Wuhan University
Journal of Natural Sciences

2017, Vol.22 No.2, 103-113

Article ID 1007-1202(2017)02-0103-11
DOI 10.1007/s11859-017-1223-8

BotGuard: Lightweight Real-Time Botnet Detection in Software Defined Networks

☐ **CHEN Jing, CHENG Xi, DU Ruiying, HU Li, WANG Chiheng**

State Key Laboratory of Software Engineering / School of Computer, Wuhan University, Wuhan 430072, Hubei, China

Abstract: The distributed detection of botnets may induce heavy computation and communication costs to network devices. Each device in related scheme only has a regional view of Internet, so it is hard to detect botnet comprehensively. In this paper, we propose a lightweight real-time botnet detection framework called BotGuard, which uses the global landscape and flexible configurability of software defined network (SDN) to identify botnets promptly. SDN, as a new network framework, can make centralized control in botnet detection, but there are still some challenges in such detections. We give a convex lens imaging graph (CLI-graph) to depict the topology characteristics of botnet, which allows SDN controller to locate attacks separately and mitigate the burden of network devices. The theoretical and experimental results prove that our scheme is capable of timely botnet detecting in SDNs with the accuracy higher than 90% and the delay less than 56 ms.

Key words: botnet detection; software defined network; graph theory

CLC number: TP 305

Received date: 2016-10-20

Foundation item: Supported by the National Natural Science Foundation of China (61272451, 61572380)

Biography: CHEN Jing, male, Ph.D., Professor, research direction: network security, cloud security. E-mail: chenjing@whu.edu.cn

0 Introduction

The rapid development of computer communication technology results in a range of risks in networks [1]. Botnet has become one of the most threatening cyber security phenomena in the last decade. Numerous well-known malwares, e.g. Stuxnet [2], and Zeus [3], use botnets' unlimited attack ability to obtain the information of economy, military, and politics. McAfee's threat report [4] predicted that the intensity of botnet attacks increase with the growth of Internet. Thus, botnet detection problem attracts wide spread attention.

Most of existing researches on detecting and defending bot infection in networks focus on host-based anti-virus solutions that depend on fully distributed way [5,6]. Though these distributed detection and defense schemes are more easily to deploy, they are only efficient to detect some compromised nodes but not sensitive to global threats because of the scarcity of complete landscape.

Software defined network (SDN) is a notable network architecture that allows network administrators to manage network services through abstracting higher-level functionality by separating the control plane from the data plane of underlying systems. The controller of SDN owns the global view of the whole network and the flexible configurability. It provides a fast way to access the basic data sources on security problem. We can make use of SDN controller to solve the problems brought by distributed detection. Hence, SDN technology can provide a promising way to deal with security problems

such as detecting botnets.

Up to date, there are only a few related researches about this issue. Based on SDN, Haq *et al* [7] presented NetworkRadar, a centralized prototype to detect botnets for Internet service providers (ISPs) and enterprises. The detection task is completed by cooperation of data plane devices and a controller. However, this scheme is still in the traditional network framework, which has heavy loads while realizing deep packet analysis and configuration on every distributed device. Wijesinghe *et al* [8] proposed a machine learning technique to detect botnet by analyzing the traffic flow information in SDNs. The algorithm of machine learning is time-consuming and inefficient for most of conventional botnet detection schemes. These researches have not discovered a botnet until the attack is accomplished.

Based on the above analysis, there are still two challenges to detect botnets in SDNs.

1) Lightweight Machine learning algorithms may induce heavy loads to network devices. When detecting botnets, to improve overall network performance, a lightweight scheme with low computation and communication costs among network devices is in need.

2) Real-time The related researches about botnet detection in SDNs show that bots cannot be detected in the early stage of botnet's lifecycle. These methods can only conduct some remedial measures, but they are not sensitive to attacks from botnets. Therefore, a real-time detection scheme is an urgent need. The timely detection provides a possibility to hold back bots before the outbreak of massive attacks.

In this paper, we propose a scheme named BotGuard, which is a lightweight real-time botnet detection system in SDNs. A convex lens imaging graph(CLI-graph) is concluded based on the analysis and summary of botnets' topology characteristics. In addition, compared with existing detection methods in SDNs, we design a graph-based detection algorithm with a lower computation cost. These operations decrease the delay of botnet detection and solve the first challenge.

For detailed design, our scheme contains the following three parts: topology collection, flow graph extraction, and detection engine. The main idea of BotGuard is to use SDN controller as a supper brain with ubiquitous eyes to detect botnets. Based on the consequence of data preprocessing, the SDN controller with centralized control and global view accomplishes topology collection and flow graph extraction. The SDN controller extracts a real-time flow graph from a complete network topology and conduct an analysis in time in flow graph extraction. Consequently, we detect botnets and prevent the explosion of large-scale attacks, and the second challenge about real-time detection is solved. The detection engine which includes a graph-based algorithm determines whether there is a botnet.

We focus on the situation that a botnet is detected before its whole lifecycle ends with low computation and communication costs. To the best of our knowledge, this is almost the first work to implement a lightweight real-time botnet detection system in SDNs.

Specifically, our major contributions can be summarized as follows:

1) We propose BotGuard, a real-time lightweight botnet detection framework, which identifies botnets with the real-time flow information in SDNs. Our proposed scheme locates the position of attacks rapidly once a botnet is activated, thus, it can limit threat in a small scale.

2) We introduce a universal botnet topology model named CLI-graph, which extracts the topology characteristics of botnets. According to CLI-graph, the controller of SDN can scan the network and find potential botnets quickly, which alleviates devices' computation and communication costs.

3) We implement BotGuard and evaluate its performance from several aspects. Our evaluation factors include detection availability, detection ratio, and delay. A high detection ratio with acceptable delay is obtained by our scheme.

The rest of this paper is organized as follows. We briefly review the related work in Section 1. In Section 2, we state the problems that our scheme will solve and the detailed design of our scheme. Section 3 gives the experiment analysis and evaluations of our scheme. The final section concludes the paper.

1 Related Work

1.1 Botnet Detection Technology

There are many references on detecting and mitigating botnets [9-11]. Botnet detection technique can be classified into two categories: A) Detecting by signature; B) Detecting by cooperative behaviors.

The schemes of category A are based on some predefined patterns of activity signatures. For training this patterns, BotFinder [12] extracted statistic features (e.g. average time, average duration, average number of bytes)

from the network traffic of known malware samples. These patterns are then applied to identify the network traffic and to detect malware infections. For category B, the cooperative behaviors are the intrinsic property of botnets activities that all devices in a botnet have the same goal. These cooperation behaviors could be detected by various correlative techniques, e.g. active botnet probe [13], BotSniffer [14], BotMiner [15], and BotTrack [16]. BotHunter was introduced by Gu *et al* [6], in which a botnet infection model is proposed to deal with IDS-driven dialog correlation. The application of this scheme was designed to track the communication flows and match a state-based infection sequence model for detection.

However, in these studies, data packets are accessed in distributed way, and each node of botnets needs to independently participate in the detection process at high computation and communication costs. For this distributed design, no one can exactly describe the whole network's situation and other nodes' behaviors. Thus, the comprehensiveness and efficiency of these detection schemes need to be improved. In SDNs, the existence of a centralized controller makes this task much easier.

1.2 SDN Security Issues

The concept of SDN originated from the Ethane [17] project of Stanford University. Scott-Hayward *et al* [18] presented a significant number of issues on SDN, and the security issue has gained more attention. In Ref. [19], the security problems were divided into two classes: the security of SDN platforms and security enhancing schemes with the unique ability of SDN. Xia *et al* [20] put forward that the SDN architecture can be utilized to enhance network security by reactive monitoring, analysis and response. Similarly, SDN makes it easy for an administrator to orchestrate network switches, and a defensive flow management can be performed in that scenario.

Dhawan *et al* [21] presented SPHINX to detect both known and unknown attacks within SDN, which has a dynamical network behavior learning that does not exist in traditional detection methods. The dynamical process provides a beneficial condition of detecting unknown attacks by the advantages of SDN controller.

SDN architecture can help researches deal with many network challenges. The programmable capability will enhance the flexibility of the network. Nevertheless, the methods for botnets detection in SDNs are quite few. Although Haq *et al* [7] and Wijesinghe *et al* [8] discussed botnet detection methods in SDNs, they cannot accomplish detection in time and achieve a better performance with the advantage of SDN controller's global view. In order to fill this gap, we try to design a more effective botnet detection scheme by using SDN. Our framework takes both detection performance and time into account to remedy the defects of other related schemes. Using the centralized capability of SDN controller and topology analysis, botnets detection can be especially easier and gain better performance as described in the following parts.

2 Proposed Scheme: BotGuard

2.1 Scene of Botnets in SDNs

Figure 1 shows an attack model of botnets in SDNs. We only deploy detection scheme in this scene. From Fig.1, it is obvious that the SDN controller can manage the whole network, which operates the flow-entries in the flow tables of SDN through a southbound application program interface(API). An attacker who acts as a botmaster in traditional botnets compromises a host to launch botnet attacks. Then, the botmaster controls many other hosts in the same or different domains. The controlled hosts are called bot-infected devices. The bots who used specific communication patterns are totally controlled by a botmaster. Afterwhile, the attacker expands attacks by using bots in that network. To detect botnets in SDNs, some counter measures are needed under this attack scenario.

For botnets deployment in SDNs, we have two assumptions: 1) A malicious attacker may cheat a host in the given network; 2) The attacker has limited capability, so it ensures that the controller is reliable. Those two assumptions ensure SDN architecture to be reliable in our scheme.

2.2 Scheme Model of BotGuard

To detect botnets in SDNs, we have designed a scheme model named BotGuard, which mainly depends on topology analysis of the SDN controller. The scheme has a data preprocessing. This process is a warm-up part of BotGuard to obtain various network data from OpenFlow switches in data plane. The data are gathered and sent back to the controller for further analysis and processing. In the detection process, OpenFlow messages are used to establish a real-time flow graph in current network topology. Then according to a graph-based detection algorithm, we compare the extracted flow graph with CLI-graph and judge the existence of botnets. This scheme includes four modules corresponding to the

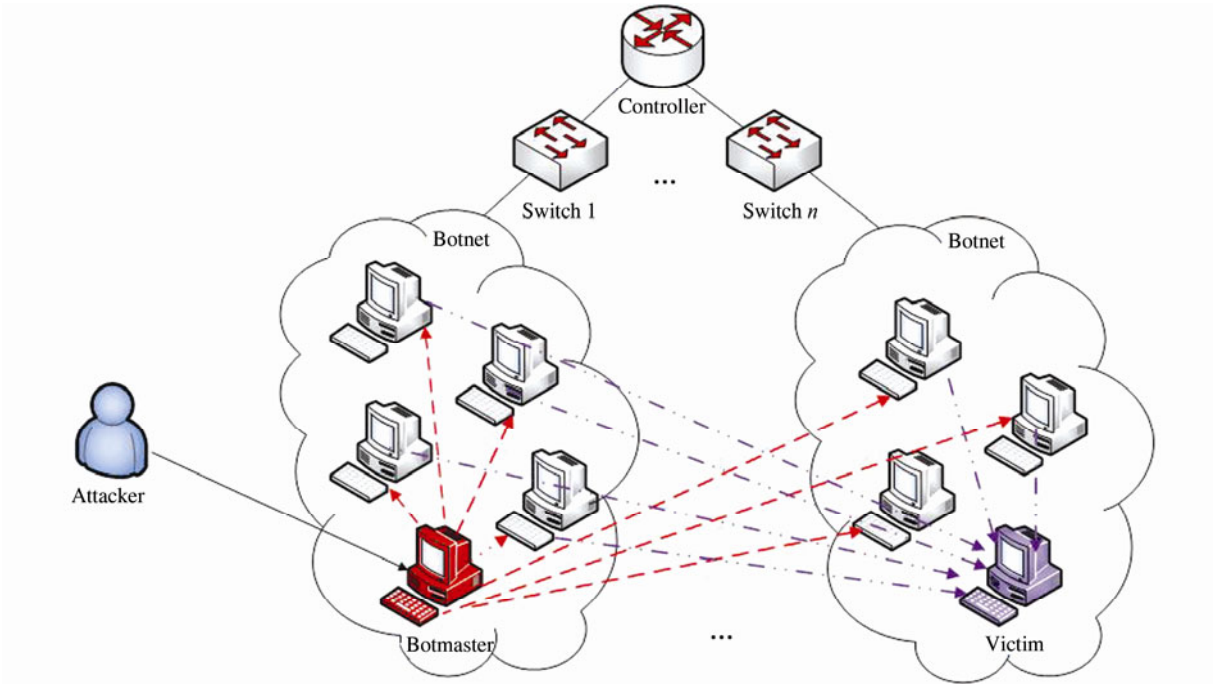

Fig. 1 Botnet architecture in a SDN

The solid line shows the process for infecting a host; The host controls many other hosts as the upward dotted line indicated; The controlled hosts will attack a victim together as the downward dotted line indicated

detection sequence of the POX controller [22], as depicted in Fig. 2.

The following parts mainly introduce the three key detection modules:

● Topology collection This part plays an important role in the whole lifecycle of BotGuard. The SDN controller can collect network topology information easily. The module is shown in the first box of Fig. 2.

● Flow graph extraction To analyze real-time network traffic, a flow graph should be extracted from the collected network topology. The second box in Fig. 2 provides us an example of our flow graph. This process is accomplished by parsing OpenFlow messages within a certain time interval.

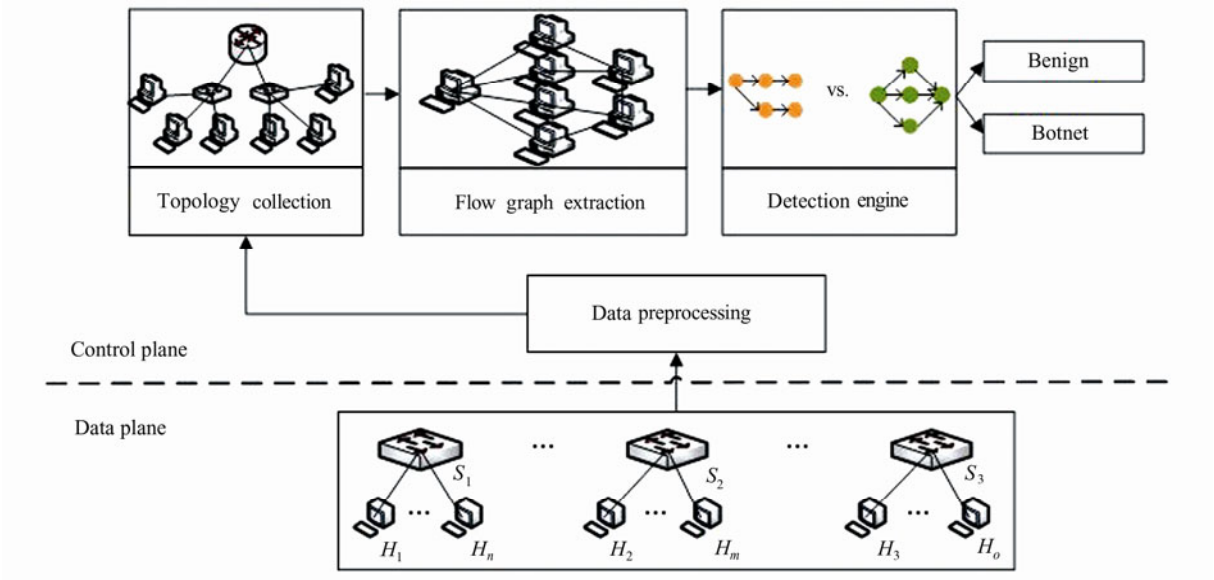

Fig. 2 Overview of BotGuard

● Detection engine This is the key part of BotGuard. After comparing the extracted flow graph model with CLI-graph model in Section 2.4, we can get the detection result (as the final square shown in Fig.2) and take corresponding blocking measures.

2.3 Topology Collection

This section proposes the method of getting topology graph by SDN controller. Topology collection is generally completed by two important modules in POX controller: discovery.py and spanning-tree.py. The whole flow chart of such a process is summarized in Fig. 3. The top part of Fig. 3 gives a brief look at related modules (as the dotted box) and message mechanism (as the middle-big box). The bottom half of Fig. 3 illustrates the topology formation process.

The function modules, discovery.py and spanning-tree.py, realize the function of topology discovery. The principles of these two modules have been proposed in the early work [22]. The core tasks are treated by the functions related to event triggers and event handlers. Once connected to a new switch represented in the leftmost rectangle, this action triggers the ConnectionUp event. As a chain reaction, an instance of class LLDPSender will get all messages about ports from its connected switches. The instance calls function add_ port() for constructing LLDP packet. Then it calls _set_ timer() to invoke function timer_hander() which is in charge of sending LLDP packet.

When a LLDP packet is sent to controller, message PacketIn will be triggered. Consequently, one instance of class Discovery will parse the LLDP packet from switches. The first step for parsing is constructing an object Link. If this connection does not exist in adjacency dictionary, it will be added into the dictionary as a new entry, and LinkEvent is triggered in the meantime. If this connection exists already, it will update the topology graph. This update operation is related to LinkEvent. When a connection is closed or a port is changed, _calc_spanning_tree() will be used to get the updated network topology.

To meet requirements of BotGurad, the final tree structure of network topology is regarded as the result of our topology collection, shown as an example in formula (1). The final tree structure is also an input for flow graph extraction. The relationship between ports and switches is

$$\begin{aligned}&\{\text{switch}_1:([(\text{switch}_2, \text{port}_1), (\text{switch}_3, \text{port}_2)]\cdots)\\&\text{switch}_2:([(\text{switch}_1, \text{port}_1), (\text{switch}_3, \text{port}_1)\cdots])\cdots\}\end{aligned} \quad (1)$$

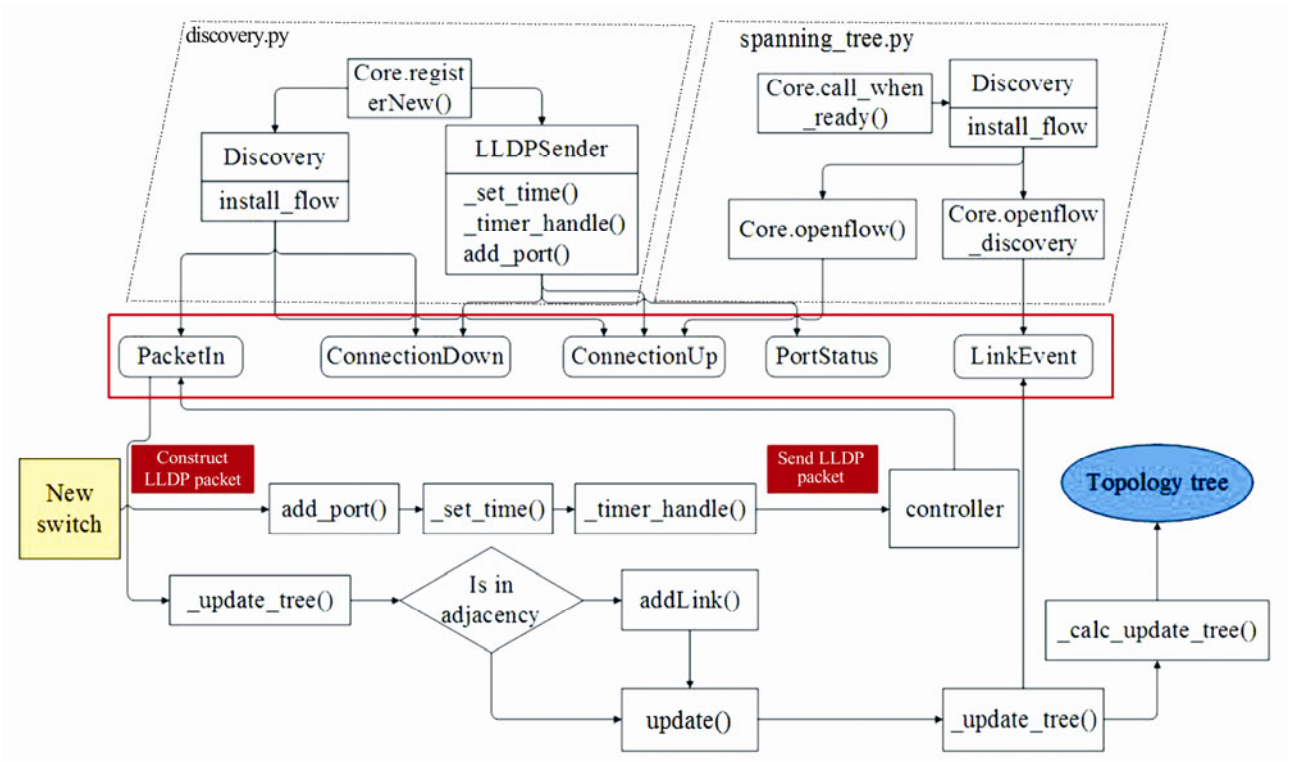

Fig. 3 Topology collection using POX controller

2.4 Flow Graph Extraction

Based on the outcome of topology collection and the global network view of SDN controller, we can extract the flow graph in time. The process of extraction is depicted in Fig. 4. The example illustrates that the flow graph is a real-time connected topology that consists of communicating hosts in given time interval. H_3 is not in the flow graph because it has no communication with other machines during this time interval. We define the detection time internal as the time between the first package obtained and the start of detection in detection engine. It is also an important parameter in latter assessment experiments.

According to the message mechanism in OpenFlow

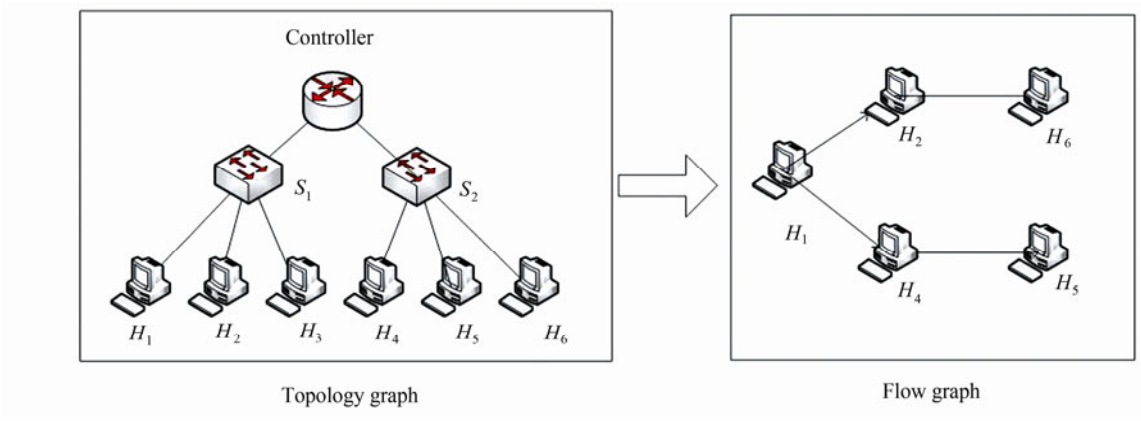

Fig. 4 An example of flow graph extraction

protocol, we take advantage of Packet-in to realize a passive detection. SDN switches send these messages to the controller, and the controller can establish a new link if there is no matched flow table in switches. On the arrival of new packets, controller can acquire flow characteristics about ports and switches, then get the five-tuple of packets to construct the real-time flow graph. Forwarding operations of network devices in SDNs, which go through the above process, can be described as a transfer function T. This function T simulates forwarding operations, consisting of the actions of receiving a packet and forwarding it to the corresponding output ports. These ports are specified by flow table in OpenFlow switches. An instance of function T is listed as formula (2) ; s means a switch while p means a port.

$$T:(\mathrm{s},\mathrm{p})\to\{(\mathrm{s}_1,\mathrm{p}_1)(\mathrm{s}_2,\mathrm{p}_1)\cdots\} \tag{2}$$

For example, the transfer rules $R_{a\to b}$ between host a and host b can be defined as follow, and Γ stands for a set of function T.

$$R_{a\to b}=\bigcup_{a\to b}\{T_n(\Gamma(T_{n-1}(\cdots(\Gamma(T_1(\mathrm{s},\mathrm{p})\cdots)))))\} \tag{3}$$

Thus, the route from host a to host b is a combination of the transition functions, which can be denoted by the following:

$$\{T_1,T_2,T_3,\cdots,T_n\} \tag{4}$$

Then we analyze the five-tuple of network flow from a packet-in message to grasp which host is connected in such a network. Finally, a real-time flow graph is established by iteration. The results of flow graph extraction can be listed as:

$$\{(H_1,H_2)(H_1,H_3)\cdots(H_n,H_m)\} \tag{5}$$

This graph is made up of points (stand for hosts) and edges (stand for communication between hosts). Because the packet-in message is an instant message, we can make real-time detection possible. So it is easy to utilize passive packet-in message gathering data before an attack is accomplished. In our scheme, the time interval of detection is set up to 80 ms, which is a proper detection setup according to relevant experience. The real-time flow graph is important for detection engine.

2.5 Detection Engine

In this part, our scheme tries to compare the real-time flow graph extracted from SDN controller with CLI-graph which is put forward based upon special topology analysis of botnets. A real-time flow graph obtained from flow graph extraction is a crucial source of detecting data needed in this module. The next problem is how to implement this graph matching. In the first step, a benchmark general comparison graph model is built as a template, and a graph-based matching algorithm is designed as the second step.

1) CLI-graph model CLI-graph is a conclusion based on the analysis of basic botnets' topology graph. In

BotGuard, we make full use of CLI-graph to identify botnets. Though the overall architecture and implementation of botnets are complex, such as bot codebase's variety in size, structure, complexity and implementation approach, there is a unified model based on the set of topology. Thus, we pay attention on the topology of botnets.

Reviewing the botnets' framework mentioned above, botnets, networks of malware-infected machines controlled by an attacker, are the root of a large number of internet security problems. Those machines, controlled by an attacker, are called bots. Once an attacker has the ability to manipulate a number of bots, he can use these machines to launch an attack. As a result, the attacker can accomplish an attack by forcing those bots to eavesdrop some victim at the same time.

From the topology graph, we can discover that the graph has two processes: convergence and divergence. In the divergence process, an attacker tries to actively control a lot of bots. And in the convergence process, the attacker forces those controlled bots to attack one target. It is an exceptional case of bipartite graph [23], shown in Fig. 5. In physics, we know that if you place a divergent light source at focal point of one convex lens, you will get some convergence light behind the convex lens. The topological graph of botnets is the same as the convex lens imaging. Then we form the following definition of CLI-graph.

Definition CLI-graph: CLI-graph is a directed graph, which means that edges have orientations in the CLI-graph. It is denoted as an ordered pair $G = (V, A)$ with V (a set whose elements are called vertices) and A (a set of ordered pairs of vertices, called directed edges).

In CLI-graph, the vertex set V can be partitioned into three sets, U, W and X (U, W and X are each independent sets to each other), such that any pair of vertices in every sets share different directed edges. U and X have only one vertex which are just like v_1 and v_n in Fig. 5, but W must have a great number of vertices such as m_1, m_2, ⋯. Vertex set U mapping vertex set W can be seen as a divergence form. While vertex set W mapping vertex set X looks like a convergence form. Figure 5 is a typical drawing of CLI-graph containing three sets: origin point, middle parts and the target. The ellipse in the middle including a large number of vertices looks like a convex lens.

2) Graph-based algorithm CLI-graph is a special example of bipartite graph. Therefore, we utilize some characteristics in maximum matching of bipartite graph for flow graph matching in botnet detection. In graph-based algorithm, how to save the graph structure is a primary problem to be solved. The most common data structures for a graph in network are adjacency list and Star. Comparing the two methods, the Star needs a lot of time when one connection is added or deleted, while the adjacency list is more efficient. We choose the adjacency list to store our topology directed graph in our scheme as a consequence.

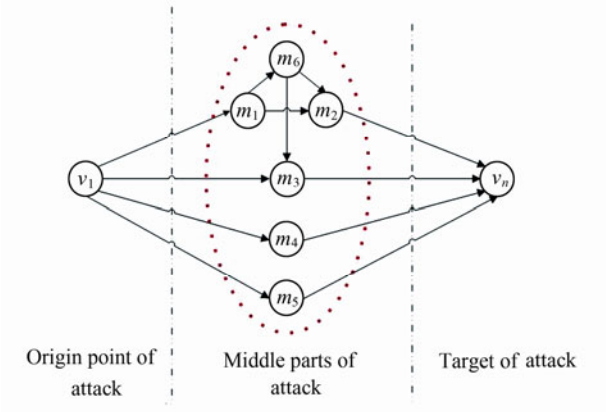

Fig. 5 CLI-graph model

The CLI-graph-based algorithm is set in SDN controller. The characteristics of the CLI-graph are summed as: 1) Origin points only have outgoing edges; 2) Target points only have ingoing edges; 3) A number of points between the origin and the target have different kinds of edges. Matching algorithm is designed by these characteristics with consideration about degrees of vertices. Therefore, inDegree computation algorithm is listed in Algorithm 1 and outDegree computation algorithm is the same as Algorithm 1 with different head pointer as $head_2$.

Algorithm 1 inDegree and outDegree algorithm

Input: AdjGraph $g(v, e)$
Output: The number of each vertex s inDegree: inD
for each $i \in [1, g.\text{vernum}]$ **do**
  inD = 0; $p = g.\text{adj}[i].\text{head}_1$;
  while $p \neq$ NULL **do**
    inD++;
    $p = p \rightarrow$ next;
  end while
  return inD of v_i;
end for

We use breadth-first-search (BFS) algorithm to traverse each vertex in the graph, which is achieved in flow graph extraction. In combination with the degrees of each vertex, if the degrees of vertex meet the characteris-

tics of CLI-graph, we assume that there is a botnet model in our network. Decision process is described in Algorithm 2 for details and constant a may be changed according to the number of machines in the network.

Once botnet topology graph is confirmed, this algorithm will send alert information to the controller. We also establish effective flow rules to block the bots for visualizing the results in our experiments. The way to block bots is the same as that in Ref. [24], which blocked entries for malicious users by flow tables.

Algorithm 2 A graph-based botnet detection algortihm

Input: Flow topology graph $g(v, e)$
Output: Result of detection
  for each $i \in [1, n]$ **do**
    initialize the degree of v_i with calculating of inDegree(v_i)
    and outDegree(v_i), v_i.mark=0; $t_1 = 0$; $t_2 = 0$
  end for
for each $v_i \in g(v, e)$ **do**
  if v_i.mark = 0 **then**
    BFS(v_i)
     if inDegree(v_i)==0 && outDegree(v_i) $\geqslant a$ **then**
       t_1=1
      for each $k \in [1, n]$; $k \neq i$ **do**
       if inDegree(v_k) = outDegree(v_k) =1 **then**
         t_2=1
       else if $\exists j \in [1, n], j \neq i$ and $j \neq k$,
       inDegree(v_j)=0 && outDegree(v_j)$\leqslant a$ && t_1 && t_2 **then**
            return botnet attack confirm and report
       else
          not matched
       end if
      end for
    end if
  end if
end for

2.6 Complexity Analysis

In order to analyze the complexity of the graph based algorithm in this part, we suppose that there are n hosts as vertices with e edges in our SDN topology.

According to the description of Algorithm 2, we choose a host from n hosts and traverse from it by BFS traversal. Then, the inDegree and outDegree are calculated based on the topological structure and connection between/among the hosts.

● The initialization procedure: Computing the degrees of each vertex contributes to a time complexity of $O(n\times e)$;

● The matching procedure: Based on BFS traverse, the matching process consists of two loops in Algorithm 2, and the time complexity of matching operations is $O(n^2)$.

To sum up, time complexity for our graph based matching algorithm is $O(n\times e) + O(n^2)$, the final time complexity of the detection algorithm for CLI-graph matching on random graphs is $O(n^2)$. Because the number of hosts n is no more than 10 000, BotGuard can be seen as a lightweight scheme.

3 Implementation and Experimental Evaluation

As discussed in Ref.[25], the network security community lacks suitable real large-scale botnet attack datasets, which is the same as DDoS attacks. As an emerging network, it is also hard for SDN to find suitable and acknowledged datasets to do comparison with different methods. So our experiments are based on simulation designed in SDNs to demonstrate the performance of our work.

3.1 Implementation

To implement our system, we choose Mininet as SDN simulation environment with POX controller. Mininet [26] is a network emulator that can establish a network of virtual hosts, switches, controllers, and links for SDN. In particular, our BotGuard code runs on real POX platform and uses the OpenFlow protocol to access the data plane. The graph-based algorithm is set up for our controller in advance. We replace the original Open vSwitch in Mininet with our modified one for better message management. The experiments are realized at Ubuntu 12.04, i5 CPU and 8 GB RAM. As shown in Fig. 6, we apply our detection system to a resembled experimental topology. We emulate a botnet attack in such a SDN framework. The host H_1 acts as botmaster while H_n is the victim as an example. For massive attack experiments, we deploy about 550 virtual machines in mininet. The whole process of communication in the attack is made by OpenFlow protocol v1.1.0 [27].

3.2 Experimental Evaluation

In order to validate the correctness of the logic in BotGuard scheme and evaluate its performance, we test the BotGuard code with POX.

1) Detection availability Figure 7 shows the results of the emulation with 150 bots in the network. We measure two entities in our experiment: blocked flow tables and connections to attacked address. The peak of

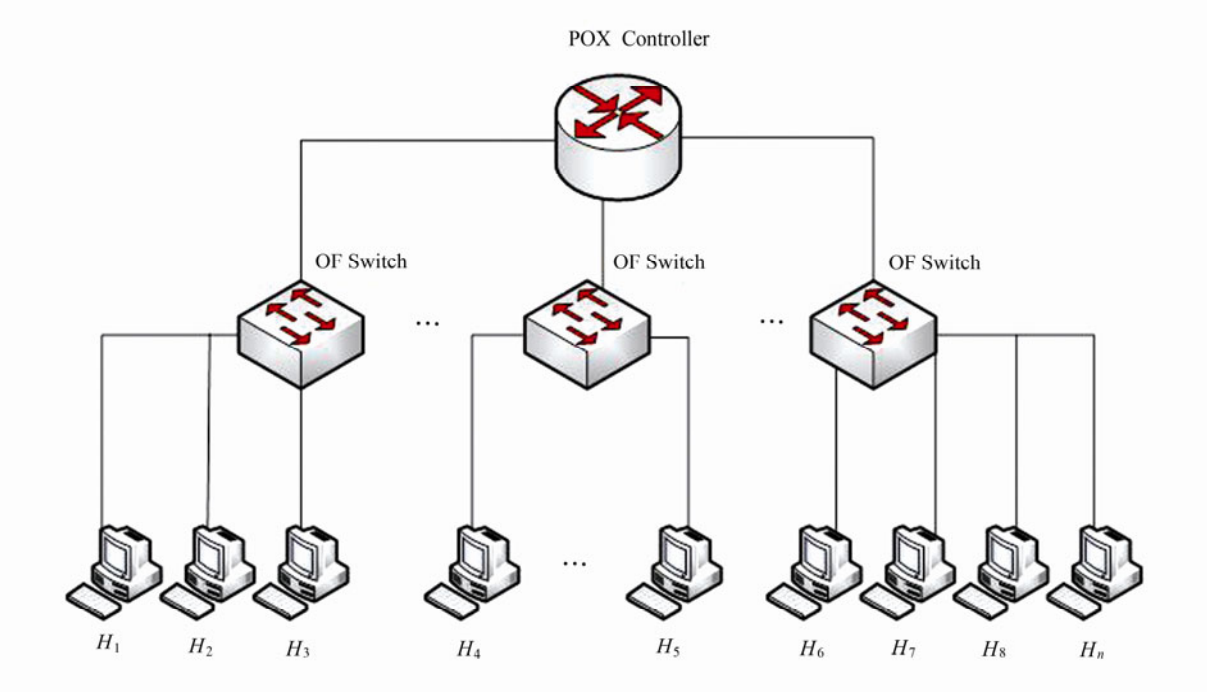

Fig. 6 Approximate topology in test scenarios

curve with triangles is the scenario when machines are bot-infected and ready to commit an attack. Therefore, we can detect botnets before the attack completed. At around t=80 ms, bots begin to be classified and blocked. The connections to the attacked address are reduced to zero because bots are blocked with the growth of blocked flow tables. Almost all bots are blocked by the predefined flow rules at t=128 ms. The experiment demonstrates that BotGuard is available in detecting botnet, and it can prevent attack traffic before the explosion of large-scale botnet attacks. That result gives an answer to our second challenge.

2) Detection comparison The second experiment evaluates scalability through multi-angle vertical comparison by two key performance metrics: the ratio and delay of botnet detection. We vary the time interval of detection Δt in BotGuard, and record the detection ratio and delay along with the increment of the number of bots. The lower false positive ratio and false negative ratio reflect the better performance of BotGuard in detection.

Each line in the chart represents a kind of experimental setup with different Δt such as 40 ms, 80 ms and 120 ms. Figure 8 demonstrates that the false positive ratio is always low in different size of botnets. As shown in Fig. 9, false negative ratio with different bots has a small fluctuation but still within the tolerable range. Δt=80 ms is the best time interval from such two charts, and it is the best time to gain a lower false positive ratio with a lower false negative ratio.

We define detection delay as the time between a controller receiving many messages and the first instant

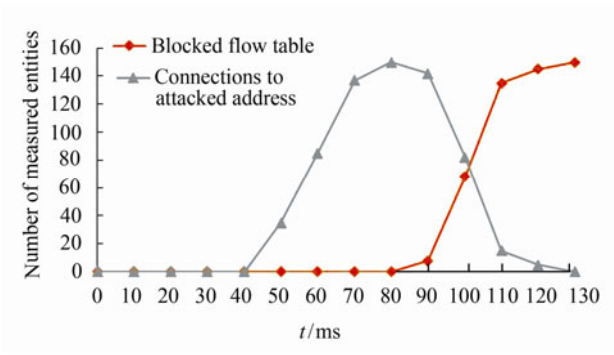

Fig. 7 Bot-blocking dynamics of the proposed system

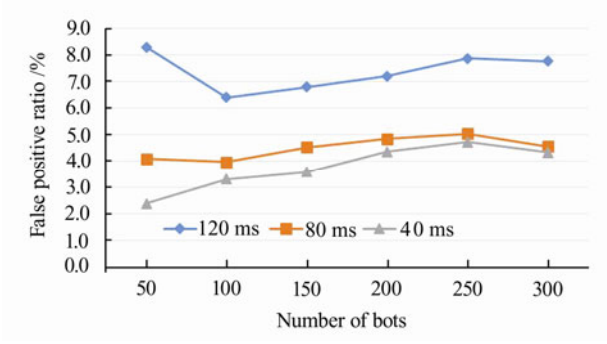

Fig. 8 False positive ratio in different botnets

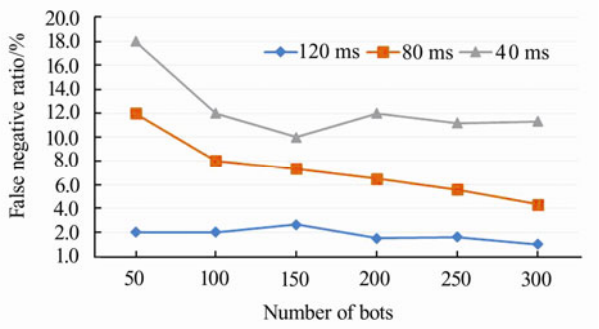

Fig. 9 False negative ratio in different botnets

that one host is declared as a bot. In this experiment, we set Δt = 80 ms as a better experimental conditions. More importantly, in Fig. 10, delay increases with the number of bots: The worst-case delay is about 56 ms which is still a reasonable value in our detection scenarios. Thus the low error detection ratio with low delay together proves that BotGuard can achieve a lightweight real-time detection model in SDNs. The negligible ratio of false negatives may have relationship with losses of Openflow messages. Consequently, we need to strengthen SDN message mechanism to improve our experiments in the future. Overall, our scheme can detect a botnet more quickly before the range of attack expanded, with an acceptable performance and accuracy.

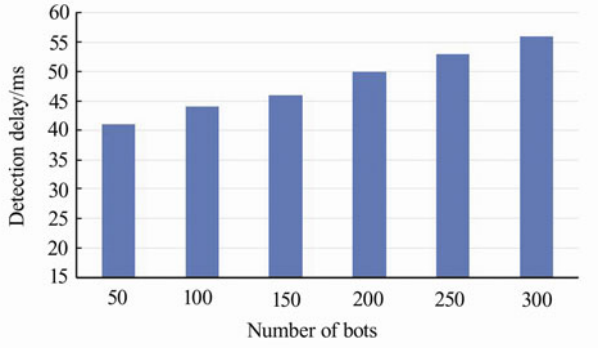

Fig. 10 Detection delay in different botnets

4 Conclusion

In order to resist botnet attacks by software defined networks, we propose a novel lightweight scheme based on graph theory to detect botnets. This scheme equips a SDN controller as a smart brain, named BotGuard. This smart brain has a pair of eyes along with the global view of the whole network and the ability of centralized controlling over the internet. Moreover, BotGuard, which is different from existing researches in SDNs, can reduce the delay of detection botnets in the early monitoring time intervals of its lifecycle. The theoretical basis proves that our detection scheme can detect botnets in real time. The experimental results show that BotGuard obtains a high detection accuracy with an acceptable delay. We consider studying the different botnet topology models to extend our scheme in the future.

References

[1] McCarty B. Botnets: Big and bigger [J]. *IEEE Security & Privacy Magazine*, 2003, **1**:87-90.

[2] Langner R. Stuxnet: Dissecting a cyberwarfare weapon [J]. *IEEE Security & Privacy Magazine*, 2011, **9**: 49-51.

[3] Ormerod T, Wang L, Debbabi M, *et al*. Defaming botnet toolkits: A bottom-up approach to mitigating the threat [C] // *The 4th International Conference on Emerging Security Information, Systems and Technologies*. Los Angeles: IEEE Press, 2010:195-200.

[4] Christiaan B, Carlos C, Cedric C, *et al*. Mcafee labs threat report [EB/OL]. [2016-03-21]. *http://www.mcafee.com/us/resources/reports/rp-threats-predictions-2016.pdf*.

[5] Khattak S, Ahmed Z, Syed A A, *et al*. BotFlex: A community-driven tool for botnet detection [J]. *Network and Computer Applications*, 2015, **58**: 144-154.

[6] Gu G F, Porras P, Yegneswaran V, *et al*. BotHunter: Detecting malware infection through IDS-driven dialog correlation [C] // The 16*th USENIX Security Symposium on USENIX Security Symposium*. Berkeley: USENIX Association, 2007:1-16.

[7] Haq O, Abaid Z, Bhatti N, *et al*. SDN-inspired, real-time botnet detection and flow-blocking at ISP and enterprise-level [C] // *IEEE International Conference on Communications*. New York: IEEE Press, 2015: 5278-5283,.

[8] Wijesinghe U, Tupakula U, Varadharajan V. Botnet detection using software defined networking [C] // *International Conference on Telecommunications*. New York: IEEE Press, 2015: 219-224.

[9] García S, Zunino A. Campo M. Survey on network-based botnet detection methods [J]. *Security & Communication Networks*, 2014, **7**(5): 878-903.

[10] Mahmoud M, Nir M, Matrawy A, *et al*. A survey on botnet architectures, detection and defenses [J]. *International Journal of Network Security*, 2015, **17**:1-18.

[11] Bailey M, Cooke E, Jahanian F, *et al*. A survey of botnet technology and defenses [C] // *Cyber Security Applications*

& Technology Conference for Homeland Security. Los Alamitos: IEEE Press, 2009: 299-304.

[12] Tegeler F, Fu X, Vigna G, *et al*. BotFinder: Finding bots in network traffic without deep packet inspection [C] // *Proc* 8*th International Conference on Emerging Networking Experiments and Technologies*. New York: ACM Press, 2012: 349-360.

[13] Gu G F, Yegneswaran V, Porras P, *et al*. Active botnet probing to identify obscure command and control channels [C] // *Proc the* 2009 *Annual Computer Security Applications Conference*. Los Alamitos: IEEE Press, 2009: 241-253.

[14] Gu G F, Zhang J, Lee W. BotSniffer: Detecting botnet command and control channels in network traffic [C]// *Network & Distributed System Security Symposium*. San Diego: NDSS, 2008:1-19.

[15] Gu G F, Perdisci R, Zhang J, *et al*. BotMiner: Clustering analysis of network traffic for protocol- and structure- independent botnet detection [C] // *Proc the* 17*th Conference on Security Symposium*. San Diego: NDSS, 2008: 139-154.

[16] Franois J, Wang S, State R, *et al*. BotTrack: Tracking botnets using NetFlow and PageRank [C] // *International Ifip Tc* 6*th Networking Conference*. Berlin: Springer-Verlag, 2011: 1-14.

[17] Casado M, Freedman M J, Pettit J, *et al.* Ethane: Taking control of the enterprise [C] //*Proc the* 2007 *Conference on Applications, Technologies*, *Architectures*, *and Protocols for Computer Communications*. New York: ACM Press, 2007: 1-12.

[18] Scott-Hayward S, Natarajan S, Sezer S. A survey of security in software defined networks [C] // *IEEE Communications Surveys & Tutorials*. Los Alamitos: IEEE Press, 2016: 623-654.

[19] Scott-Hayward S, O'Callaghan G, Sezer S. SDN security: A survey [C] // *Future Networks & Services*. Los Alamitos: IEEE Press, 2013: 1-7.

[20] Xia W, Wen Y, Foh C H, *et al.* A survey on software-defined networking [C] // *IEEE Communications Surveys & Tutorials*. Los Alamitos: IEEE Press, 2015: 27-51.

[21] Dhawan M, Poddar R, Mahajan K, *et al.* SPHINX: Detecting security attacks in software-defined networks [C] // *Proc* 22*nd Annual Network and Distributed System Security Symposium*. San Diego: NDSS Press, 2015:8-23.

[22] Murphy M, Shabib A A. POX Wiki [EB/OL].[2016-12-14]. *https://openflow.stanford.edu/display/ONL/POX+Wiki.*

[23] Diestel R. *Graph Theory* [M]. Heidelberg: Springer-Verlag, 2016.

[24] Dao N N. A feasible method to combat against DDoS attack in SDN network [C] // 2015 *International Conference on Information Networking* (*ICOIN*). Los Alamitos: IEEE Press, 2015: 309-311.

[25] Wang R, Jia Z, Ju L. An entropy-based distributed DDoS detection mechanism in software-defined networking [C] // 2015 *IEEE Trustcom/BigDataSE/ISPA*, Washington D C: IEEE Press, 2015: 310-317.

[26] Mininet Team. Mininet: An instant virtual network on your laptop (or other PC) [EB/OL]. [2016-12-14]. *http*: *// mininet. org*/.

[27] Ben P, Bob L, Brandon H, *et al*. OpenFlow switch specication [EB/OL]. [2016-10-14]. *http://archive.openflow.org/documents/openflow-spec-v*1.1.0*.pdf*.

□