Realtime DDoS Defense Using COTS SDN Switches via Adaptive Correlation Analysis

Jing Zheng, Qi Li, *Senior Member, IEEE,* Guofei Gu, *Senior Member, IEEE,* Jiahao Cao, David K.Y. Yau, *Senior Member, IEEE*, and Jianping Wu, *Fellow, IEEE*

***Abstract*—Distributed denial-of-service (DDoS) defense is still a difficult problem though it has been extensively studied. The existing approaches are not capable of detecting various types of DDoS attacks. In particular, new emerging sophisticated DDoS attacks (e.g., Crossfire) constructed by low-rate and short-lived "benign" traffic are even more challenging to capture. Moreover, it is difficult to enforce realtime defense to throttle these detected attacks since the attack traffic can be concealed in benign traffic. Software Defined Networking (SDN) opens a new door to address these issues. In this paper, we propose RADAR to detect and throttle DDoS attacks via adaptive correlation analysis built upon unmodified commercial off-the-shelf (COTS) SDN switches. It is a practical system to defend against a wide range of flooding-based DDoS attacks, e.g., link flooding (including Crossfire), SYN flooding, and UDP-based amplification attacks, while requiring neither modifications in SDN switches/protocols nor extra appliances. It accurately detects attacks by identifying attack features in suspicious flows, and locates attackers (or victims) to throttle the attack traffic by adaptive correlation analysis. We implement RADAR prototype using open source Floodlight controller, and evaluate its performance under various DDoS attacks by real hardware testbed based experiments. We observe that our scheme can successfully detect and effectively defend against various DDoS attacks with acceptable overhead.**

I. Introduction

The Internet has a long history of suffering from DDoS attacks. Recently there is a dramatic escalation in DDoS attacks, for example, the attacks on Dyn DNS services disconnected many popular Internet services, e.g., Amazon and GitHub, on October 21, 2016 [4]. Traditional methods for DDoS defense [1], [3], [22] have a number of limitations. First, they often require expensive hardware appliances, thus introducing extra deployment cost and complex routing hacks [15]. In addition, they are often unable to detect sophisticated DDoS attacks, e.g., Crossfire [18], Pulsing DDoS attacks [27], as well as some real-world DDoS attacks [14], which were constructed to plague the Internet in a stealthy way. Worse still, it is extremely difficult to enforce realtime defense against detected attacks since the attack is stealthy and the attack traffic can mimic behaviors of benign traffic [14], [17].

Recently, Software Defined Networking (SDN) enables a new way to defend against DDoS attacks. A natural way of design is to rely on SDN switches to collect necessary flow information and report it to the SDN controller. By utilizing the controller's global view on the network-wide flow features, an SDN-based approach can be potentially effective to detect and defend against DDoS attacks, and in fact, such methods have been proposed recently [12], [15], [17], [28]. Although these approaches have different implementation details to detect attacks, they share a similar design principle and architecture, as illustrated in Figure 1(a). A key issue in the designs is to identify which flow information is necessary and should be reported to the controller. Note that, due to its capacity limitation, a controller is not able to receive and analyze based on original flow counter information for all network flows from all switches. A natural idea that all current designs adopt is to rely on switches to perform pre-processing on flow counter information, generate some brief statistics (e.g., changes of flow rates), and report such brief information to the controller.

Such deployments have been shown a number of advantages, but they are facing some fundamental challenges. First, an SDN switch is only able to report flow counter information, but is not designed to accomplish complicated tasks in flow information pre-processing. Thereby, it has to rely on extra components (e.g., appliances) to complete them, which disables the ability to detect DDoS attacks using unmodified commercial off-the-shelf (COTS) SDN switches and incurs extra deployment costs. Second, flow pre-processing in switches (e.g., analysis of changes of flow rates) may lose important original information and thereby the controller will mistakenly omit attack flows. This is especially serious when stealthy attacks (e.g., Crossfire) or non-link-flooding attacks (e.g., TCP flooding) take place. Under such attacks, the information of changes of flow rates that switches report to the controller is always normal, and thus it will be very difficult for the controller to know if attacks happen. Therefore, such approaches may neither be able to detect certain attacks, nor the detection accuracy is questionable. Table I provides a summary of the above representative approaches built upon SDN[1].

It is the lack of deployability on COTS SDN switches, as well as our pursuit of high effectiveness and generality of defense, that motivate our research. We would like to ask:

- *Is it possible to develop an SDN-based approach to effectively detect and defend against a wide range of DDoS attacks by using COTS SDN switches without extra appliances?*

J. Zheng, Q. Li, J. Cao, and J. Wu are with Graduate School at Shenzhen, Tsinghua University, Shenzhen, China 518055, and Department of Computer Science, Tsinghua University, Beijing, China 100084, e-mail: {zhengj14@mails, qi.li@sz, caojh15@mails}.tsinghua.edu.cn, jianping@cernet.edu.cn.

G. Gu is with Department of Computer Science & Engineering, Texas A&M University, Texas 77843, e-mail: guofei@cse.tamu.edu.

D. Yau is with the Information Systems Technology and Design pillar, Singapore University of Technology and Design, Singapore 487372, e-mail: david_yau@sutd.edu.sg.

[1]We want to point out that though Bohatei seems not require switch modifications, it does not detect DDoS attacks but defend against them assuming such attacks are already detected.

TABLE I
COMPARISON WITH EXISTING DDOS DEFENSE BUILT UPON SDN.

Scheme	Functionality[1]				Deployment Cost			
	TCP	UDP/link flooding	Crossfire	Defense	COTS SDN	No Appliance	Deployability	Comm.
RADAR	✔	✔	✔	✔	✔	✔	**w/o changes**	**low**
BROCADE [12]	**N/A**[2]	✔	✖	✔	✖ **(sFlow)**[3]	✖ **(sFlow)**	**customized systems**	**medium**
SPIFFY [17]	✖	✔	✔	**N/A**	✖ **(Sketch)**	✔	**customized systems**	**medium**
Bohatei [15]	✖	✖	✖	✔	✔	✖ **(DC/NFV)**	**w/o changes**	**N/A**

[1] TCP represents TCP SYN flooding detection, UDP and link flooding represents UDP amplification detection and traditional link flooding detection, Crossfire represents detection of sophisticated DDoS attacks, and defense represents if a scheme is able to throttle the attacks in realtime.
[2] N/A indicates the metric of the scheme is unknown or the scheme is not comparable to others.
[3] sFlow requires extra equipment to collect sFlow data.

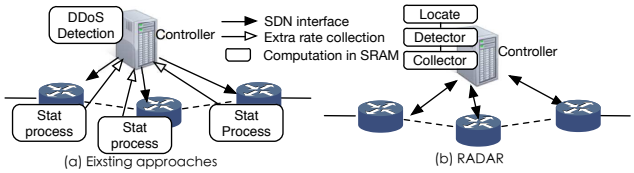

Fig. 1. DDoS defense enabled by SDN.

Our answer is yes. We propose RADAR (Reinforcing Anti-DDoS Actions in Realtime), whose architecture is illustrated in Figure 1(b). Comparing with the conventional approaches, we have some fundamental innovations. First, we enable interactions between controllers and switches so that aggregated anomaly pattern in network-wide traffic triggers collecting traffic in the controller and adaptively zooming in the suspicious set. Thereby, the controller can dynamically tell each switch which flows (instead of all flows) to investigate. Second, based on these limited number of flows under investigation, the switches do not need to pre-process flow information anymore but directly report original flow counter information to the controller. Since the controller only receives such flow counter information of *limited* flows being investigated, it is able to analyze it and keep itself scalable even in a large-scale network. In other words, facing the scalability challenge, traditional approaches rely on switches to simplify the flow information (which is costly and may lose important information) of all network-wide traffic, while our approach relies on the controller to dynamically identify only *a partial set* of suspicious flows, instruct the switches to monitor them and report original flow counter information of such *partial flows* to the controller. We enable RADAR's new features on *deployability, generality*, and *effectiveness* by:

- **Interactions between Switches and Controllers** allow controllers to instruct switches on how to collect partial information, such that switches can perform collection tasks by themselves *without requiring appliances*. Flow filtering is achieved by the controller based on *original flow information*, and thus it is *more accurate* without omitting attack flows.
- **Adaptive Correlation Analysis performed on Controllers** enables identifying and locating *a wide range* of DDoS attacks (e.g., sophisticated link flooding like Crossfire, amplification, and SYN flooding attacks) based on original flow counts collected from switches *in realtime*.

We also point out that SDN approaches are often with scalability concerns. RADAR uses a distributed flow rule placement to significantly reduce the number of rules needed in each switch, and we will show it is highly scalable. In summary, our contributions are three-fold:

- We design the RADAR architecture to detect a wide range and *sophisticated/stealthy* DDoS attacks without any modifications in SDN protocols or COTS switches. It is the first system built upon COTS SDN switches that can detect and throttle various DDoS attacks.
- We develop detailed algorithms in RADAR so that various DDoS attacks are detected and throttled in realtime.
- We implement RADAR prototype in the Floodlight controller, and perform experiments on a real hardware testbed with real traces. The experiment results demonstrate that it can effectively detect and throttle DDoS attacks in realtime and is shown to be scalable.

II. BACKGROUND

A. DDoS Attacks

A distributed denial-of-service (DDoS) attack occurs when attack traffic floods the bandwidth or resources of a targeted system. Although the traffic patterns of traditional DDoS attacks are well defined, it is still difficult to practically defend against them in realtime. Sophisticated DDoS attacks are recently created, e.g., Crossfire [18] and Coremelt attacks [30]. Instead of directly flooding victims, these attacks flood backbone links of ISPs and create a large number of attack flows crossing the links that connect the victims to the Internet. By congesting the links, the victim networks are disconnected from the Internet. The attackers leverage different bots to generate low-rate traffic with real IP addresses, making the detection very difficult. Such attacks have attracted interests, and some potential techniques [17], [19], [20] have been proposed. However, up to now we are not aware of any existing deployment that can effectively defend against such type of DDoS attacks.

Example. Let us use a simple example to illustrate how Crossfire attack works. In Figure 2, a victim node of such attack has N paths to connect to the Internet. Let us first focus on one particular path, i.e., path 1 composed of two serial links, i.e., A and B. Two bots with addresses 32.0.0.1/24 and 240.0.0.1/24 generate traffic with real addresses to two decoy servers that can be reached through A and B, respectively. Two bots cooperate to generate traffic, such that the traffic shifts

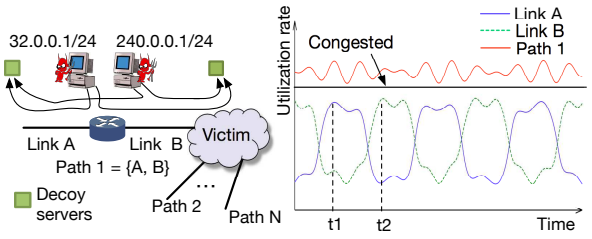

Fig. 2. A simple example of Crossfire attack

between A and B and the links are periodically congested (as shown in the right part of Figure 2)[2]. Thereby, the path to the victim composed of A and B is *always* congested. For ease of illustration, here we only focus on path 1, while in practice, a large number of bots generate various attack flows on all possible links so as to congest the paths, and then the victim is completely disconnected. In fact, there are real-world attacks [14], [16] similar to Crossfire attacks, which use short-lived and small flows to construct DDoS attacks.

B. Problem Statement

We aim to propose a practical system built upon SDN to defend against a wide range of flooding-based DDoS attacks, i.e., link flooding (including sophisticated attacks, e.g., Crossfire), SYN flooding, and UDP-based amplification attacks. The proposed system does not need to modify the current packet forwarding diagram and it is compatible with the current IP data plane. With same assumptions as in the BROCADE DDoS defense products [12] and the software-defined measurement approaches [23], [34], we use SDN switches capable of large flow rule entries so as to throttle attacks. Such powerful switches are achievable in the market [7]. Based our experimental results (see Section VI), we observe that scalability is not a big concern.

In this paper, we use OpenFlow [9] as a representative of network control APIs of SDN and study how SDN can be leveraged to efficiently detect various DDoS attacks. It directly leverages standard SDN (or OpenFlow) APIs without any modifications in any protocols or implementations of SDN. It can work with various releases of SDN controllers, e.g., OpenDaylight [8] and Floodlight [5], and various brands of COTS switches that comply with OpenFlow specifications. Therefore, it can be deployed with one or multiple (or distributed) controllers in large scale networks. Moreover, it does not require any collaborations among ISPs. Consider a particular victim of an attack. The attack traffic will be eventually aggregated in the local network containing the victim. We can detect and throttle the attack traffic to the victim by only investigating this local network, even if the attacks originate from different places of the wide area network.

[2]Both bots can generate any amount of traffic to either decoy server, i.e., a particular bot can vary its traffic on A and B from time to time, as long as the total traffic generated by both bots periodically congests A and B, making the attacks more stealthy.

III. Architecture

In this section, we propose the architecture of RADAR. RADAR is designed as an application, which can be embedded into an SDN facility. Recall that RADAR allows COTS switches to report flow counter information of *a subset* of network-wide flows to the controller. To achieve this, switches need to know which flow information is required by the controller. The key challenge is to design an online and unified detection architecture which is able to capture attacks based on a limited number of counters for a wide range of attacks without any prior knowledge. In principle, this is possible because we have the following important observation.

Key Observation: Given the flooding victims of a particular DDoS attack, any attack flow always correlates with the attack traffic aggregated on the victims irrespective of flow dynamicity.

Although flows from any individual client can be very small, the aggregated traffic from groups of clients at the targeted links is not, which is the attack goal. Aggregated traffic allows us to capture the traffic anomalies. Further, the detected anomalies notify the controller to dynamically collect and analyze the traffic correlating with the aggregated traffic, and thus the controller can detect the attack traffic. In practice, in order to have such online architecture to capture the correlation, we develop three main components in RADAR, i.e., the collector, the detector, and the locator, which adaptively interact with each other to collect a limited number of required flow counters (see Figure 3).

- **RADAR collector** receives collection rules from the detector and the locator depending on what type of attack is of interest, and instructs the switches to collect their interested flows from network-wide traffic, which is triggered by the aggregated traffic.
- **RADAR detector** receives flow statistics from the collector, and each modular in the detector performs the correlation analysis of its suspicious flows to adaptively generate fine-grained collection rules and detect attacks.
- **RADAR locator** receives signals from the detector, and then adaptively generates collection rules to collect the suspicious flows. It performs adaptive correlation analysis between single suspicious flow and flows aggregated in victims to locate and throttle attack traffic.

Note that, RADAR requires switches to be deployed close to the victims. RADAR captures the attack flows by analyzing the correlation between the flow captured at various locations and the aggregated traffic close to the victim. Since the Crossfire attack is one of the most sophisticated DDoS attacks, in the rest of this section, we will use it as a typical case to illustrate how RADAR detector works. Under Crossfire attacks, it is not easy to identify attack packets even if the switches close to the attack sources can capture the increase of flows. In particular, by interactions among the three components, RADAR adaptively correlates traffic captured at different locations such that it detects various attacks without any prior knowledge of the attack flows.

The approach of Crossfire detection can be simplified to detect other attacks. For instance, SYN flooding can be captured by analyzing the ratio of the number of SYN packets to that

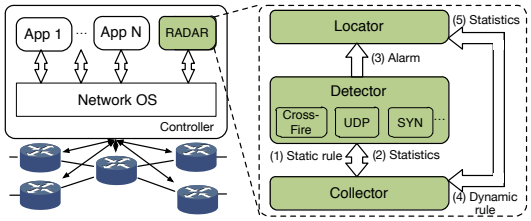

Fig. 3. The architecture of RADAR

TABLE II
NOTATIONS USED IN ALGORITHMS

Notation	Meaning
i	the round of the statistic retrieval
$root$	the root pointer of the trie
$L_d, \mathcal{P}_d$	a set of victim links and victim paths, respectively
$F_q[]$	an array with each element $F_q[l]$ being the change frequency of link l
F_s	a set of possible tuples $\langle f_j, c_{f_j} \rangle$ collected by switch s, with the first element of a tuple being a suspicious flow f_j and the second element being the counter number of this flow f_j
P_s	a set of possible tuples $\langle p_j, c_{p_j} \rangle$ collected by switch s, with the first element of a tuple being a port p_j delivering suspicious flows and the second element being the counter number of all flows at this port p_j
$C1_s[][]$	a matrix with each element $C1_s[j][i]$ being the counter number of flow f_j in round i at switch s
$C2_s[][]$	a matrix with each element $C2_s[j][i]$ being the counter number of all flows at port p_j in round i at switch s
$R1_s[][]$	a matrix with each element $R1_s[j][i]$ being the rate of the change of the counter number of flow f_j in round i at switch s
$R2_s[][]$	a matrix with each element $R2_s[j][i]$ being the rate of the change of the counter number of all flows at port p_j in round i at switch s
$\aleph[][]$	a matrix of link congestion indicators where an element $\aleph[j][i]$ equals 1 if there is a link congestion at port p_j in round i, or 0 otherwise
$\mathcal{D}[][]$	a matrix of the duration of link congestion, with each element $\mathcal{D}[j][i]$ being the time duration of congestion at port p_j in round i
$\mathcal{T}[][]$	a matrix of beginning time of congestion, with each element $\mathcal{T}[j][i]$ being the first time point when congestion starts at port p_j in round i
$S[]$	an array with each element $S[j]$ being the suspiciousness level of flow f_j
$t[]$	an array with each element $t[j]$ being the score of flow f_j

of ACK packets generated by the same set of hosts, and UDP amplification attacks can be detected by analyzing the statistics of UDP request and response packets.We can build different algorithms with respect to the attacks in the RADAR detector to achieve the goal. The details of detecting the traditional DDoS attacks will be described in Section V.

It is non-trivial to implement such an architecture. Collecting suspicious traffic from a large traffic pool, identifying sophisticated attacks, and accurately locating the attack traffic in realtime, are all of high difficulty. In what follows, we will describe how we realize the key techniques in our design.

IV. DESIGN

In this section, we present the detailed design of the three components in RADAR. The notation used in the algorithms are summarized in Table II.

A. RADAR Collector

RADAR collector receives static and dynamic rules to adaptively retrieve statistics of suspicious flows and maintains them in the controller, such that (1) RADAR detector can identify whether an attack exists, and (2) RADAR locator can identify which flows are the attack traffic. It first passively monitors network flows according to the static rules in a coarse-grained manner[3], and then actively pulls flow statistics if flows are detected as suspicious. Meanwhile, it adaptively issues dynamic flow rules to enforce fine-grained data collection of suspicious flows (see Section IV-B). Here, a flow is defined as a set of data packets forwarded by the same flow entry on OpenFlow switches. It is treated as suspicious if the traffic anomaly appears in the flow. For example, a significant increase in the number of SYN packets indicates possible existence of SYN flooding attacks; a significant increase in the number of UDP request packets indicates possible existence of UDP-based amplification attacks (in this paper, we use DNS amplification attacks as the representative of this type of attacks); while a significant increase of aggregated flows indicates possible existence of flooding attacks (either normal flooding or sophisticated Crossfire attacks). These anomalies can be captured by leveraging OpenFlow group tables [9], which will be detailed later.

Note that, although flows from any individual client in the Crossfire attack are very small, the aggregated traffic from groups of clients at the targeted links are not (by the definition of link flooding, this is the attack goal). Thus, this traffic anomaly composed of aggregated flows can be suspicious. Since the above traffic anomalies can also be triggered by normal flows, we cannot simply treat suspicious flows at this stage as the detection results. Instead, we need to feed them into the actual detector as detailed in the next section. Therefore, RADAR collector can efficiently narrow down and collect suspicious flows by adaptive interactions, and effectively avoid flow rule table overflow incurred by data collection.

Mechanism. In Figure 4, we show that RADAR collector first creates flow rules in SDN switches, which can be generated according to addresses, ports, or any other fields supported by the OpenFlow specification [9]. It utilizes dedicated flow rules to trigger statistics collection. To achieve this in SDN, it leverages OpenFlow group tables attached to the dedicated flow rules with the group type *SELECT* [9] such that packets can be operated by actions defined in different buckets. Each bucket in a *SELECT* group table has an assigned weight, and each packet is sent to a single bucket. We use weighted round robin to distribute packets to different buckets. As shown in Figure 4, two buckets with different weights are set up in a group table to implement two types actions. The bucket assigned with a lower weight (e.g., the weight value of 1% shown in Figure 4) is associated with the action of packet reporting to the controller, while the bucket with a higher weight is associated with the action of normal packet

[3]For example, RADAR uses one static flow rule to monitor all TCP SYN packets, or uses per-port static flow rules to monitor flows received from each port to capture suspicious Crossfire traffic.

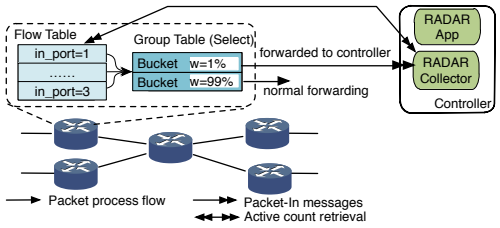

Fig. 4. The collector installs dedicated flow rules and OpenFlow group tables in the switches so that the switches can capture the suspicious flow and report them to the controller.

forwarding. If a flow has more packets counted by the group table, the packets will be more likely captured by the bucket with a lower weight that is associated with the action of packet reporting than other flows. Here, the weight values are set according to network configurations to ensure that traffic increase close to victims will trigger reporting to the controller [9]. Thereby, RADAR collector can capture and understand the suspicious flows by analyzing packet-in packets sent by switches.

After RADAR collector receives packet-in packets, it records and updates the statistics of flows in F_s. Meanwhile, it enables active statistics retrieval of such flows, and periodically pulls the statistics of suspicious flows to update F_s and that of the corresponding ports in P_s. Such active pull actions are triggered by sending request messages on flow statistics [9] to switches. The statistics retrieval intervals τ are set by corresponding attack detection modules according to the types of suspicious flows. The detection modules will verify if attacks exist by evaluating the data retrieved from the collector. If suspicious flows are not regarded as attack traffic, then the statistic retrieval will be deactivated by the collector. The collector notifies the corresponding RADAR modules according to the types of static collection rules. The details of the algorithm can be found in [11]. Note that, we cannot apply sFlow [13] to collect flows since it is unable to monitor particular types of traffic (e.g., TCP packets with specific flags). Since packets generated by sophisticated attacks (e.g., Crossfire attacks) are low-rate attack and concealed in benign traffic, it is not easy for sFlow to capture such packets if it simply samples traffic. However, if sFlow disables sampling, it will incur significant communication overhead.

B. RADAR Detector

Triggered by RADAR collector, RADAR detector identifies attacks by performing correlative analysis of the patterns of suspicious flows collected from different switches, and notifies RADAR locator to locate accurate attack traffic.

Mechanism. RADAR detector utilizes the statistics of suspicious flows reported from switches and the flow forwarding path information learned from the topology database, to detect Crossfire attacks. The attack traffic always shifts among various victim links, bursting and descending alternately. Recall Figure 2, a Crossfire attack can be captured if at least two links in a path are periodically congested and there is at least one link congested at any particular time. A Crossfire attack is successful only when all paths connecting to the victims are congested [18]. However, for efficient detection, RADAR identifies the attacks after capturing one victim path.

To detect Crossfire attacks, RADAR needs to analyze the rate of link utilization changes and the correlation between links and the paths composed of these links. Specially, a Crossfire attack will be identified if (i) the total number of link utilization changes reaches or exceeds a threshold β; (ii) the number of links have been congested in each congested path (according to the forwarding paths) reaches or exceeds a threshold α; and (iii) the congestion duration of a path equals the total congestion duration of links composing this path. Note that, since Crossfire attacks can only be constructed by periodically flooding at least two victim links in a path, we set α to be 2. Also, the traffic shifting frequency cannot be low. Otherwise, the Crossfire will fail [18]. In order to trade off between detection efficiency and accuracy, we set β to be 3. In Section VI, we will show that the impact of β on detection accuracy.

Algorithm. Algorithm 1 shows the pseudo-code of detecting Crossfire attacks. It compares the rate of flow statistic changes on all ports (lines 1-11). If a congestion indicator of a port is changed from 0 to 1, indicating that a round of traffic shifting is detected, then the algorithm computes the link associated with the port according to the topology database (line 3). Then, the algorithm counts the total number of times that traffic shifting happens in this link (or the change frequency of this link, line 4). If this value is larger than β, link l will be included in the set of suspicious links L_d (steps 5-7).

The algorithm computes the congestion duration of all paths constructed by the links in L_d and check if the accumulated duration of congestion caused by such links is equal to the congestion duration of the path (lines 12-34). If it is true, then a Crossfire attack is detected. In details, in order to compute the congestion duration of a path, the algorithm first computes a set $\mathcal{P}$ of all suspicious paths. This is done by investigating all paths constructed by any possible link in the set of all suspicious links, according to flow tables maintained by the controllers (line 13). It then computes the congestion duration of each path if the number of suspicious links in the path is larger than α (line 15). Here, since the congestion duration of links in a path overlaps with each other, to accurately compute the accumulated duration of the links, it sorts the congestion time and eliminates all overlapped period during congestion duration computation (lines 15-25). In an ideal case, if the accumulated link congestion duration $\mathcal{D}'[0]$ computed from all link congestion duration is equal to the path congestion duration $\mathcal{T}'[\text{sizeof}(\mathcal{T}')-1]$ - $\mathcal{T}'[0]$, it indicates a victim path is identified (lines 26-33) and then the algorithm returns *true*. The detector will stop detecting the attack if one victim path is captured. Note that, since flow statistics stored in RADAR collector may not be perfectly accurate, in practice, an attack will be regarded as identified if the difference of duration of accumulated link congestion and that of path congestion is bounded by ϵ, i.e., $\mathcal{D}'[0] \geq (1-\epsilon)(\mathcal{T}'[\text{sizeof}(\mathcal{T}')-1]-\mathcal{T}'[0])$.

Algorithm 1 Crossfire Attack Detection

Input: i; $P_s = \{\langle p_j, c_{p_j}\rangle, \forall j\}$; $\aleph[][]$; $\mathcal{D}[][]$; $\mathcal{T}[][]$;
Output: **true**: attack detected, **false**: otherwise;

```
for ∀{⟨p_j, c_{p_j}⟩} ∈ P_s do
    if (ℵ[p_j][i] == 1) && (ℵ[p_j][i − 1] == 0) then
        l ← get_mapping(p_j);
        Fq[l] ++;
        if (changeFreq[l] ≥ β) then
            L_d ← L_d ∪ l;
        end if
    else
        continue;
    end if
end for
D' ← ∅; T' ← ∅;
P_d ← compute_paths(L_d);
while (path ∈ P_d) do
    if (|path| ≥ α) then
        for (l' ∈ path) do
            p ← get_mapping(l');
            for (j = 0; j++; j < i) do
                if (ℵ[p][i] = 1) then
                    k ← mergesort(T', T[p][j]);
                    D'[k] ← D[p][j];
                end if
            end for
        end for
    end if
    for (j = 0; j++; j < sizeof(T')) do
        if ((T'[j + 1] + D'[j + 1]) > (T'[0] + D'[0])) then
            D'[0] ← T'[j + 1] + D'[j + 1] − T'[0];
        end if
    end for
    if D'[0] ==(T'[sizeof(T') − 1] - T'[0]) then
        return true;
    end if
end while
return false;
```

C. RADAR Locator

We have stated that RADAR detector can detect victim links and paths, but it is still unable to identify which traffic over such links/paths is really attack traffic. RADAR locator is responsible to identify such attack traffic. This is achieved by leveraging adaptive correlation analysis of the rates of flow statistic changes on each link and victims that are detected by RADAR detector. RADAR locator identifies flows as attack traffic if their rates of statistic changes correspond to those of aggregated flows delivered on victim links. Then, it captures the attack traffic associated with the detected attacks and identifies the exact prefixes generating or receiving the traffic. Note that currently RADAR design identifies and tracks flows by using source or destination addresses, since in existing DDoS attacks, either source or destination addresses used in attack traffic are real. In this paper, we use the source address field to illustrate how RADAR locates Crossfire attack traffic, or attackers (/bots). This can be easily extended to any packet fields enabled by OpenFlow specification [9].

Mechanism. Upon receiving an alarm from the detector, RADAR locator is triggered to locate attack traffic according to the attack type identified. Initially, the locator does not have any sense of what are the attack flows. Therefore, it regards all addresses as suspicious source addresses of attack flows (i.e., all flows are suspicious). Such addresses can be categorized according to their prefixes[4]. Then we use correlation analysis to determine which prefix(es) are possible to generate the attack traffic, so we can shrink the size of the set of suspicious flows. By multiple rounds of analysis, we will obtain fine-grained visibility on the flows and finally identify all source addresses of attack traffic.

In order to achieve this, we utilize multibit tries [21][5], each trie representing a particular type of attack. Let us focus on a particular trie for Crossfire. Each node in this trie corresponds to a prefix, recording statistics of flows, whose source addresses match the prefix, and statistics of the ports delivering these flows. To locate Crossfire attack flows, when an attack is identified, RADAR locator first constructs a trie with only the root node corresponds to the default address 000/0, and generates a flow rule where the source address field is set to be the prefix represented by the root node 000/0 to tracks all flows in the network[6]. Note that, 000/0 indicates an address space covering all IP addresses and it can be split into different lengths of address blocks according to a splitting rate. For instance, an address space 000/0 can be split into four blocks with a splitting rate 2, and then four address blocks with prefix length 2 are generated, i.e., 000/2, 064/2, 128/2, and 192/2. Later, we will expand the trie, i.e., 000/0 will be split and children nodes will be created as leaf nodes to attach to the root node (see Figure 5(a)).

RADAR locator will send dynamic flow rules to the collector so that the statistics can be received from the collector and updated periodically. Correspondingly, the statistics associated with nodes in the trie will also be updated. Meanwhile, RADAR locator analyzes the updated statistics associated with each leaf node. If a prefix associated with a leaf node generates flows exhibiting the flow pattern of Crossfire attack, i.e., flows generated by the prefix change proportionally with respect to that on the victims, the prefix associated with the node will further be split to longer prefixes. Each newly generated prefix will be associated with a new child node attached to the original node. Meanwhile, a new flow rule associated with each new prefix is issued to the collector to monitor the corresponding traffic. The splitting procedure repeats until the maximum splitting limit is reached. The round of prefix splitting (or trie expansion) is controlled by the prefix splitting rate. Thereby, by periodically expanding the trie and splitting the prefixes associated with the existing nodes into longer prefixes, RADAR locator captures the attack traffic associated with longer prefix lengths.

Flows whose statistics exhibit the attack pattern associated with leaf nodes will be regarded as attack traffic. The attack traffic will be blocked by issuing OpenFlow meter table rules [9] attached to the locating rules that are matched by the traffic. Ideally, RADAR can locate a host with the prefix with length 32, i.e., an IP address. However, in practice, we observe that the probability that benign traffic matches the locating rules with various specified fields is low as long as the prefix specified in the rules is long enough (e.g., 24). Therefore, we set the maximum prefix length to be 24 in order to achieve a good tradeoff between detection accuracy and detection delay.

[4]In our algorithm, we use a general form of prefix where its length can be any integer between 0 and 32.

[5]Multibit tries are similar to traditional tries, but they allow nodes to have different numbers of children (see Figure 5).

[6]In the rest of this section, flow rules refer to dynamic rules to locate attack flows.

Note that, detected prefixes may still generate benign traffic whose packet fields by chance match the meter table rules even if the prefix length limit is 32. To mitigate the impact on benign traffic, RADAR locator only sets limits for each flow in meter tables instead of directly dropping flows such that it allows bots to generate a very limited number of packets. In Section V, we will discuss a fine-grained approach to throttling attack flows.

Algorithm. Algorithm 2 shows the pseudo-code of locating attack traffic. It first updates the trie with new received statistics (line 1). For each flow, it searches the port that delivers the flow (line 3). If the rate of the change of the current link is higher (or lower) than that in the last statistics retrieval, the suspiciousness of the corresponding prefix $S[f_j]$ increases (or reduces) by the rate of the change (lines 4-7). It computes the depth d of the trie (lines 10) and performs a breadth first search on all nodes in the trie, recording the statistics of prefixes in set $Flist$ (line 12). The locator deactivates the nodes where the flow statistic does not change proportionally with the link statistics and removes them from the set $FList$, so their offspring nodes will not be visited either. Note that such nodes and their offsprings will be activated and visited during later rounds of tree search.

For nodes in $FList$, the algorithm computes the values of ζ-th percentile among flow statistics of nodes in the same level of the trie, i.e., $Count$. Here, ζ-th percentile ensures that flows accidentally matching locating rules will be excluded from further analysis. In our experiments, we observe that the impact of ζ on the locating performance does not vary significantly if $\zeta \in [0.01, 0.1]$. For simplicity, we set ζ to 0.1. Then the algorithm examines the flows associated with each node of that level, If $S[f_j]$ is larger than $Count$, i.e., the flow correlates with the aggregated traffic, the score of flow f_j increases by 1 (line 22). Here, a score of a flow indicates the suspiciousness of the flow. If the score is larger than β, then the flow associated with the prefix will be blocked by issuing a filter rule with the prefix in OpenFlow meter tables [9] (line 24). If the score is less than β and the length of the prefix associated with the flow is shorter than the maximum splitting length, then the prefix will be split again. Meanwhile, a set of new nodes will be created and updated in the trie, and the corresponding locating rules are generated to replace the rules associated with flow f_j (lines 26-27). Note that, scores measure the times of the flow shifting, while the suspiciousness is the total amount of changes in statistics.

Algorithm 2 Locate Attackers

Input: $root$; $F_s = \{\langle f_j, c_{f_j}\rangle, \forall j\}$; $P_s = \{\langle p_j, c_{p_j}\rangle, \forall j\}$; $R1_s[][]$; $R2_s[][]$; $S[]$; t[];
Output: $S[]$;

```
1: update_trie(root, F_s, P_s, R1_s, R2_s);
2: for ∀{⟨f_j, c_{f_j}⟩} ∈ F_s do
3:     p ← locate_port(f_j);
4:     if (R2_s[p][i] > R2_s[p][i − 1]) then
5:         S[f_j] ← S[f_j] + R1_s[f_j][i];
6:     else
7:         S[f_j] ← S[f_j] − R1_s[f_j][i];
8:     end if
9: end for
10: d ← compute_depth(root);
11: for (k = 0; k ≤ d − 1; k + +) do
12:     Flist ← BFS(root, k);
13:     for (∀nd ∈ Flist) do
14:         if (!flow_port_correlate(nd)) then
15:             deactivate_descendants(nd);
16:             remove(nd, Flist);
17:         end if
18:     end for
19:     Count ← calculate_percentile(Flist, ζ);
20:     for (∀nd ∈ Flist && f_j associated with nd) do
21:         if (S[f_j] ≥ Count) then
22:             t[f_j]++;
23:             if (t[f_j] == β) && (get_prefix_len(f_j) > Γ_max) then
24:                 block(f_j);
25:             else if (get_prefix_len(f_j) < Γ_max) then
26:                 split(get_prefix(f_j));
27:                 create_children(nd);
28:             end if
29:         end if
30:     end for
31: end for
32: return S;
```

D. Illustration

Let us recall the example in Figure 2, and use it to illustrate how RADAR detects and locates a Crossfire attack. Assume a static flow rule and a group table are installed in the switch between A and B, where the source address field is set as 0.0.0.0/0 to count all forwarded packets. At a particular time, the hosts with source addresses 32.0.0.1/24 and 240.0.0.1/24 generate traffic to the decoy servers and congest A and B, respectively. The group table receives the statistics for flow rules, and will capture the flows and notify the controller by generating packet-in packets. Thereby, RADAR collector understands that these flows are suspicious. It will periodically pull the statistics of the flows and that of the ports of switches forwarding these flows, and feed the data to RADAR detector.

After receiving the statistics, the RADAR detector starts analyzing the data. As shown in the right part of Figure 2, flows generated by hosts 32.0.0.1/24 and 240.0.0.1/24 match the patterns: (1) the total number of changes on link utilization on A and B exceeds 3; (2) the number of links in the path is 2; and (3) the congestion duration of the path is equal to the total congestion duration of A and B. Thus, the detector sends an alarm to RADAR locator, i.e., a Crossfire attack is detected.

RADAR locator starts locating attack traffic by creating a trie set a root node 000/0 and issuing a flow rule associated the node to the switches that have detected congestion. It will detect that the statistics associated with the trie node exhibit attack pattern. Thereby, prefix 000/0 will split into 2^i blocks where the splitting rate i is set as 2 (see Figure 5(a)). Meanwhile, RADAR locator issues new locating flow rules corresponding to the new prefixes to track the corresponding flows. Hence, flows generated by hosts 32.0.0.1/24 and 240.0.0.1/24 will be monitored by the flow rules with source prefixes 000/2 and 192/2, respectively. After receiving updated statistics, assume the flow generated by host 240.0.0.1/24 first exposes the attack pattern, then the trie will be expanded by splitting prefix 192/2 with the same splitting rate into four prefixes (see Figure 5(b)), and the new rules are issued via RADAR collector to count the corresponding flows. Similarly, if a flow exhibits attack pattern, the trie will be further expanded (see Figure 5(d)). For ease of presentation, we assume that the limit of prefix splitting is 4. In reality, RADAR will eventually determine the prefixes with the maximum length permitted that generate such

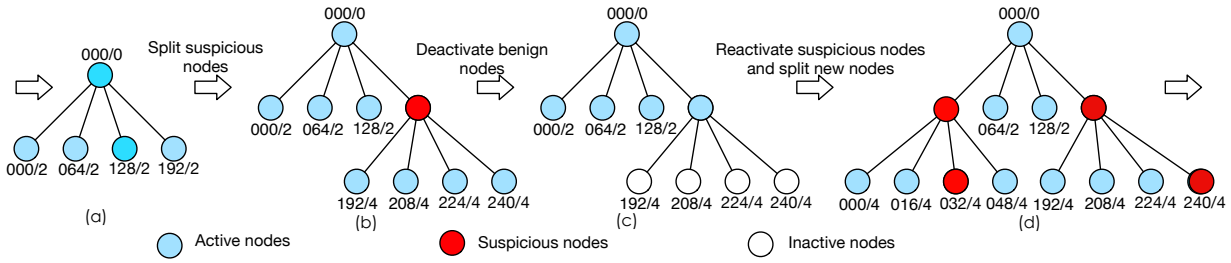

Fig. 5. RADAR locator gradually splits prefixes associated with the nodes in the trie such that it can effectively locate nodes associated with the prefixes that generate the attack traffic: (a) a trie with four split prefixes; (b) node of 192/2 is split into four child nodes since the traffic generated by the prefix exhibits the attack pattern; (c) the new generated child nodes will be inactive if their traffic does not exhibit the attack pattern; (d) node of 000/2 is further split since the traffic associated with the prefix match the attack pattern. It can be applied to track prefixes associated with the victims receiving attack traffic.

attack traffic. RADAR locator issues a meter table RATE:5 and attaches it to the locating rule, which allows flows matching the locating rule to pass five packets per second. Up till now, the attack traffic is identified, located, and throttled.

V. Optimization

In this section, we present some optimizations that we have implemented.

Scalability. One major possible concern of RADAR design is its scalability, i.e., whether it can work in large scale detection. Similar to BROCADE DDoS defense products [12], [26] and existing SDN-based measurement approaches [23], [34], RADAR requires dedicated flow rules to monitor flows. It consumes a small number of flow rules for SYN flooding and DNS amplification detection (see Section VI), and thus the scalability is not a big concern. The main challenge is in sophisticated attacks that may require a large amount of flow rules. In order to prevent rule explosion, we set limits on the number of flow rules (which we refer to as the maximum number of flow rules) that can be used to locate attacks. Upon reaching the limit, we merge nodes in the trie, so that the corresponding flow rules will be removed and spaces are made up for further new splitting. The nodes to be merged are those whose children nodes all have low suspiciousness. The merge mechanism is similar to that in [41]. Note that, intuitively, if we have a reasonably large space for flow rules, then merge will only happen when all children nodes are really safe to remove. Our experiments also validate that such merge mechanism works pretty well in practice. We would point out that it is very difficult to have an accurate theoretic analysis on the tradeoff between rule space and rule deletion safety, and we will leave it in our future work.

We have to emphasize that not all flow rules can be merged, otherwise it will incur high false positive. Another important optimization we have implemented in RADAR is a distributed flow rule placement strategy, i.e., for each flow rule of a monitored flow, we implement it in only one particular switch in the packet forwarding path, rather than installing it in all switches along the path. This mechanism enables that various flow rules spread in different switches in the path are non-overlapping and the number of required dedicated flow rules in each switch is significantly reduced. Similar strategies for packet filter placement are deployed on the current Internet [29], which are demonstrated to be effective. In the experiment section, we will verify its effectiveness.

Note that, although RADAR requires extra TCAM consumption, in most cases, RADAR requires a relatively small number of TCAM entries compared to the number of TACM capacities according to our experiment results, in particular under TCP SYN flooding and UDP amplification attacks (see SectionVI). The improvements above can effectively reduce the TCAM consumption incurred by RADAR. Actually, our experimental results show that RADAR can still effectively detect attacks if even the number of assigned flow entries for detection is less than that required for detection. Also, since RADAR requires each switch to report suspicious flows to the controllers, the number of flows captured by each switch is limited, which is demonstrated by our experimental results (see Section VI.)

Specialization for Traditional DDoS Detection. In order to detect traditional DDoS attacks, e.g., SYN flooding and DNS amplification attacks, we can leverage a similar correlation analysis technique discussed in Section IV-B. Different from Crossfire detection, SYN flooding can be detected by computing the ratio of the number of ACK packets to SYN packets during a detection interval. If the number is close to 1, SYN flooding attacks are detected. Also, if the ratio of the average sizes of DNS response packets to that of DNS request packets significantly deviates from a threshold, DNS amplification attacks are detected. Note that, according to OpenFlow specification [9], OpenFlow switches can count the number of their interested packets (e.g., the number of TCP SYN/ACK packets and the number of DNS request packets) and the total sizes of different types of these packets. Therefore, RADAR can directly retrieve the required numbers from the corresponding switches. In order to detect TCP SYN flooding and DNS amplification attacks, RADAR only requires two dedicated flow rules to monitor each type of flows with the particular port or flag information. For example, RADAR uses two rules to monitor TCP packets with SYN and SYN/ACK flags, respectively, and uses two rules to monitor DNS packets with source port 53 and destination port 53, respectively. Therefore, RADAR does not require more granular flow rules in monitoring or blocking attack flows. Note that, attack packets may be with fake source or destination addresses. Thus, it is not possible to track the exact attack packets by

using more granular flow rules. Instead, we use the max-min fairness technique to achieve fine-grained packet dropping and throttle attack packets.

Note that, as we have discussed before, RADAR locator is designed to locate various types of attacks by issuing different types of locating rules, e.g., locating rules specified with DNS protocol number are used to detect DNS amplification attacks. By specializing it to locate such SYN flooding and DNS amplification attacks, RADAR is specialized to work for traditional DDoS attacks. We will illustrate this functionality in our experiments.

Parellalization of Detecting and Locating Attacks. In order to reduce the delay of locating attackers, RADAR combines phases 2 and 3, i.e., detecting attacks and locating attack traffic, so as to locate different attackers or victims during traffic correlation analysis. In other words, RADAR builds the trie and splits nodes in advance, and periodically updates the trie during detecting attacks.

Fine-Grained Packet Dropping. In the basic RADAR design, we leverage OpenFlow meter tables to throttle attack traffic according to its addresses. However, the meter tables may falsely drop the benign traffic if such traffic matches the features of attack traffic by accident. To address this issue, RADAR locator is extended to incorporate a port-based max-min fairness technique to drop the attack traffic. It is different from the traditional max-min fairness technique. It enforces packet throttling strategies according to all statistics aggregated on ports. RADAR locator installs a traffic throttle by enforcing a meter table at each OpenFlow port. For those packets that match the features of the attack traffic, e.g., traffic whose source and destination addresses match that of Crossfire attack traffic, the throttle will limit the rate of such packets to be forwarded by the port at switches. Traffic that exceeds the rate limit will be dropped. The underlying rationale is that most packets matching the features of the attack traffic are malicious and they can be regarded as the attack traffic if the number exceeds the throttle at each port. Thereby, port-based max-min fairness significantly increases the accuracy of dropping attack flows compared with the traditional max-min fairness.

VI. Performance Evaluation

In this section, we evaluate the performance and overhead of RADAR with Mininet and hardware testbed experiments with real CAIDA trace [2], and simulation-based experiments with the real trace.

A. Prototype and Experiment Setup

We implement RADAR in an open source controller, Floodlight [5]. Currently we provide detection modules for link flooding (including Crossfire), SYN flooding, and DNS amplification attacks. In particular, we use a real CAIDA trace [2] and replay it as background traffic. The traces were collected on a backbone link of a Tier-1 ISP in Chicago on February 2015. We conduct microbenchmark and macrobenchmark experiments to evaluate the performance of RADAR. The microbenchmark experiments aim to study the effectiveness and performance of RADAR with different parameters under various attacks, while the macrobenchmark experiments aim to demonstrate the feasibility of RADAR in hardware OpenFlow switches and evaluate its performance in a large scale.

We measure the effectiveness of RADAR and its overhead by constructing different DDoS attacks: (1) Crossfire attack by allowing different bots to send stealthy and low rate attack traffic according to the description of the attack [18]; (2) SYN flooding attack by sending only SYN packets with spoofed IP address; (3) DNS amplification attack by sending DNS requests with fake source IP addresses.

Microbenchmark experiments: We use Mininet to perform various experiments. In Mininet experiments, we construct various attack traffic with the real Internet 2 topology [6] on Ubuntu server 14.04 with an 8-core 2.8 GHZ Xeon CPU, 128GB RAM. We assign different numbers of bots to the switches so as to generate different attack traffic. Due to limitation of computation resources, we are unable to set up a large number of bots. Here, we use two strategies to select bot addresses and use these addresses to generate attack traffic: (1) *HN strategy*: We randomly select addresses in different class C prefixes as bots addresses and randomly choose different numbers of bot nodes in the prefixes. The number of bot nodes in a prefix varies between 0 and 254. According to existing studies, real bot distributions exhibit a high degree of clustering [18], so these addresses can be aggregated to a small number of prefixes. (2) *SA strategy*: We randomly select class C prefixes and select different numbers of addresses in each class C prefix that are not aggregated at all. The reason is that this strategy introduces the heaviest overhead. In both strategies, the number of class C networks is set to be 10, 25, 50, 75, and 100, respectively. The total number of bots varies between 1 and 19,125. For simplicity, in the following experiments, we use bots to indicate a set of bots with the same prefix.

Macrobenchmark experiments: We use real OpenFlow testbed experiments with real traces and large scale trace-based simulations to evaluate the performance. We replay the CAIDA trace as background traffic and construct the Crossfire attack traffic. In real testbed experiments, we use two Pica8 P-3297 OpenFlow switches [10] to forward all traffic including attack traffic and measure if RADAR with these two switches can effectively capture and defend against the attacks. For simplicity, we use HN strategies to select bot addresses and use them to generate attack traffic. Since the Pica8 P-3297 switches cannot support a large number of flow rules, the number of class C prefixes is set to 50, 100, 500, 1000, and the number of class C prefixes in the large scale simulations is set to be 1,000, 2,000, 5,000, 10,000, 50,000, and 100,000, respectively. Moreover, note that the Ethernet speed of hardware switches in our testbed experiments is 1 Gbps and the packet rate in the CAIDA trace is up to 10 Gbps. We cannot directly replay the trace, but replay 10% of the data per second by using TCPreplay. We directly replay the trace in the simulation-based experiments. As we observe, SYN flooding and DNS amplification attacks can be accurately captured by RADAR. We do not present the results in this paper due to page limit. Note that, according

to our experimental results, the size of the networks does not impact the detection delays of RADAR but the overhead of RADAR. A small network topology incurs more overhead on switches since they require more flow rules in each switch to track attack flows. Fewer switches, more overhead on them. Therefore, in our experiments, we evaluate the performance of RADAR in the extreme case, i.e., the network only includes two switches.

We use the following metrics to measure the performance.

- **Accuracy:** We use the metrics of true positive rate (TPR), false positive rate (FPR), and mitigation rate (MTR) to measure the detection accuracy. In particular, we use MTR to measure the percentage of blocked attack traffic volume in all attack traffic flows.
- **Delay:** Delay refers to the time duration from the beginning of the attack to the time point it is located.
- **Overhead:** We measure the number of extra flow tables used to detect attacks under various attack scenarios with and without flow rule limits. Since the number of static rules is very small, we mainly evaluate the consumption of dynamic rules.

B. Microbenchmark Experiments

We use Mininet experiments to conduct experiments with a real network topology, i.e., the Internet 2 topology. Our goal is to measure the impact of the number of bots and different settings of RADAR on the accuracy and overhead of the detection.

Experiment 1: Impact of ζ on accuracy. We evaluate detection accuracy when ζ varies. Figure 6 and 7 show the impact of ζ on TPR under HN and SA strategies, respectively. We observe that, when ζ increases, TPR decreases under various prefix splitting rates. In particular, under SA strategy, when ζ is larger, the number of searched trie nodes reduces and then scores of the nodes associated with attack flows are much smaller than real scores. Therefore most of the attack flows are falsely excluded during traffic locating. However, RADAR achieves more than 95% TPR if ζ is set less than 0.01 under both strategies when the splitting rates are set to be 1, 2, and 4. TPR with splitting rate 8 under SA strategies is almost zero because RADAR does not have enough flow rules to locate new captured suspicious flows.

We measure the impact of ζ on FPR. We observe that FPR under the two strategies is rather low. Figure 8 shows FPR under SA strategy. The worst FPR is around 5% when $\zeta = 0$ since benign traffic with traffic burst is counted by mistake. Here, ζ can be any real number between 0 and 0.1. We do not observe FPR when ζ is set to other values in this experiment. Moreover, we also measure MTR with various ζ values (see Figures 9). The distributions of MTR under various splitting rates is very similar to that of TPR since RADAR can effectively drop detected attack flows. MTR achieves more than 98% when ζ is less than 0.01. In the following experiments, we set ζ to be 0.01.

Experiment 2: Impact of splitting rates on accuracy. We measure the impact of splitting rates under two bot distribution strategies. TPR under the two strategies is very similar (see Figures 10 and 11). The only difference is that using SA strategy, RADAR achieves much worst TPR when the splitting rate is set as 8. The reason is similar to what we have stated above. In the following experiments, we will not include the experimental results with splitting rate being 8. FPR is negligible, and MTR is similar to TPR. We do not repeat the results here.

Experiment 3: Impact of splitting rates on overhead. In this experiment, we evaluate the number of flow rules consumed for locating bots. As shown in Figure 12, under HN strategy, on average, RADAR consumes less than 1,000 flow rule entries under attacks from various numbers of bots. However, RADAR consumes much more flow entries when using SA strategy (see Figure 13). In particular, for attacks with 100 prefixes, it consumes more than 5,000 flow entries. The number can be constrained by setting a limit, while not impacting the accuracy (see Experiment 7). Actually, the number of flow rule entries in commodity OpenFlow switches can be up to more than 160,000 [7]. Therefore, we believe 5,000 flow entries for Crossfire detection is acceptable. We will evaluate the overhead with DDoS detection at a larger scale in the macrobenchmark experiments.

We also evaluate the communication overhead of collecting statistics. As shown in Figure 14, under Crossfire attacks, initially the sizes of packets delivering the statistics increase over time since RADAR gradually finds more and more suspicious flows. At the 120th second, RADAR detects all attack traffic. But it still enables active statistics retrieval to ensure that no more bots can be detected. After the 200th second, the number of packets starts decreasing. The most incurred communication overhead is only 0.25 MB/s. In fact, we observe RADAR incurs similar overhead under various attack scenarios. Therefore, we can conclude the communication overhead does not exacerbate network performance.

Experiment 4: Detection delay under attacks with different settings. In this experiment, we evaluate the detection delay when the splitting rate set to be 4 and the maximum number of flow rules is set to be 5,000, where SA strategy is enforced. As we observe in Figure 15, the detection delay is stable when the numbers of bots vary. More than 90% attack traffic is detected within 90 seconds no matter how many bots participate in the attacks. Since most of the attack traffic can be correctly damped by RADAR, RADAR can effectively throttle the traffic within 90 seconds. Actually, as we observed in Figure 15, RADAR detects more than 50% attack traffic within 60 seconds. Therefore, the impact of the attack is eliminated or significantly mitigated within one minute.

Also, we measure detection delay where the maximum number of flow rules is set to be a smaller value, i.e., 2,000. We choose three duration of shifting the target links as 10, 20, and 30 seconds. We observe that if a small number of bots participate in the attack with 10 seconds link shifting duration, and RADAR detects most of the attack traffic within 100 seconds that is the product of the link shifting duration and the shifting round (see Figure 16). The delays are reasonable since it takes time for RADAR to split flow rules to spot attack traffic and confirm the attacks by at least β rounds of link shifting. However, when the number of bots reaches 100,

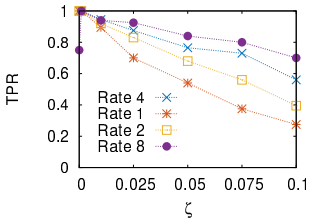

Fig. 6. Experiment 1: Impact of ζ on TPR using HN.

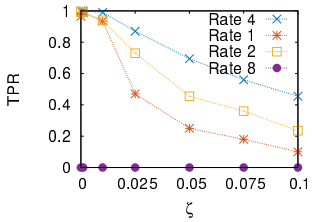

Fig. 7. Experiment 1: Impact of ζ on TPR using SA.

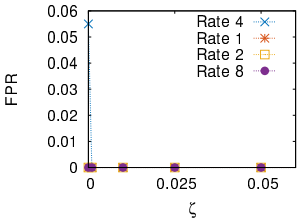

Fig. 8. Experiment 1: Impact of ζ on FPR using SA.

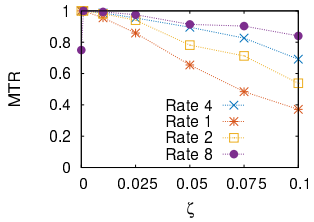

Fig. 9. Experiment 1: Impact of ζ on MTR using HN.

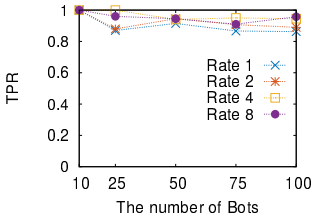

Fig. 10. Experiment 2: Impact of splitting rate using HN.

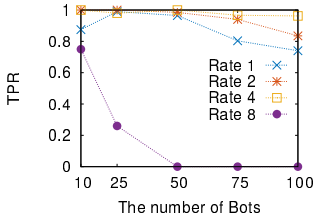

Fig. 11. Experiment 2: Impact of splitting rate using SA.

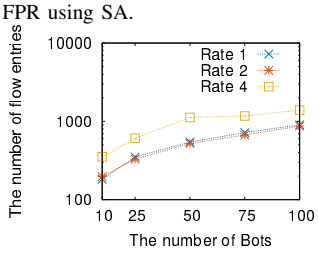

Fig. 12. Experiment 3: Flow rule consumption using HN.

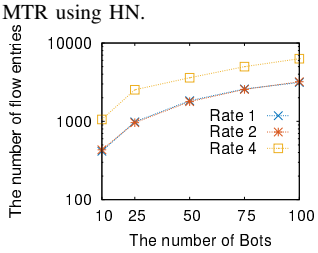

Fig. 13. Experiment 3: Flow rule consumption using SA.

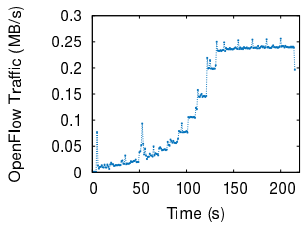

Fig. 14. Experiment 3: Communication overhead of statistics collection.

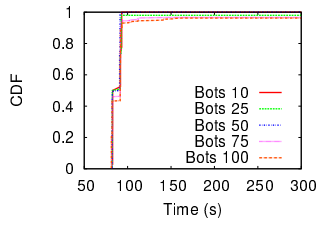

Fig. 15. Experiment 4: CDF of detection delays with splitting rate being 4.

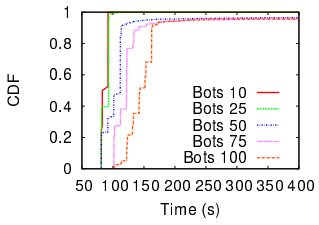

Fig. 16. Experiment 4: CDF of detection delays with 2000 flow rules.

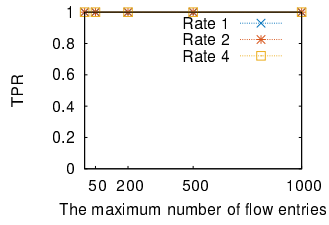

Fig. 17. Experiment 5: TPR under the DNS amplification attacks.

it needs to take around 160 seconds to detect 90% of attack traffic. The reason is obvious: more flow rules can monitor and locate more attack flows. The detection delays increase with respect to the link shifting duration that are controlled by the attackers. However, the link shifting duration will not be very long. Otherwise, the attacks will be captured and treated as the traditional flooding attacks. RADAR can drop the detected the attack traffic in realtime. Therefore, we can conclude that RADAR can effectively defend against and throttle DDoS attacks in realtime, in particular, Crossfire like sophisticated DDoS attacks.

Experiment 5: Detection overhead and delay under traditional DDoS attacks. We measure the detection delay under SYN flooding attacks and UDP-based amplification attacks (with nine reflectors) with 100 bots. Here, we use DNS amplification attacks as a representative of the UDP-based amplification attacks. Figure 17 shows that RADAR achieves 100% TPR with various splitting rates. In particular, it only requires at most 50 flow rules to detect all attack flow. We observe a similar result in SYN flooding detection. Figure 18 and 19 illustrate the delays of SYN flooding and DNS amplification detection, respectively. Since RADAR can easily detect the attack traffic by correlation analysis, it is not surprising that it only takes around 15 seconds to detect all attack traffic, which is not impacted by the maximum number of flow rules. Therefore, we can conclude that RADAR can efficiently detect the traditional DDoS attacks above with a very small overhead.

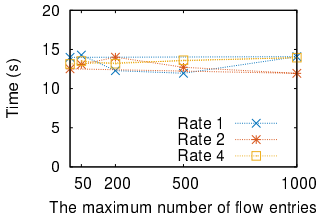

Fig. 18. Experiment 5: Delays of detecting SYN flooding attacks with different rates.

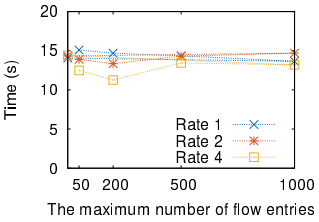

Fig. 19. Experiment 5: Delays of detecting amplification attacks with different rates.

C. Macrobenchmark Experiments

Now we run real testbed experiments and real-trace based simulations to conduct macrobenchmark experiments. Our goal is to demonstrate the feasibility of RADAR with real hardware OpenFlow switches and measure the impact of different settings of RADAR on attack detection accuracy and overhead at a large scale.

Experiment 6: Impact of the maximum number of flow entries on accuracy on the testbed. In this experiment, we evaluate the detection accuracy under attacks from different numbers of bots. As shown in Figure 20, TPR is impacted by the number of consumed flow rules. When the number of flow rules reaches 2,000, RADAR achieves more than 85% TPR. We do not observe any FPR no matter how many flow rules are used to detect attacks (see Figure 21). Figure 22 shows the impact on MTR. After detecting bots, RADAR can accurately capture the attack traffic from the bots. Since some bots generate more attack traffic than other uncaptured bots,

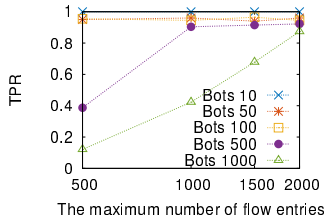

Fig. 20. Experiment 6: Impact of flow rules on TPR on testbed.

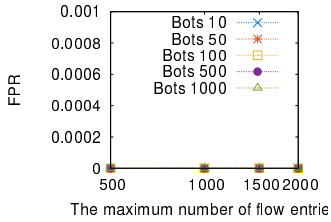

Fig. 21. Experiment 6: Impact of flow rules on FPR on testbed.

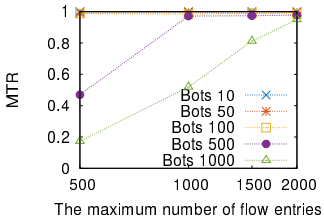

Fig. 22. Experiment 6: Impact of flow rules on MTR on testbed.

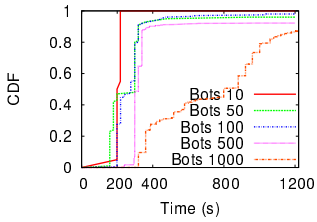

Fig. 23. Experiment 7: CDF of detection delays on testbed.

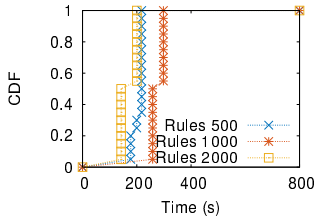

Fig. 24. Experiment 7: CDF of detection rate with 10 bots on testbed.

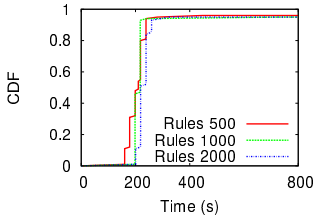

Fig. 25. Experiment 7: CDF of detection rate with 50 bots on testbed.

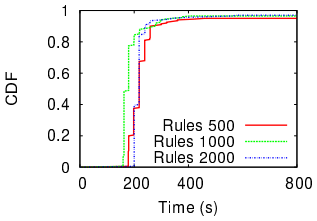

Fig. 26. Experiment 7: CDF of detection rate with 100 bots on testbed.

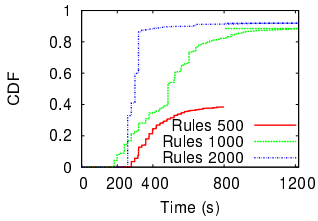

Fig. 27. Experiment 7: CDF of detection rate with 500 bots on testbed.

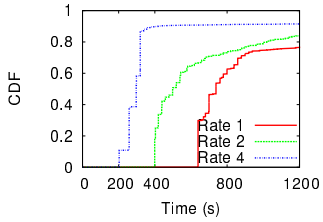

Fig. 28. Experiment 8: CDF of detection delays on testbed.

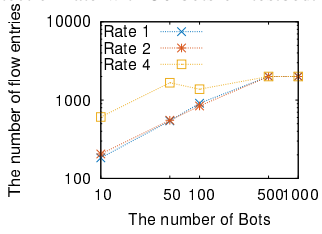

Fig. 29. Experiment 9: The consumed flow rules on testbed.

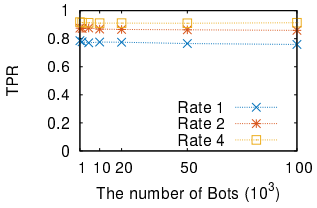

Fig. 30. Experiment 10: Impact of splitting rate in large scale detection.

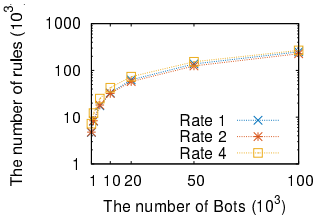

Fig. 31. Experiment 11: Flow rule consumption in large scale detection.

MTR is higher than TPR. In particular, MTR almost reaches 100% even though around 10% bots are not captured.

Experiment 7: Detection delay under attacks with different settings on the testbed. We measure the detection delays under the attacks from different numbers of bots with different flow rules. In this experiment, we set the link shifting duration to be 20 seconds. The detection delays are impacted by the duration of link shifting in the Crossfire attacks, and the detection delays are proportional to the link shifting duration. In other words, most of the bots are captured within around 200 seconds no matter how many bots are involved to construct the attacks (see Figure 23). We also evaluate the detection delays with different number of flow rules that are used to detect attacks. As illustrated in Figure 24, 25, 26, and 27, under the attacks from different numbers of bots, the number of flow rules does not significantly impact the delay rates, though the detection delays slightly vary. Moreover, the deviations among different numbers of flow rules are not so large. Note that, since under the attacks from 500 bots, as we discussed above, the detection delays will be longer if the used flow rules are less than 2000. RADAR cannot track attack traffic from 500 bots at the same time if it is only allowed to use 500 flow rules.

Experiment 8: Detection delay with a different splitting rate. In this experiment, we evaluate the detection delay with respect to different splitting rates. RADAR captures more than 85% bots with different splitting rates. Figure 28 shows CDF of detection delays under the attacks from 500 bots. Since it takes more time to split flow rules and capture bots when the splitting rate is set to 1 or 2, the detection delays with these two splitting rates are slower than that when the splitting rate is equal to 4. Therefore, in order to quickly capture bots and throttle DDoS attacks, it would be better to reserve more flow rule space for RADAR. However, as we discussed above, both can still successfully capture most of the bots no matter which splitting rates RADAR uses.

Experiment 9: Detection overhead on the testbed. Since we observe that the communication overhead incurred by RADAR in microbenmarck experiments is relatively stable and small (see Section VI-B), in this experiment, we focus on evaluating the overhead of flow rules. As illustrated in Figure 29, the required flow rules for DDoS detection is not proportional to the increase in the number of bots. The increase in the number of the flow rules is slower than that of the number of bots. In Experiment 6, we will evaluate the detection overhead under large scale attacks.

Experiment 10: Detection accuracy with various splitting rates in large scale detection. In this experiment, we measure the detection accuracy when the number of bots varies. Figure 30 shows that under various splitting rates, on average RADAR achieves more than 80% bots. In particular, when the splitting rate is 4, RADAR detects more than 90% bots. Note that here each bot indicates a set of bots, where the number of detected bots can reach 25 million. Therefore, RADAR is still effective when Crossfire detection is on a large scale.

Experiment 11: Impact of splitting rate on overhead in large scale detection. We observe when the number of bots increases, the required flow rules increase as well (see

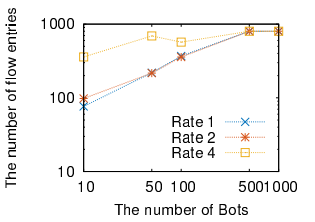

Fig. 32. Experiment 12: Flow rule consumption on testbed.

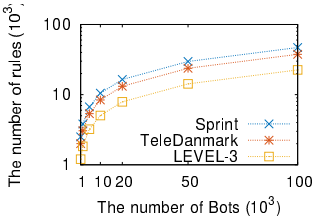

Fig. 33. Experiment 12: Flow rule of each switch in large scale networks.

Figure 31). When the number of bots reaches 10,000, the number of rules is more than 10,000. It seems to be a large value, but note that the increase in the number of consumed flow rules is slower than that of the number of bots, so our mechanism is scalable. In particular, Crossfire attacks are mostly constructed in the Internet backbone [18] that appears to be over-provisioned with high performance switches. The switches should be able to absorb the increase of flow rules. Moreover, Crossfire detection is the extreme case for DDoS detection, and we considered an extreme case where there are 10,000 bots, so the consumed flow rules should be much more than normal cases. Overall, the cost of our mechanism is reasonable. We will evaluate the overhead of each switch with real topologies.

Experiment 12: RADAR Scalability. Finally, we evaluate the scalability of RADAR in the experiments. Since we cannot construct complete topologies of large-scale networks in our experiments, we use measured average path lengths in different ISP networks [25] to compute the consumed flow rules in each switch. We compute average flow rule consumption in each switch by dividing required flow rules among switches in the path. Figure 32 illustrates the average number of flow rule in each switch on our testbed. We observe that the number of flow rules required in each switch is far below 1K. Moreover, we use three typical ISP networks (i.e., Sprint, TeleDanmark, and Level-3 networks) to validate scalability in large-scale detection. The average path lengths of these networks are 12, 15, 25, respectively. As shown in Figure 33, on average, the flow rules required in each switch in these networks are around 6.9K, 5.5K, and 3.3K, respectively, when the split rate is set to be 4. The number of flow rules will be much smaller if the split rate is less than 4. Therefore, the required number of flow rules are significantly reduced when using distributed flow rule installation, and such cost is affordable for real-world commercial COTS switches (e.g., [7]). In particular, these flow rules are still able to detect and throttle very large-scale DDoS attacks by a botnet composed of more than 100K bots, e.g., the recent DDoS attacks on Dyn DNS services [4]. In future work, we will investigate more optimal flow rule placement such that the number of flow rules can be further reduced.

Remark: As shown in the experiment results above, RADAR can effectively detect attacks by correlation analysis even if each attack flow is very small. Specially, RADAR makes a trade-off between detection delays and overhead.

VII. Related Work

DDoS detection in IP networks has been extensively studied [1], [22] via various approaches, e.g., traceback, pushback, puzzle actions, and profile-based defense. Such approaches are not generally easy to be deployed because they need complicated operations of the data plane and/or collaborations of ISPs. Such approaches, though interesting to academia, are often too complicated for the industry to apply. Instead, current industrial solutions [3] usually rely on expensive hardware appliances, increasing cost and packet forwarding delay. These traditional approaches cannot effectively detect the current sophisticated DDoS attacks that are launched with small and short-lived attack traffic.

Recently, SDN opens new doors to defend against DDoS attacks. A number of researchers have done pioneer and insightful works based SDN or software-defined approaches [15], [23], [32], [33]. In particular, Xu et al. [32] detect DDoS attacks by traffic monitoring in SDN. Fayaz et al. [15] redirect traffic to virtual machines in datacenters to detect DDoS attacks. Yu et al. [33] developed a software-defined measurement framework to implement efficient network measurement. Such measurement frameworks aim to detect heavy hitters/changes with iteratively refined traffic monitoring, and attempts show great potential to better detect DDoS attacks via SDN, while the limitation is that they are unable to capture flows collaborating to construct sophisticated ones like Crossfire.

Up till now, we find three pieces of most closely related work to ours, all using traffic engineering (TE) approaches [17], [19], [20] to detect DDoS attacks, (potentially) including Crossfire attacks. Such pioneer works provide valuable insights and great potential on how to address Crossfire-like attacks. In particular, Lee et al. [19] propose a cooperative traffic engineering approach via an extra communication protocol to detect attack traffic. An obvious overhead is that it needs adoption of self-defined protocols and cooperations of different ISPs (and most likely, extra appliances as well). Christons et al. [20] present an analytic model to capture Crossfire, providing interesting insights on practical mechanisms design, however, the paper itself is analytical based and still far away from practical implementation. Kang et al. [17] leverage SDN to implement efficient traffic engineering for link flooding detection. One obvious cost is that it requires major modifications in switches to implement sketch algorithms so as to capture different flows, and it is unclear how it can be implemented in today's SDN switches; meanwhile, it is specifically for link flooding based attacks, but does not apply for other types like SYN flooding.

Several machine learning based approaches [24], [31] were proposed to identify and throttle attacks in SDN. For instance, Wang et al. [31] leveraged a graphical inference model to detect attack flows. By leveraging SDN, these approaches can effectively block identified attack flows in realtime. These approaches are orthogonal to RADAR. The RADAR detector can utilize these approaches to detect more attacks in SDN.

VIII. Conclusion

We propose RADAR, an architecture aiming to detect various DDoS attacks via adaptive correlation analysis on COTS SDN switches. It does not require any modifications in SDN protocols and switches, nor does it need any extra appliance to detect attacks. RADAR is able to capture and throttle

sophisticated DDoS (e.g., Crossfire) attacks in realtime. We evaluate the performance by experiments based on a real testbed, and demonstrate that RADAR can effectively and efficiently detect various attacks within short delays.

ACKNOWLEDGEMENT

This work was supported in part by the National Key R&D Program of China under Grant No. 2016YFB0800102, the National Natural Science Foundation of China (NSFC) under Grant No. 61572278 and U1736209, and the National Science Foundation (NSF) under Grant No. 1617985, 1642129, and 1700544. We thank Weijie Wu for his constructive comments to improve the paper.

REFERENCES

[1] "Arbor," https://www.arbornetworks.com/.
[2] *CAIDA Passive Monitor: Chicago B*, http://www.caida.org/data/passive/trace_stats/chicago-B/2015/?monitor=20150219-130000.UTC.
[3] *Cloudflare*, https://www.cloudflare.com/ddos.
[4] "Dyn reveals details of complex and sophisticated iot botnet attack," http://www.computerweekly.com/news/450401857/Dyn-reveals-details-of-complex-and-sophisticated-IoT-botnet-attack.
[5] "Floodlight," https://www.projectfloodlight.org/floodlight.
[6] *Internet 2*, http://www.internet2.edu.
[7] "NEC ProgrammableFlow," https://www.necam.com/docs/?id=5ce9b8d9-e3f3-41de-a5c2-6bd7c9b37246.
[8] "Opendaylight," https://www.opendaylight.org/.
[9] *OpenFlow Switch Specification*, https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf.
[10] "Pica8 white box sdn," http://www.pica8.com/.
[11] *RADAR Supplement*, https://www.dropbox.com/s/2l65iprly4klgs6/radar_supplement.pdf?dl=0.
[12] *Real-Time SDN and NFV Analytics for DDoS Mitigation*, https://www.brocade.com/content/dam/common/documents/content-types/datasheet/brocade-interop-sdn-ddos-mitigation-flyer.pdf.
[13] "sFlow," http://www.sflow.org.
[14] P. Bright, *Can a DDoS break the Internet? Sure... just not all of it*, http://arstechnica.com/security/2013/04/can-a-ddos-break-the-internet-sure-just-not-all-of-it/.
[15] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and Elastic DDoS Defense," in *Proc. USENIX Security*, 2015, pp. 817–832.
[16] D. Goodin, *How extorted e-mail provider got back online after crippling DDoS attack*, http://arstechnica.com/security/2015/11/how-extorted-e-mail-provider-got-back-online--after-crippling-ddos-attack.
[17] M. S. Kang, V. D. Gligor, and V. Sekar, "SPIFFY: Inducing Cost-Detectability Tradeoffs for Persistent Link-Flooding Attacks," in *Proc. ISOC NDSS*, 2016.
[18] M. S. Kang, S. B. Lee, and V. D. Gligor, "The Crossfire Attack," in *Proc. IEEE Symposium on Security and Privacy*, 2013, pp. 127–141.
[19] S. B. Lee, M. S. Kang, and V. D. Gligor, "CoDef: Collaborative Defense Against Large-scale Link-flooding Attacks," in *Proc. ACM CoNEXT*, 2013, pp. 417–428.
[20] C. Liaskos, V. Kotronis, and X. Dimitropoulos, "A Novel Framework for Modeling and Mitigating Distributed Link Flooding Attacks," in *Proc. IEEE INFOCOM*, 2016.
[21] D. Medhi and K. Ramasamy, *Network Routing: Algorithms, Protocols, and Architectures*. Morgan Kaufmann Publishers Inc., 2007.
[22] Mirkovic, Jelena and Reiher, Peter, "A taxonomy of DDoS attack and DDoS defense mechanisms," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, 2004.
[23] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: Dynamic Resource Allocation for Software-defined Measurement," in *Proc. ACM SIGCOMM*, 2014, pp. 419–430.
[24] S. Nanda, F. Zafari, C. DeCusatis, E. Wedaa, and B. Yang, "Predicting network attack patterns in SDN using machine learning approach," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, pp. 167–172.
[25] P. Pantazopoulos and I. Stavrakakis, "Low-cost enhancement of the intra-domain internet robustness against intelligent node attacks," in *Proc. IEEE/IFIP DRCN*, 2015, pp. 219–226.
[26] Ram Krishnan and Muhammad Durrani and Peter Phaal, *Real-time SDN Analytics for DDoS mitigation*, http://opennetsummit.org/archives/mar14/site/pdf/2014/sdn-idol/Brocade-SDN-Idol-Proposal.pdf.
[27] R. Rasti, M. Murthy, N. Weaver, and V. Paxson, "Temporal lensing and its application in pulsing denial-of-service attacks," in *Proc. IEEE Symposium on Security and Privacy*, 2015, pp. 187–198.
[28] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *Proc. ISOC NDSS*, 2013.
[29] F. Soldo, A. Markopoulou, and K. Argyraki, "Optimal filtering of source address prefixes: Models and algorithms," in *Proc. INFOCOM*, 2009, pp. 2446–2454.
[30] A. Studer and A. Perrig, "The coremelt attack," in *Proc. ESORICS*, 2009, pp. 37–52.
[31] B. Wang, Y. Zheng, W. Lou, and Y. T. Hou, "Ddos attack protection in the era of cloud computing and software-defined networking," *Computer Networks*, vol. 81, pp. 308–319, 2015.
[32] Y. Xu and Y. Liu, "DDoS attack detection under SDN context," in *Proc. IEEE INFOCOM*, 2016.
[33] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. USENIX NSDI*, 2013, pp. 29–42.
[34] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in *Proc. of CoNEXT*, 2013, pp. 25–30.

Jing Zheng received the M.Eng. degree from Tsinghua University in 2018. His current research interests include SDN and NFV security.

Qi Li received his Ph.D. degree from Tsinghua University. Now he is an associate professor of Graduate School at Shenzhen, Tsinghua University. He has ever worked at ETH Zurich, the University of Texas at San Antonio, The Chinese University of Hong Kong, Chinese Academy of Sciences, and Intel. His research interests are in network and system security, particularly in Internet and cloud security, mobile security, and big data security. He is currently an editorial board member of IEEE TDSC.

Guofei Gu is an associate professor in the Department of Computer Science & Engineering at Texas A&M University. Before joining Texas A&M, he received his Ph.D. degree in Computer Science from the College of Computing, Georgia Tech, in 2008. He is currently directing the SUCCESS (Secure Communication and Computer Systems) Lab at TAMU.

Jiahao Cao received his B.Eng. degree from the Beijing University of Posts and Telecommunications in 2015. Now he is a Ph.D. student at Tsinghua University. His current research interests include SDN and NFV security.

David K. Y. Yau received the B.Sc. degree (first class honors) from the Chinese University of Hong Kong, and the M.S. and Ph.D. degrees from the University of Texas at Austin, TX, USA, all in computer science. He has been Professor at Singapore University of Technology and Design since 2013. Since 2010, he has been a Distinguished Scientist at the Advanced Digital Sciences Center, where he leads the cybersecurity research program. He was an Associate Professor of computer science at Purdue University, West Lafayette, IN, USA. He also serves as Qiushi Chaired Professor at Zhejiang University, China. His research interests include cyber-physical system and network security and privacy, wireless sensor networks, smart grid IT, and quality of service.

Jianping Wu received his Ph.D. degree from Tsinghua University. He is currently a professor in the Depermant of Computer Science, Tsinghua University. He has authored over 200 technical papers in the academic journals and proceedings of international conferences, in the research areas of the network architecture, high-performance routing and switching, protocol testing, and network security. He is a member of the Chinese Academy of Engineering (CAE).