and Mobile Computing (WiCOM), 2012 8th International Conference on*. IEEE, 2012, pp. 1–7.

[19] L. Carl *et al.*, "Using machine learning technliques to identify botnet traffic," in *Local Computer Networks, Proceedings 2006 31st IEEE Conference on. IEEE*, 2006.

[20] S. Garcıa, "Identifying, modeling and detecting botnet behaviors in the network," *Unpublished doctoral dissertation, Universidad Nacional del Centro de la Provincia de Buenos Aires*, 2014.

[21] [Online]. Available: https://www.tensorflow.org/

[22] S. Garcia, M. Grill, J. Stiborek, and A. Zunino, "An empirical comparison of botnet detection methods," *computers & security*, vol. 45, pp. 100–123, 2014.

[23] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *computers & security*, vol. 31, no. 3, pp. 357–374, 2012.

[24] B. Rahbarinia, R. Perdisci, A. Lanzi, and K. Li, "Peerrush: Mining for unwanted p2p traffic," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 62–82.

[25] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks," in *20th Annual Network & Distributed System Security Symposium*. NDSS, 2013.

[26] J. Sonchack, J. M. Smith, A. J. Aviv, and E. Keller, "Enabling practical software-defined networking security applications with ofx." in *NDSS*, 2016.

[27] F. Tegeler, X. Fu, G. Vigna, and C. Kruegel, "Botfinder: Finding bots in network traffic without deep packet inspection," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012.