A SDN-Oriented DDoS Blocking Scheme for Botnet-Based Attacks

S. Lim, J. Ha, H. Kim
School of Informatics
Korea University
Seoul, Korea

Y. Kim, S. Yang
Next Communication Research Laboratory
ETRI
Daejon, Korea

***Abstract*—DDoS attacks mounted by botnets today target a specific service, mobilizing only a small amount of legitimate-looking traffic to compromise the server. Detecting or blocking such clever attacks by only using anomalous traffic statistics has become difficult, and devising countermeasures has been mostly left to the victim server. In this paper, we investigate how a software-defined network (SDN) can be utilized to overcome the difficulty and effectively block legitimate looking DDoS attacks mounted by a larger number of bots. Specifically, we discuss a DDoS blocking application that runs over the SDN controller while using the standard OpenFlow interface.**

Keywords—DDoS; botnet; blocking; SDN; POX

I. INTRODUCTION

Security is expected to be an important application area for the software defined network (SDN) [1]. This is because network security requires a close coordination of many network components to defend against an attack. Conventional routers are so difficult to modify their behavior. SDN architecture based on OpenFlow specification [2] makes it much easier to dynamically modify the behavior of network switches[1]. Moreover, in the traditional Internet architecture, routers perform routing and control in a distributed manner. It is hard to elicit an orchestrated network-wide behavior. However, in SDN, the existence of a central controller makes the task so much easier. In this paper, we show that SDN makes it really easy to orchestrate network switches to perform a defensive flow management. In particular, we take the example of distributed denial-of-service (DDoS) attack that relies on a large botnet. By using only the standard OpenFlow interface, a DDoS blocking system can be immediately constructed.

As DDoS attack techniques evolve, effective defense is becoming a challenging task. The representative DDoS attacks today typically target specific services so that only the targeted application is disabled while other network components (*e.g.* links, switches, routers) are not affected much. Such a targeted attack can easily hide itself in normal traffic due to low required attack intensity. For example, a HTTP GET flooding attack utilizes the vulnerability of HTTP web server implementation (*e.g.* the maximum number of concurrent connections for HTTP) [3]. In particular, modern DDoS attacks exploit a potentially large number of bots, or compromised hosts. Since these otherwise innocent hosts issue legitimate looking service requests to the attacked server, the attack traffic looks like normal traffic in terms of pps (packets per second), packet size, and packet contents so that it is harder for existing DDoS solutions to block them out from normal packets. In the same vein, signature-based defenses for DDoS attack are not enough to counteract effectively. Also, because the attack traffic volume is small, packet block method based on anomalous traffic statistics does not work, either.

Due to the difficulty in coping with the botnet-based attack inside the network, the load of identifying and blocking the attacks tends to shift to the victim server side. For example, URL redirection method [4] intentionally drives users to other route for the wanted services with a supplementary domain name. Pre-programmed bots cannot easily follow to the supplementary domain name because it is hard to immediately change the programmed victim's domain name. Consequently, the servers can evade the denial of service from bots and provide continued service.

In this paper, we show that the emergence of SDN can reverse the trend and make the network easily configurable to build a robust defense against cleverly organized and targeted DDoS attacks. In particular, we show that the proposed scheme can effectively block the botnet-based DDoS attacks that do not exhibit any statistical anomalies that traditional network anomaly detection schemes can exploit for detection and blocking. The contributions of this paper can be summarized as follows.

- A DDoS blocking scheme that uses only the standard OpenFlow APIs is designed for operation in general SDN environments.
- The scheme is shown to work with minimal support from the server under attack. In particular, the server does not have to have a physical replicated service unlike the URL redirection schemes.
- The scheme is implemented as a SDN application that runs on a popular SDN controller, POX [5].
- The implemented code is tested through emulation on Mininet [6], where it is shown that DDoS attack using botnet is effectively blocked.

[1] In this paper, we use the terms "router," "switch," and "flow switch" interchangeably.

The rest of the paper is organized as follows. Section II defines the system architecture we consider. It first clarifies the assumptions as to the behavior of the components in the SDN-controlled network. Then it explains the main ideas, in terms of the functions and the interactions of the system components. It also discusses how the designed system is implemented in Python to run on the POX controller, along with the standard OpenFlow interfaces used in the implementation. Section III validates the implemented Python logic using the Mininet emulator. The emulation shows that the POX application successfully blocks the DDoS attack traffic and safely redirects legitimate traffic from the attacked server address to a new address. Section IV briefly discusses prior works that are related with this paper. Finally, Section V concludes the paper with a consideration for future work.

II. System Architecture

A. Assumptions

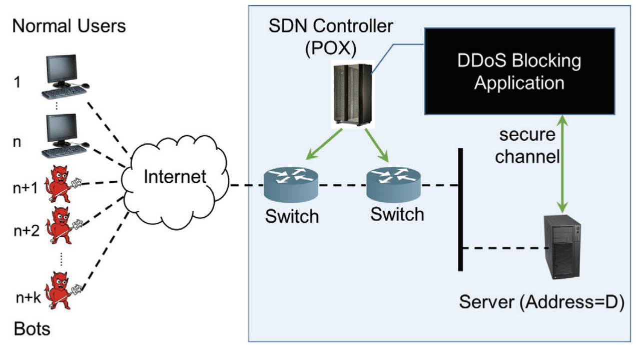

Fig. 1. System architecture

Figure 1 shows the simplified illustration of the system architecture. It is composed of a protected service, SDN controller and the applications running on the controller, OpenFlow switches that are controlled by the controller through the OpenFlow interface. In addition, there are *n* legitimate users of the service and *k* bots that access the service. For subsequent discussions based on the system architecture, we make the following assumptions about the components in this paper.

- Protected servers inside the SDN-based network establish secure communication channels with the DDoS blocking application that runs on a SDN controller, *e.g.* POX. For convenience, we will call the blocking application DDoS Blocking Application (DBA) in this paper. The secure channel is used by the server to notify DBA of a DDoS attack, and by the blocking application to provide the redirected address to the server.

- DBA manages a pool of public IP addresses that can be used for redirection of the protected service. The addresses may be of physically replicated service at some other location, but another address in the same subnet of the protected server can be used. This is especially easy if the network uses IPv6 addressing, since there are a large number of usable addresses under the same prefix. In this paper, we assume the latter, but the proposed system is not dependent on the assumption. The server under attack can logically move its service from the attacked address to a redirected address. The legitimate clients are asked by the server to access the service at the redirected address.

- The DDoS attack is mounted by a botnet, and the botnet does not use IP address spoofing. The non-IP spoofing assumption contrasts to some related works (*e.g.* [7]), and is based on recent trend where most botnets do not have to rely on IP spoofing to confound any tracking effort. Also, the bots do not exhibit any statistically anomalous features that can be observed by the network.

Based on these assumptions and the system architecture depicted in Fig. 1, we design a SDN-based DDoS blocking scheme and in particular, the DDoS blocking application (DBA) component.

B. Work flow

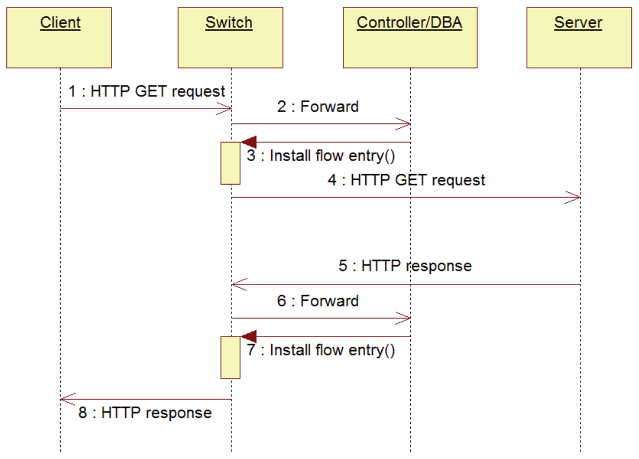

Fig. 2. New flow arrival (HTTP GET and response)

When a new request from a client arrives at an OpenFlow switch, it does not match any existing flow. It is reported to the controller, which directs the switch to create a flow table entry for the new packet (Fig. 2, step 3). When the server responds to the request from the client, another flow entry is created at the switch, but in the reverse direction.

Using the new flow reports, DBA monitors the number of flows at each flow switch. Meanwhile, the server monitors metrics that indicate a possible DDoS attack. When the server determines that a DDoS attack has commenced and the server collapse is imminent, it notifies DBA through the secure channel. In response, DBA provides the server a new IP address *D'*, at which the service should resume (Fig. 3). Once the redirect address is given, the server opens a new socket for the service at the new address (Fig. 3). Then it begins to provide any client arriving afterwards with the address *D'* as the new location for the service, and tears down the connection (Fig. 4). In order to prevent bots from decoding the redirect message and follow along to address *D'*, the redirect address information is provided to the client in the form that imposes high computation barrier or cost for bots. In this paper, we use CAPTCHA [8] for this purpose, but any scheme that makes the bots experience difficulty understanding the redirect message will serve the purpose. In this paper, we assume that *D'* is an address that shares the same prefix with *D*. Note that should the bots manage to follow to the address *D'*, another address *D''*, *D'''*, and so on, will be used to redirect requests from the clients. Note that it does not necessarily mean that the server has to have multiple physical replicas, although the option is not excluded. In this paper, however, we assume that these addresses are all assigned to the same physical server by the SDN controller.

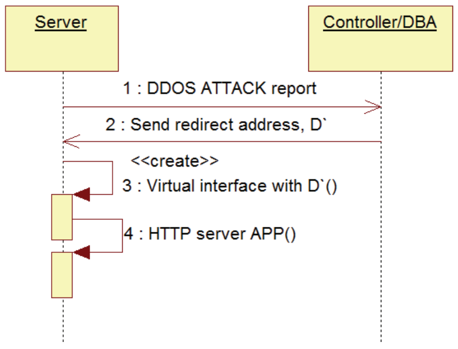

Fig. 3. Redirection begins after the onset of a DDoS attack

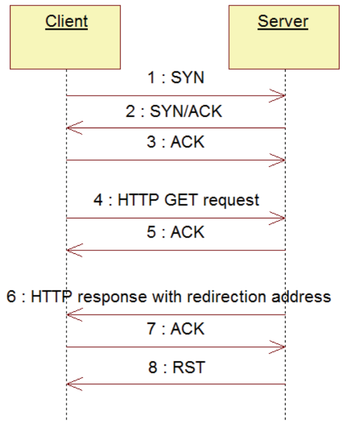

Fig. 4. Redirection begins after the onset of a DDoS attack

Once the server begins to issue the redirect messages for incoming requests, the DBA instructs the switches to allow flows destined to *D'*, which is the redirect IP address of the attacked server. Also, it turns on the counter for the newly reported flows destined to *D*, and if the counter for a particular client goes over a threshold θ, classifies it as a bot, and drop the arriving packets the bot (Fig. 5, steps 11 and 12).

In Fig. 5, steps 1 through 8 are repeated for each new arriving connection for the server. The redirection response with CAPTCHA-coded address *D'* is given to the clients. Note that the response does not have the HTTP Redirect response, but a normal response carrying data, which is the redirection information. Bots are assumed to fail to understand it and ignore the request. After repeated violations exceeding θ, the client is classified as a bot, and steps 9 through 12 are executed for the packets arriving from the client. Although not shown in Fig. 5, all flow entries created by the incoming packets from the client and those created by the server to respond to them are deleted from the flow switch to prevent flow table from overflowing with unnecessary entries.

In step 3 and 7 in Fig. 5, we create a flow entry in the flow table. The difference is that in step 3, the flow entry has three matching fields: source and destination IP addresses, and source port number. On the other hand, the flow entry for step 7 has only two matching entries: source and destination IP addresses. The reason that the flow table entry made for the client-originated packet has the source port number is to differentiate repeated requests from the same client after the redirection is initiated. We let the server reset the connection after the redirection response by a TCP reset (see Fig. 4). Then the next request from the client cannot use the same connection. Thus for the next request, a new connection establishment is necessary, which will create another flow entry as the new connection uses a different source port number. Note that the flow entry creation is only done after the arrival of a (connection establishment) packet that does not match any existing flow entry thus the controller is notified. Therefore, as the number of such consecutively created entries exceeds threshold, the controller notices. DBA can easily keep track of the number of flow table entries that have the same IP source-destination pair, based on which a client is classified as a bot.

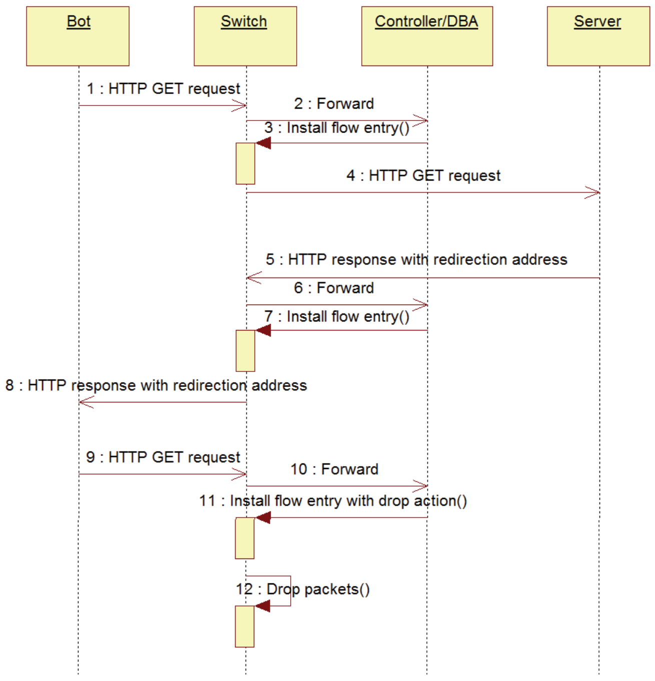

Fig. 5. System behavior against repeated bot requests after the redirection decision

When a client is classified as a bot, a flow entry with "drop" as the associated action is installed (Fig. 5, step 11). We will call such entry the Drop entry for convenience. One problem is the flow table explosion due to the explosive growth of the Drop entries that are created by bot requests. Since each flow switch in the given network on the path from the bot to the attacked server will see the attack packets traversing them in sequence, there will be identical flow entries created by the attackers. When a bot is identified, we could install drop entries at each of the switches, but it would be both unnecessary and wasteful. It is unnecessary because the first flow (*i.e.*, ingress) switch on the path will block all subsequent attack packets from the bot, so that downstream switches will not see the packets. Maintaining the Drop entries for all bots would be only wasteful for these switches. Therefore, DBA instructs all flow switches downstream of the ingress switch to entirely delete the flow entry for the bot. As a consequence, Drop entries will remain only in the perimeter switches. There is a remaining issue of reducing the flow table entries at the perimeter switches in face of a large botnet-mounted attack, and we will investigate it in our future work.

C. DBA implementation

```
function handle_PacketIn(under_ddos_attack, $\theta$, p)
1.  if under_ddos_attack /* only executed under attack */
2.    if p.srcIP $\notin$ flowCount
3.      flowCount[p.srcIP] = p.tcpSrcPort
4.    else if p.tcpSrcPort $\notin$ flowCount[p.srcIP]
5.      flowCount[p.srcIP].add(p.tcpSrcPort)
6.  if length(flowCount[p.srcIP])> $\theta$  /* bot */
7.    match.srcIP = p.srcIP
8.    match.dstIP = p.dstIP
9.    action = drop
10. else /* flow count is still small or server is not under attack */
11.   match.srcIP = p.srcIP
12.   match.dstIP = p.dstIP
13.   match.tcpSrcPort = p.tcpSrcPort
14.   action = forward
15.   if packet.dstIP == D′  /* a redirected flow arrives */
16.     match.srcIP = p.srcIP
17.     match.dstIP = D
18.     sendFlowDeletionCommandToSwitch(match)
19.     match.srcIP = D
20.     match.dstIP = p.srcIP
21.     sendFlowDeletionCommandToSwitch(match)
22. sendFlowAdditionCommandToSwitch(match,action)
```

Fig. 6. Pseudo code of DBA logic that is executed upon a new flow arrival

Fig. 6 shows the pseudo code of `handle_PacketIn` function that is executed upon a new flow arrival, which is the main part of the DBA implementation. Note that DBA is an application running on a SDN controller such as POX, not on flow switches. Flow switches are only required to use standard OpenFlow implementation when they communicate with the SDN controller. The `handle_PacketIn` function takes three input parameters. The `under_ddos_attack` is a flag that is set when the server let DBA know that it is being DDoS attacked. Note that the judgment is made by the server itself, not by DBA. This is because unless the DDoS attack utilizes obviously anomalous measures such as IP spoofing [7], network statistics such as flow count may not be a clear metric to determine the presence of attacks. Under legitimate looking attacks by large botnets, it is the server that knows with certainty that the incoming requests are beyond what it can handle. The next parameter, θ, is the maximum flow count for a client when redirection is under way, as we described above. Lastly, `p` is the packet that was forwarded by the flow switch due to the lack of matching flow entry.

Lines 1–5 are executed only when the server is `under_ddos_attack`. If the sender of the packet is not being flow counted (line 2), a new counting entry is created for the client IP, under the source port number from the packet (line 3). Here, `flowCount` is the dictionary data structure saving information on the TCP source port number of flow entries. If the client IP address is being counted already but it is under other source port number (line 4), however, the new source port number is added to the list of those counted (line 5). Again, note that this counting preparation is not done when the server is not under attack.

If the flow is being counted and the count exceeded the prescribed threshold θ (line 6), the client should be classified as a bot, and its further traffic should be dropped by flow switches before it reaches the server. To this end, a Drop entry should be created at the switches, with `p.srcIP` and `p.dstIP` as the matching condition (lines 7 and 8). If the server is not under attack or the count has not exceeded the threshold, subsequent packets in the flow should be processed without alerting the controller. Thus a matching Forward entry is made with the IP address pair and the source port as the matching condition (lines 11–13). If the packet is sent to *D'*, according to a prior redirect response, any remaining flow table entries from this client to *D* should be removed. Also, the response flow entry from the server (*D*) to the client should be removed. Then flow deletion message for these entries are sent to the flow switch (lines 18 and 21). Finally, flow entry addition message is sent to the flow switch, whether for Drop or for Forward.

D. POX and OpenFlow interfaces

We implemented the workflow discussed above for POX, a Python-based SDN controller [5]. The Python implementation of the entire DBA logic is approximately 300 lines. The code is immediately usable on the POX platform, but for a large-scale test we run the implemented code on the Mininet emulator in the next section.

The following two standard OpenFlow APIs are enough to implement most of the logic required in the workflow that we discussed above:

- OFPT_PACKET_IN
- OFPT_FLOW_MOD

OFPT_PACKET_IN message is used by the OpenFlow switch to report the arrival of a certain packet. Most typically, it takes place when the packet does not match any flow in the flow table. The reason code is also included in the message. In case of the absence of a matching flow entry, it is OFPR_TABLE_MISS. OFPT_PACKET_IN message is used in steps 2 and 6 of Fig. 2, steps 2, 6, and 10 of Fig. 5.

OFPT_FLOW_MOD message is used in our system in steps 3 and 7 of Fig. 2, and steps 3, 7, and 11 of Fig. 5. It is used by the controller to instruct the OpenFlow switch to create, delete, or modify a flow table entry. In each case, the command part of the message is one of the following:

- OFPFC_ADD
- OFPFC_DELETE
- OFPFC_MODIFY

OFPFC_ADD is used when a flow table entry should be created at the flow switch. Obviously, it takes place when a new flow arrives. Another use of the command is when a source IP address is found to be a bot by DBA, and the flow table entry to drop the packets from the bot should be installed at the flow switch. Normally, OFPFC_DELETE is used when a flow terminates. Additionally, in our system, it is used to purge all unnecessary flow entries after a source IP address is determined to be a bot. By the time the event takes place, there are θ entries created by the requests from the bot, and as many entries created by the server issuing the redirection responses to them. OFPFC_MODIFY is not used in our system.

When a flow table entry is made, an action is associated with it. We use the following actions in our system, both of which are mandatory in the OpenFlow standard:

- Output
- Drop

Output is the action to forward the packet. For instance, it is used in steps 4 and 8 of Fig. 5. Drop is obviously to drop the given packet. It is used in step 12 of Fig. 5, after the flow entry to match and drop the bot-originated packets is installed, in step 11 of Fig. 5. In order to find a matching flow in the OpenFlow API, various fields can be used in the packet, such as IP protocol number, IPv4 addresses, IPv6 addresses, TCP/UDP port numbers, incoming switch port, Ethertype, and MAC addresses. For our system, we use IP protocol number, IP addresses, and the source port number.

III. EXPERIMENTS

In order to validate the correctness of the logic in the proposed scheme and evaluate its performance, we test the DBA code for POX. For a massive attack experiment, we choose to use network emulator called Mininet [6]. Mininet is a network emulator that can model a network of virtual hosts, switches, controllers, and links for SDN. Compared to simulators, Mininet runs real, unmodified code including application code, OS kernel code, and control plane code (both OpenFlow controller code and Open vSwitch code), and easily connects to real networks. In particular, our DBA code that runs on real POX platform also runs on Mininet in emulation. In emulation, Mininet hosts run standard Linux network software, and Mininet switches support OpenFlow for highly flexible custom routing and software defined networking. For our experiments, we use Mininet 2.1.0, which supports OpenFlow v1.3 standard. Also on the Mininet testbed, we create a large number of bots to mount a DDoS attack on a protected server in a SDN-managed network.

A. Parameters

TABLE I. MININET EMULATION PARAMETERS

Parameter	Meaning	Default value
θ	Attack flow entry count threshold	3
C_{max}	No. of maximum allowable connections	600
C_{attack}	No. of attack signaling connections	300
n	No. of legitimate users	700
k	No. of bots	300
λ_{norm}	Average request arrival rate from legitimate users	0.33 (/s)
λ_{attack}	Average request arrival rate from bots	1 (/s)
μ	Service rate	0.1(/s)
-	Request arrival process	Poisson

Table I summarizes the parameters used in our experiment. We assume there are two groups of clients, legitimate and malicious. The legitimate users issue a request every 3 seconds, whereas the bots every 1 seconds. Namely, bots are modeled as being more active than legitimate users, but not as anomalously as can be noticed by the network. The requests are issued with the exponential inter-arrival time distribution. Today's botnets can be huge, sometimes boasting up to half a million-strong army of bots. Unfortunately, we could not emulate such a large-scale attack, due to the Mininet limitation. As Mininet can emulate only up to several hundred nodes we scaled down the server to have a maximum of 600 connections at a time. The server alerts the DBA when a large fraction of the maximum allowable connections is exceeded. In this paper, we assume that the server suspects a DDoS attack if the load reaches 50% of the maximum capacity, which is 300 connections. We assume that there are n=700 legitimate users and k=300 bots that issue requests at their given intensities. For each request, the server is assumed to be responsive, so the response is returned in no time.

B. Emulation results

Fig. 7 shows the results of the emulation. The horizontal axis is time in seconds, and the vertical axis is the number of measured entities. We measure three entities here: blocked bots, redirected connections, and (remaining) connections to attacked address. In the emulation scenario, the legitimate connection arrival rate at the server is 5/s. On average, each connection is maintained 10s, during which the clients issue HTTP GET at λ_{norm} =0.33/s. Bots on the other hand issue the request at λ_{attack} =1/s as long as the connection is maintained. In Fig. 7, there are only legitimate users until t=32, when bots begin to attack the server, sending HTTP GETs at λ_{attack}. It results in steep increase in the number of connections to the attacked address. At t=57, C_{attack}=300 is violated, and DBA kicks in at the request of the server. At around t=62, bots begin to be classified and blocked. The connections at the attacked address are reduced to zero as bots are blocked and legitimate connections are redirected to a new service address. All bots are blocked by the perimeter flow switches at t=71.

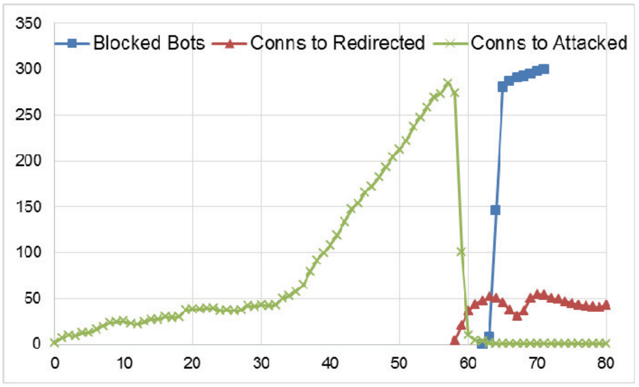

Fig. 7. Bot-blocking dynamics of the proposed system

C. Discussion

As we can observe, the attack blocking strength of the proposed scheme comes from the fact that bots are poorly programmed to understand the redirect directive from the server under attack. We assumed CAPTCHA is used for this purpose in this paper. However, CAPTCHA is rather a plug-in component than an essential component. As there are a variety of CAPTCHA techniques and as many CAPTCHA breakers, the proposed scheme cannot entirely depend upon the strength of CAPTCHA to deter bots from understanding the redirection message and following to the new server address. It is up to the server to present the redirection information to legitimate clients in a format machine decoding is computationally expensive. As the design of DBA is orthogonal to the details of the presentation, servers can freely change the presentation method as necessary. Another issue to be further investigated in our future work is relaxing the assumption that the server needs to cooperate with DBA, that it has to use the redirect address provided by DBA. To relieve the server of the burden, we will look into the possibility that the address redirection and the redirection response can be done by DBA and by the network through OpenFlow interface.

IV. Related Work

Although security is said to be a “killer app” for the SDN, most existing security-related works in SDN is focused on the vulnerabilities of SDN and on how to resolve them and there is a dearth of work on using SDN to improve security in future networks. In the literature, there are a couple of prior works that take resemblance to our work. Jafarian *et al.* propose to mutate the server address frequently so that the network scanning prior to attack delivers an address of the victim that will be invalidated at the time of the attack [9]. But this scheme requires support from external network components such as DNS, and is not scalable. Braga *et al.* [7] discuss how to detect DDoS attacks in SDN. In the proposed scheme, the detection application on the NOX controller computes six features by obtaining flow statistics from switches through the OpenFlow interface. Using the features and a neural network method called Self Organizing Maps (SOM), it determines if a given flow comprises a DDoS attack. A shortcoming of the scheme, however, is that it assumes IP address spoofing by the attacker. The six features are all derived based on the assumption. Today, most DDoS attacks are mounted through botnets, and there is little incentive for the attacker to employ IP spoofing because the bots are otherwise legitimate users, so the use of IP spoofing only raises the alarm. Shin *et al.* [1] discuss a Click-inspired programming framework that enables easy implementation of security applications through modular composition of security constructs.

V. Conclusion

This paper proposes a DDoS blocking scheme applicable to a SDN-managed network. The scheme requires communication between the DDoS blocking application running on the SDN controller and the server to be protected. But all other interactions are performed in the standard OpenFlow interfaces. The proposed scheme culminates in an application that orchestrates the defense through simple and few OpenFlow interfaces. The implemented code for POX controller is validated on Mininet emulator, and is shown to block DDoS attack from bots. In our future work, we plan to reduce the dependence on the pre-arranged cooperation between the SDN controller and the protected server so that SDN can provide transparent protection to servers inside it.

Acknowledgment

This research was funded by the MSIP (Ministry of Science, ICT & Future Planning), Korea, in the ICT R&D Program 2013.

References

[1] S. Shin *et al.*, “FRESCO: Modular Composable Security Services for Software-Defined Networks,” Proc. ISOC NDSS, 2013.

[2] Open Networking Foundation, OpenFlow Switch Specification v.1.4.0, Oct. 2013.

[3] The Honeynet Project, Uses of Botnets, Aug. 2008. http://www.honeynet.org/node/52.

[4] D. Andresen, T. Yang, V. Holmedahl, and O.H. Ibarra, “SWEB: towards a scalable World Wide Web server on multicomputers,” Proc. IPPS, Apr. 1996.

[5] About POX, http://www.noxrepo.org/pox/about-pox/.

[6] Mininet. http://mininet.org/overview.

[7] R. Braga, E. mota, and A. Passito, “Lightweight DDoS Flooding Attack Detection Using NOX/OpenFlow,” Proc. IEEE LCN, pp. 408-415, 2010.

[8] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford, “CAPTCHA: Using Hard AI Problems for Security,” Lecture Notes in Computer Science 2656, pp. 294-311, 2003

[9] J. H. Jafarian, E. Al-Shaer, and Q. Duan, “OpenFlow Random Host Mutation: Transparent Moving Target Defense using Software Defined Networking,” *Proc. ACM HotSDN*, pp. 127-132, 2012.