Credibility-Based Countermeasure Against Slow HTTP DoS Attacks by Using SDN

You-Chiun Wang
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, Taiwan
ycwang@cse.nsysu.edu.tw

Ren-Xuan Ye
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, Taiwan
m073040025@student.nsysu.edu.tw

***Abstract*—In *slow HTTP DoS (SHD)* attacks, the attacker sends HTTP requests in pieces slowly, one at a time to a web server to exhaust its resource and achieve denial of service. Such attacks are easy to launch but hard to defend by conventional solutions like firewall. By exploiting the *software-defined networking (SDN)* technique, the paper proposes a *credibility-based countermeasure against SHD attacks (CCSA)*, which appraises each client by its connections and the frequency that it sends fragmented requests. The connections of low-credibility clients will be blocked to avoid them depleting resource. When the server is short of resource, suspicious connections are then suspended to ensure the server's availability. Simulation results verify that CCSA can efficiently stop SHD attacks and keep low memory usage for the controller.**

***Keywords*—credibility, denial of service, slow HTTP DoS attack, software-defined networking (SDN), web service.**

I. Introduction

HTTP (hypertext transfer protocol) is widely used to support the web service, which is basically a request-response protocol suitable for the client-server model. Specifically, a client (e.g., computer) sends an HTTP request to the web server. Then, the server returns an HTTP response to the client, which includes the status code about the request (e.g., 200 OK) and possibly the content requested by the client in the payload.

In HTTP, a request can be fragmented and sent at different times to react to network congestion [1]. An attacker can split HTTP requests into pieces and send them as slow as possible (before timeout). In this way, the HTTP connection built by the attacker can hold for a very long time, without triggering the firewall. Even worse, the default timeout of Apache-based HTTP servers is also long (i.e., 300 seconds [2]). The attacker can exhaust the server's resource and cause denial of service (DoS) by opening just a number of long sessions. Such attacks are known as *slow HTTP DoS (SHD) attacks* [3].

Software-defined networking (SDN) is a mature technology for network management. It logically partitions a network into control and data planes [4]. The control plane is presided over by the *controller*, while the data plane is located in switches. With this architecture, the controller can query switches about their statuses and give instructions by installing flow rules in their memory. Thus, it becomes easy to monitor and configure a network. Recently, SDN has been applied to various fields, such as recognizing rogue Wi-Fi APs [5], doing authentication for cyber-physical systems [6], and managing data centers [7].

With the help of SDN technology, we propose a *credibility-based countermeasure against SHD attacks (CCSA)*. Specifically, the controller evaluates the clients that have connections with the web server. When a client builds many connections or submits a lot of fragmented HTTP requests, it will have low credibility. In this case, the client is likely to produce an attack, and its connections are blocked to defend the server. Once the server is facing resource shortage, the controller then refers to the trust counter associated to each client and the credibility of its subnet to search for suspicious connections to be blocked. The design of CCSA is lightweight to help the controller find out attacks with a low cost. Through simulations, we show that CCSA can quickly block SHD connections, efficiently reduce false alarms, and also stably keep low memory usage for the controller, as compared with other SDN-based methods.

II. Related Work

Many studies apply SDN to find out *distributed DoS (DDoS)* attacks, which are usually carried out by ordering a botnet to send numerous packets (possibly with fake IP addresses) to a victim server. In [8], switches keep monitoring UDP packets handled by their ports. Once a port has much more incoming packets than outgoing ones, the destinations of certain packets may not exist (so they will be dropped). Some studies [9], [10] ask the controller to check the signature of each packet (e.g., TCP flags) to recognize attacks, which could increase its load. Both [11] and [12] judge whether packets are generated by an attack by evaluating their entropy. Based on [13], the work [14] proposes a nested reverse-exponential storage scheme to help the controller efficiently record packet information. Then, it checks whether a flow is an attack according to the flow size, IP variability, and also duration.

The above approaches are all based on the observation that the DDoS attack produces many packets to paralyze a server. However, in SHD attacks, an attacker gradually depletes the server's resource by sending fragmented requests slowly. Since there are just few packets sent by the attacker, these schemes cannot be applied to withstand SHD attacks.

Several studies resist SHD attacks by SDN. Park et al. [15] adjust the timeout of a web server to block those connections

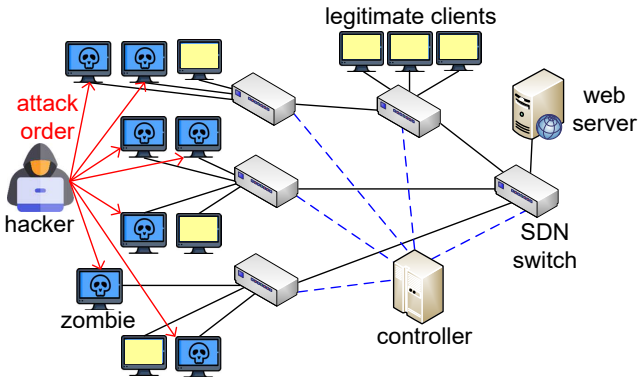


Fig. 1. Using SDN to detect SHD attacks and block their connections.

that send many fragmented requests. However, some legitimate clients that encounter network congestion will be also banned, which causes false alarms. Thus, Hirakawa et al. [16] consider not only shortening timeout but also limiting requests of each client. However, one can launch SHD attacks through a botnet, where each zombie builds just a few connections. In [17], the controller keeps each fragmented request in its buffer. When the rest of a request is received, it then forwards the complete request to the web server. Nevertheless, the controller will be busy processing requests. In view of this, our work develops a lightweight CCSA scheme to efficiently resolve SHD attacks, which considers reducing the controller's overhead.

III. The Proposed CCSA Scheme

Let us consider a system model in Fig. 1, where the network backbone is formed by SDN switches managed by a controller. Both legitimate clients and zombies (i.e., members of a botnet) coexist in the network. A hacker launches an SHD attack by ordering zombies to send fragmented HTTP requests slowly to the web server. On the other hand, some legitimate clients may also submit fragmented requests to the server (e.g., due to network congestion). The controller can obtain the information of traffic flows from switches (e.g, by OpenFlow messages) [18]. However, neither the controller nor the web server knows who are zombies in the network.

The CCSA scheme contains four mechanisms. Specifically, the *request trimming mechanism* checks if there are too many requests submitted to the server, and suspends the connection with the most requests to avoid overloading the server. The *client evaluating mechanism* measures the credibility of each client according to the number of fragmented requests sent by it, and blocks those clients with low credibility. In the *region assessing mechanism*, if the server has spent much resource, the controller finds suspicious connections to be blocked based on the credibility of their subnets. After that, the *connection limiting mechanism* prevents some clients from setting up lots of connections to cause DoS to the server. Below, we elaborate on each mechanism, followed by the design rationale of the CCSA scheme.

A. Request Trimming Mechanism

Since an attacker attempts to exhaust the resource of a web server, excessive HTTP requests can be considered a symptom of SHD attacks. In particular, let $\hat{\mathcal{U}}$ be the set of clients that have connections with the server. We then check whether the following condition obtains:

$$\sum\nolimits_{\forall u_i \in \hat{\mathcal{U}}} f_i^{\mathrm{R}} \geq N_{\mathrm{req}}/2. \tag{1}$$

where f_i^{R} is the frequency that a client u_i produces requests (measured in requests/second) and N_{req} denotes the maximum number of requests that the server can process every second. If so, it implies that the instantaneous requests have consumed more than one half of the server's resource. Once some clients increase their requests or new connections are built, the server would not have sufficient resource to handle them. To prevent this situation, the controller suspends the client u_i that sends the most requests (i.e., the largest f_i^{R} value, since u_i is the most suspicious) by installing an OpenFlow rule in the switch:

- **Match:**
  eth_type=ether_types.ETH_TYPE_IP, /* IP packet */
  ip_proto=in_proto.IPPROTO_TCP, /* TCP packet */
  ipv4_src=max(list,key=list.get), /* source: client u_i */
  ipv4_dst=server_ip,tcp_dst=80 /* target: web server */
- **Action:** drop
- **Timeout:** hard_timeout=T_s

In the match field, we use "/* */" to give a comment on each condition. The "drop" instruction in the action field indicates that when a packet meets all conditions, it will be discarded by the switch (so as to block the requests issued by u_i). Besides, a short timeout T_s is set for this rule. Thus, client u_i will be suspended for T_s seconds. After timeout, the rule is removed and u_i is allowed to send requests. Here, we use timeout T_s to avoid blocking a legitimate client (i.e., false alarm) too long. The suggested value of T_s is 60 seconds.

B. Client Evaluating Mechanism

Because SHD attacks take advantage of the vulnerability in HTTP design by slowly sending fragmented requests to a web server to exhaust its resource, we can evaluate the credibility of each client according to the number of fragmented requests submitted by it. Generally speaking, as there have been many solutions proposed to mitigate network congestion [19], legitimate clients would not transmit too many fragmented requests due to congestion. Thus, if a client has sent many fragmented requests, there is a good possibility that the client is generating an attack. In this case, the controller commands the switch to drop its subsequent requests to protect the web server.

For practical implementation, each client $u_i \in \hat{\mathcal{U}}$ is associated with a *trust counter* ζ_i for evaluating its credibility, where $\zeta_i \in \mathbb{N}$ and its initial value is ζ_{ini} (e.g., $\zeta_{\mathrm{ini}} = 100$). Every time when u_i sends one fragmented request to the web server, its trust counter is decreased by one. If the value of ζ_i falls below $\lfloor \sigma_{\mathrm{L}} \times \zeta_{\mathrm{ini}} \rfloor$, where $0 < \sigma_{\mathrm{L}} \leq 1/2$, u_i is considered an attacker and its subsequent requests will be dropped. However, to avoid blocking u_i forever (which can be viewed as another

Algorithm 1: Region Assessing Mechanism

```
1 if (S_tot − S_use)/S_tot < δ_S then
2     B_ban ← false;
3     foreach u_i ∈ Û such that ζ_i < ⌊σ_H × ζ_ini⌋ do
4         Suspend client u_i for T_s seconds;
5         B_ban ← true;
6     if B_ban = false then
7         Compute the credibility c_k for each subnet;
8         Pick the subnet n_k with the lowest c_k value
            and then suspend the client in n_k with the
            smallest trust counter ζ_i for T_s seconds;
```

type of DoS), we set a long timeout T_l for the rule (to ask the switch to drop u_i's requests). The suggested value of T_l is 300 seconds. After that, the rule will be discarded and u_i's trust counter will be reset to ζ_{ini}. In this way, the web server can accept u_i's requests again after timeout T_l.

C. Region Assessing Mechanism

When the web server is being attacked or it has served too many clients, the server will face resource shortage. Thus, the region assessing mechanism is invoked to assure the server's availability, whose pseudocode is presented in Algo. 1. In line 1, we check whether the server's residual sockets are enough or not by the following equation:

$$\frac{S_{\text{tot}} - S_{\text{use}}}{S_{\text{tot}}} < \delta_{\text{S}}, \tag{2}$$

where S_{tot} is the total number of sockets offered by the server, S_{use} is the number of used sockets, and $\delta_{\text{S}} \in (0, 1)$ defines the threshold (for example, we can set $\delta_{\text{S}} = 0.2$). If so, the server will run out of sockets soon and the controller suspends those clients whose values of trust counters are below $\lfloor \sigma_{\text{H}} \times \zeta_{\text{ini}} \rfloor$, where $\sigma_{\text{L}} < \sigma_{\text{H}} \leq 3/4$. The code is given in lines 2–5. Here, a boolean variable B_{ban} is used to indicate whether we can find out such clients from $\hat{\mathcal{U}}$.

If B_{ban} is still false in line 6, it means all clients in $\hat{\mathcal{U}}$ have relatively high credibility. In this case, we still have to choose one client to be suspended in order to let the server get back some sockets for the sake of availability. To do so, we estimate the credibility of each subnet n_k by

$$c_k = \frac{\sum_{\forall u_i \in \hat{\mathcal{U}}_k} \zeta_i}{|\hat{\mathcal{U}}_k|}, \tag{3}$$

where $\hat{\mathcal{U}}_k \subseteq \hat{\mathcal{U}}$ is a subset of clients whose IP addresses belong to subnet n_k, which can be easily checked by using IP mask 255.255.255.0. According to Eq. (3), the subnet's credibility is defined as the average value of trust counters of all clients in that subnet. Then, in line 8 we pick the subnet n_k that has the lowest credibility and suspend the client in n_k whose trust counter has the smallest value. In case of a tie, we arbitrarily select one client to be suspended.

D. Connection Limiting Mechanism

Sometimes, the hacker may command some zombies to set up numerous connections with the web server, so as to exhaust its resource in a very short time. In this case, the controller may not have enough time to do reaction by the above mechanisms. To solve this problem, we restrict the number of connections that each client can set up to δ_{C}. Here, we suggest setting

$$\delta_{\text{C}} = \left\lfloor \frac{N_{\text{con}}}{\max\{|\hat{\mathcal{U}}|, \tau\}} \right\rfloor, \tag{4}$$

where N_{con} is the maximum of connections supported by the server, $|\hat{\mathcal{U}}|$ is the number of clients, and $\tau \in \mathbb{N}$ is a coefficient to give the upper bound on the number of connections set up by each client (e.g., $\tau \geq 5$). In Eq. (4), when there are fewer clients, each client is allowed to set up more connections (but no more than $\lfloor N_{\text{con}}/\tau \rfloor$ connections), and vice versa.

For practical implementation, let p_i^{S} and p_i^{F} be the number of TCP packets issued from a client u_i to the web server whose synchronization (SYN) and finish (FIN) flags are set to 1 (also called SYN and FIN packets), respectively. If $p_i^{\text{S}} - p_i^{\text{F}} \geq \delta_{\text{C}}$, the switch will drop subsequent SYN packets sent by u_i, since the number of its connections has reached the limit. In this case, u_i cannot set up new connections with the server (unless it closes the old ones by sending FIN packets).

E. Design Rationale

In CCSA, the request trimming and client evaluating mechanisms are "precautions" against attacks, where the web server still has sufficient resource. The request trimming mechanism prevents clients from overusing the server's resource by sending many requests. Since some clients may do so *accidentally*, the controller suspends them by using a short timeout T_s in the flow rule. On the other hand, the client evaluating mechanism blocks the clients that have sent a lot of fragmented requests for a long timeout T_l, because they could be attackers. In this way, the switch can filter out potentially malicious requests to preserve the server's resource.

The region assessing mechanism is applied when the server is about to use up its resource. In this case, even though there is no attack, some clients need to be suspended to guarantee the server's availability. Therefore, this mechanism takes a more rigorous threshold (i.e., σ_{H}) on the trust counter ζ_i for each client. Moreover, it also picks some connections to be blocked according to the subnet's credibility. However, since there may be more false alarms, the controller sets a short timeout T_s for these flow rules to avoid blocking legitimate clients for a long time. For the future direction, we will consider developing a more sophisticated method to select connections to be blocked to ensure fairness among legitimate clients and maximize their satisfaction (i.e. achieving the Pareto optimality [20]). Finally, the connection limiting mechanism dynamically adjusts the maximum number of connections set up by clients according to Eq. (4). In this way, we can prevent some malicious clients from creating many connections to paralyze the server.

TABLE I
THE SLOWHTTPTEST OPTIONS USED IN THE SIMULATION.

option	description
-H	Start an SHD attack by sending fragmented HTTP requests.
-c	Specify the maximum number of connections in the attack.
-r	Specify the number of connections added per second.
-l	Specify the duration of the attack (in seconds).
-i	Specify the interval between two fragmented HTTP requests.

IV. EXPERIMENTAL RESULTS

In this section, we build our simulation on Mininet [21] for performance evaluation. To support the OpenFlow protocol, the controller and switches are implemented by the Ryu SDN framework [22] and the Open vSwitch module in Linux [23], respectively. Fig. 1 gives the network topology, where a hacker orders some zombies to launch SHD attacks against the web server. Moreover, there are six legitimate clients that establish connections with the server. Due to network congestion, these clients may also send fragmented HTTP requests to the server occasionally. As mentioned earlier in Section III, neither the controller nor the server knows who are zombies.

To simulate SHD attacks, we use the *slowhttptest* instruction provided by Ubuntu [24], where Table I lists the options taken in the simulation. There are three attack scenarios considered, where the attack lasts for 300 seconds (i.e., "-l 300"):

A1. *slowhttptest -H -c 20 -r 4 -l 300 -i 1*:
Four zombies participate in the attack. Each zombie adds four connections per second, until the number of connections reaches 20. A connection produces one fragmented request every second.

A2. *slowhttptest -H -c 120 -r 3 -l 300 -i 50*:
Two zombies launch the attack. Each of them builds three connections in a second, until 120 connections have been built. A connection submits one fragmented request every 50 seconds.

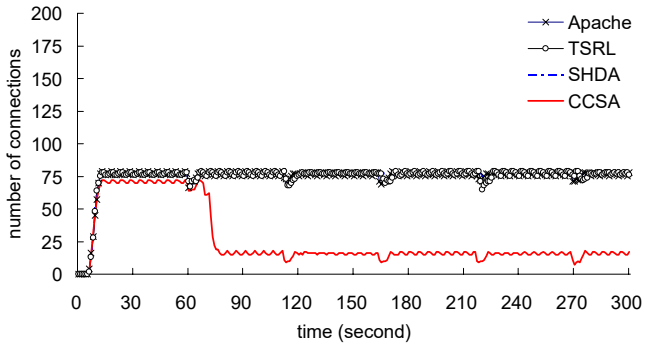
A3. *slowhttptest -H -c 300 -r 20 -l 300 -i 30*:
The attack is originated by a zombie. It sets up 20 connections per second, until there are 300 connections. A connection sends a fragmented request every 30 seconds.

In addition, there will be 15 connections set up by legitimate clients in the duration of the attack.

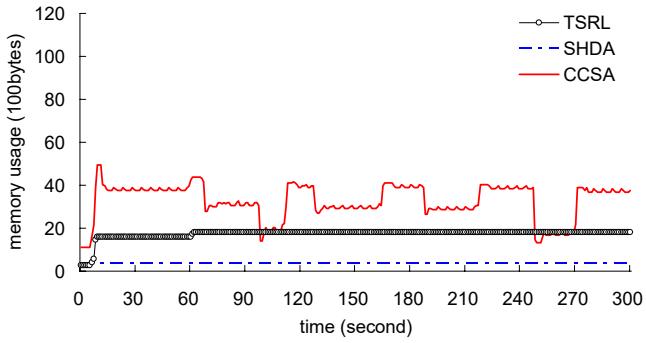
We compare our CCSA scheme with two SDN-based methods in Section II. The *timeout shortening and requests limiting (TSRL)* method [16] blocks the client that establishes the most connections to save the server's resource. In the *slow HTTP DDoS defense application (SHDA)* [17], the controller stores fragmented requests in its buffer and then relays complete ones to the server. Besides, we also observe how an Apache-based server reacts to SHD attacks, which blocks a connection if it cannot get the rest of a fragmented HTTP request after a fixed timeout (in particular, 60 seconds). The maximum number of connections supported by the server (i.e., N_{con}) is set to 100.

A. Attack Scenario A1

Fig. 2(a) shows the number of HTTP connections that the web server processes in scenario A1. There will be no more



(a) number of HTTP connections

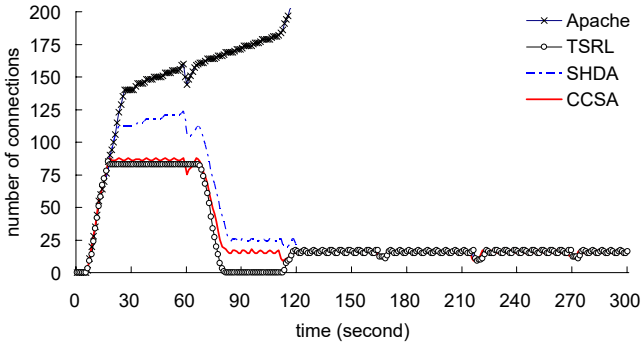


(b) memory usage by the controller

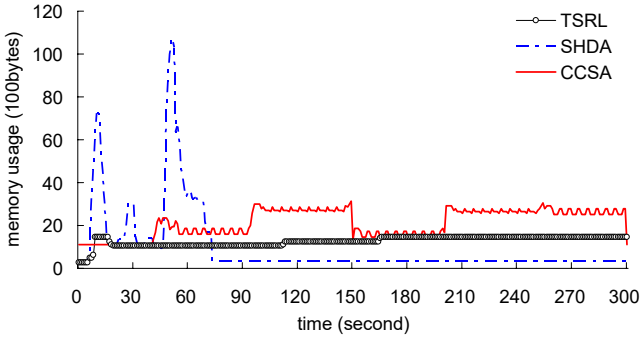
Fig. 2. Performance evaluation in scenario A1.

than 95 connections in the network (i.e., 80 SHD connections built by four zombies and 15 connections set up by legitimate clients), which is fewer than the maximum connections supported by the server (i.e., 100 connections). In other words, the server still has enough resource to offer services. In this case, both TSRL and SHDA will not react to the attack, so they have a similar number of connections with the Apache method. Here, since some connections may be terminated and rebuilt, the number of connections in these methods is kept around 80. For the CCSA scheme, as zombies frequently send fragmented requests, their credibility will decrease. Thus, after the 75th second, most SHD connections are blocked by CCSA, and the server handles no more than 18 connections. This experimental result demonstrates that CCSA can efficiently find out SHD attacks even though the server's resource is not exhausted by zombies, as compared with other methods.

Fig. 2(b) presents the amount of memory consumed by the controller in scenario A1. As the Apache method is applied in the web server (by using a fixed timeout to block connections), there is no memory usage for the controller (so its result is not shown). The TSRL method records merely the number of fragmented requests and source IP addresses, so it can spend less memory. Since the SHDA method does not find out any attack, its protection mechanism is not triggered. That is why SHDA has the least amount of memory usage. On the other hand, our CCSA scheme keeps the amount of memory usage below 4300 bytes in this scenario, which shows that it can help the controller efficiently recognize SHD attacks without consuming much memory.

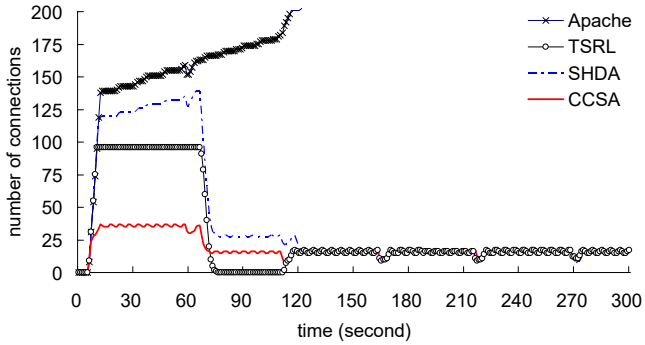


(a) number of HTTP connections

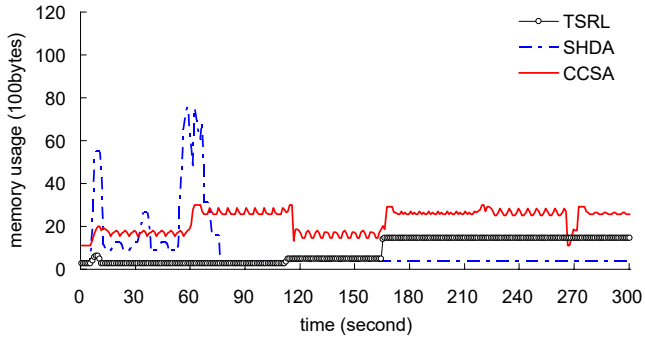


(b) memory usage by the controller

Fig. 3. Performance evaluation in scenario A2.



(a) number of HTTP connections



(b) memory usage by the controller

Fig. 4. Performance evaluation in scenario A3.

B. Attack Scenario A2

Fig. 3(a) gives the number of HTTP connections handled by the web server in scenario A2, where two zombies keep setting up connections, until there are 240 SHD connections. Since the Apache method does not block them, the number of connections continuously rises. After the 20th second, the number of connections exceeds the N_{con} threshold (i.e., 100 connections), which means that the server becomes overloaded. Even worse, the number of connections keeps growing rapidly. After the 115th second, there will be more than 200 connections, and the server is obviously paralyzed. For the TSRL method, the number of connections is kept in 83 during the 17th–67th seconds, which means that TSRL finds out SHD connections and blocks some of them to protect the server. However, during the 81st–112nd seconds, there is no connection in TSRL. This phenomenon indicates false alarms, where 15 connections set up by legitimate clients are also blocked by TSRL. For the SHDA method, there are more than 100 connections during the 20th–70th seconds, which implies that it cannot fast block SHD connections (as compared with TSRL). Thus, the server is at stake and its availability cannot be guaranteed. Our CCSA scheme avoids the drawbacks of both TSRL and SHDA. From Fig. 3(a), CCSA not only quickly reacts to the attack but also reduces false alarms, which verifies its effectiveness.

Fig. 3(b) compares the amount of memory consumption of the controller in scenario A2. As mentioned earlier, the TSRL method records less information (e.g., the number of fragmented requests and IP addresses), so it has lower memory consumption. For the SHDA method, since the controller has to store fragmented requests on behalf of the server, it will use lots of memory when many fragmented requests are submitted (i.e., during the 7th–72nd seconds). In our CCSA scheme, the amount of memory consumption is always below 3000 bytes, which is much lower than SHDA.

C. Attack Scenario A3

In Fig. 4(a), we measure the number of HTTP connections with the web server in scenario A3, where the zombie rapidly builds many connections (i.e., 20 connections per second) to exhaust the server's resource. Thus, the DoS to the server is accomplished very soon in the Apache method (i.e., after the 10th second). All other methods handle the attack during the 10th–70th seconds. As can be seen, the number of connections in SHDA overtakes the N_{con} threshold (i.e., 100 connections), so the server is out of service during that period. The number of connections is kept below 97 and 38 in TSRL and CCSA, respectively, which means that CCSA can block SHD connections more efficiently than TSRL. Furthermore, TSRL blocks all connections (including normal ones) during the 76th–111st seconds, whereas CCSA can avoid these false alarms in the same period. The experimental result in Fig. 4(a) validates that our CCSA scheme is superior to both TSRL and SHDA.

Fig. 4(b) then evaluates the amount of memory consumption of the controller in scenario A3. Similarly, the SHDA method makes the controller consume much memory during the 6th–75th seconds, as the controller has to store many fragmented requests. On the other hand, the controller spends no more

than 3000 bytes in its memory by our CCSA scheme, which shows its effectiveness in reducing the controller's overhead.

V. Conclusion and Future Work

Unlike DDoS attacks that fast overwhelm a server with numerous packets, an SHD attack gradually depletes the server's resource by sending fragmented HTTP requests slowly. Thus, existing solutions to DDoS attacks cannot be applied to resist SHD attacks. In view of this, the paper proposes the CCSA scheme by exploiting the SDN technology. It takes necessary precautions against attacks by limiting the number of requests and connections for each client and also blocking those clients with low credibility. When the server is short of sockets, CCSA further suspends some clients by referring to the credibility of their subnets to make sure that the server can have sufficient resource to provide services. Through simulations by Mininet, we show that CCSA can correctly identify SHD attacks and quickly block their connections in different attack scenarios. Moreover, it can keep lower memory usage for the controller, as compared with the SHDA method.

For the future work, we will consider extending our CCSA scheme to a *multi-domain SDN-based network* composed of connected but autonomous domains (i.e., subnetworks) [25]. Because each domain is managed by a stand-alone controller, it is a challenge to make these controllers collaborate to fast find out SHD attacks and eliminate false alarms. Besides, how to exploit SDN to protect mobile networks [26] against SHD attacks also deserves further investigation.

Acknowledgment

You-Chiun Wang's research is co-sponsored by the Ministry of Science and Technology under Grant No. MOST 108-2221-E-110-016-MY3, Taiwan.

References

[1] IETF, "Hypertext transfer protocol – HTTP/1.1." [Online]. Available: https://tools.ietf.org/html/rfc2616

[2] Apache. [Online]. Available: https://httpd.apache.org/

[3] E. Adi, Z. Baig, C. P. Lam, and P. Hingston, "Low-rate denial-of-service attacks against HTTP/2 services," in *International Conference on IT Convergence and Security*, 2015, pp. 1–5.

[4] Y. C. Wang and H. Hu, "An adaptive broadcast and multicast traffic cutting framework to improve Ethernet efficiency by SDN," *Journal of Information Science and Engineering*, vol. 35, no. 2, pp. 375–392, 2019.

[5] J. H. Cox, R. Clark, and H. Owen, "Leveraging SDN and WebRTC for rogue access point security," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 756–770, 2017.

[6] C. Wang, Y. Zhang, X. Chen, K. Liang, and Z. Wang, "SDN-based handover authentication scheme for mobile edge computing in cyber-physical systems," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8692–8701, 2019.

[7] Y. C. Wang and S. Y. You, "An efficient route management framework for load balance and overhead reduction in SDN-based data center networks," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1422–1434, 2018.

[8] H. C. Wei, Y. H. Tung, and C. M. Yu, "Counteracting UDP flooding attacks in SDN," in *IEEE Conference on Network Softwarization*, 2016, pp. 367–371.

[9] P. Rengaraju, V. R. Ramanan, and C. H. Lung, "Detection and prevention of DoS attacks in software-defined cloud networks," in *IEEE Conference on Dependable and Secure Computing*, 2017, pp. 217–223.

[10] K. Kalkan, G. Gur, and F. Alagoz, "SDNScore: a statistical defense mechanism against DDoS attacks in SDN environment," in *IEEE Symposium on Computers and Communications*, 2017, pp. 669–675.

[11] K. Kalkan, L. Altay, G. Gur, and F. Alagoz, "JESS: Joint entropy-based DDoS defense scheme in SDN," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 1–24, 2018.

[12] M. Xuanyuan, V. Ramsurrun, and A. Seeam, "Detection and mitigation of DDoS attacks using conditional entropy in software-defined networking," in *IEEE International Conference on Advanced Computing*, 2019, pp. 66–71.

[13] Y. C. Wang, Y. Y. Hsieh, and Y. C. Tseng, "Multiresolution spatial and temporal coding in a wireless sensor network for long-term monitoring applications," *IEEE Transactions on Computers*, vol. 58, no. 6, pp. 827–838, 2009.

[14] Y. C. Wang and Y. C. Wang, "Efficient and low-cost defense against distributed denial-of-service attacks in SDN-based networks," *International Journal of Communication Systems*, vol. 33, no. 14, pp. 1–24, 2020.

[15] J. Park, K. Iwai, H. Tanaka, and T. Kurokawa, "Analysis of slow read DoS attack and countermeasures on web servers," *International Journal of Cyber-Security and Digital Forensics*, vol. 4, no. 2, pp. 339–353, 2015.

[16] T. Hirakawa, K. Ogura, B. B. Bista, and T. Takata, "A defense method against distributed slow HTTP DoS attack," in *International Conference on Network-Based Information Systems*, 2016, pp. 152–158.

[17] K. Hong, Y. Kim, H. Choi, and J. Park, "SDN-assisted slow HTTP DDoS attack defense method," *IEEE Communications Letters*, vol. 22, no. 4, pp. 688–691, 2018.

[18] Y. C. Wang and H. Hu, "A Low-cost, high-efficiency SDN framework to diminish redundant ARP and IGMP traffics in large-scale LANs," in *IEEE Computer Software and Applications Conference*, 2018, pp. 894–903.

[19] J. Luo, J. Jin, and F. Shan, "Standardization of low-latency TCP with explicit congestion notification: A survey," *IEEE Internet Computing*, vol. 21, no. 1, pp. 48–55, 2017.

[20] Y. C. Wang, "A two-phase dispatch heuristic to schedule the movement of multi-attribute mobile sensors in a hybrid wireless sensor network," *IEEE Transactions on Mobile Computing*, vol. 13, no. 4, pp. 709–722, 2014.

[21] Mininet. [Online]. Available: http://mininet.org/

[22] Ryu. [Online]. Available: https://ryu-sdn.org/

[23] Open vSwitch. [Online]. Available: https://www.openvswitch.org/

[24] Ubuntu manuals, "slowhttptest – Denial of service attacks simulator." [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man1/slowhttptest.1.html

[25] Y. C. Wang and E. J. Chang, "Cooperative flow management in multi-domain SDN-based networks with multiple controllers," in *IEEE International Conference on Smart Communities: Improving Quality of Life Using ICT, IoT and AI*, 2020, pp. 1–5.

[26] Y. C. Wang and C. A. Chuang, "Efficient eNB deployment strategy for heterogeneous cells in 4G LTE systems," *Computer Networks*, vol. 79, pp. 297–312, 2015.