DIDA: Distributed In-Network Defense Architecture Against Amplified Reflection DDoS Attacks

Xin Zhe Khooi[†], Levente Csikor[†], Dinil Mon Divakaran[‡], Min Suk Kang[†]
[†]National University of Singapore, [‡]Trustwave

***Abstract*—With each new DDoS attack potentially becoming a higher intensity attack than the previous ones, current ISP measures of over-provisioning or employing a scrubbing service are becoming ineffective and inefficient. We argue that we need an *in-network* solution (i.e., entirely in the data plane), to detect DDoS attacks, identify the corresponding traffic and mitigate promptly. In this paper, we propose the first distributed in-network defense architecture, DIDA, to cope with the sophisticated *amplified reflection* DDoS (AR-DDoS) attacks. We leverage programmable stateful data planes and efficient data structures and show that it is possible to keep track of per-user connections in an automated and distributed manner without overwhelming the network controller. Building on top of this data, DIDA can easily detect if unsolicited attack packets are sent towards a victim within an ISP network. Once an attack is detected, the routers at the network edge automatically block the malicious sources. We prototype DIDA in P4. Our preliminary experiments show that DIDA can detect and mitigate 99.8% of amplification attacks containing $7,000$ different sources while requiring less than 1% of the memory of current programmable switches.**

***Index Terms*—DDoS, amplification attack, detection, mitigation**

I. Introduction

DDoS attacks are unlike other cyber threats—locally installed security appliances and software patches cannot block and prevent such an attack altogether. To mitigate DDoS attacks, ISPs today either over-provision their networks [1] or employ scrubbing services (on-premise [2] or in the cloud [3]). Since both are expensive, there is a limit to the intensity of attack they can deal with before the resources get overwhelmed; this has been demonstrated recently [4], [5].

Identifying an attack is not trivial for an ISP, especially if it looks completely legit on the wire, as is in the case of *amplified reflection* DDoS (AR-DDoS) attacks. AR-DDoS attacks have become more common of late, with each new attack being launched with higher volume and intensity [4], [5]. In AR-DDoS attacks, an attacker exploits the connectionless nature of the UDP protocol with spoofed requests sent to misconfigured or mismanaged open servers on the Internet, which respond with (amplified) replies to a victim.

The continuous success of these AR-DDoS attacks (e.g., [4], [5]) can be attributed to multiple reasons: *(i)* persistent and easy availability of weapons (i.e., reflectors with poor management), *(ii)* little effort required for an attacker (even a single powerful server is sufficient), and most importantly, *(iii)* the difficulties in distinguishing the attack traffic from benign traffic. There is hardly anything a victim ISP can do to counter *(i)* and *(ii)* effectively. Therefore, based on today's solution of employing a scrubbing service, we elaborate *(iii)*.

Since relying on a scrubbing service only enables the incoming traffic to be scoured (at least, during the time when the network seems to be under attack), it is difficult for the scrubber to identify and distinguish the attack traffic from the benign traffic. This is because a scrubber cannot track connections, hence it will not be able to identify which of the incoming packets (responses) to a particular host had corresponding outgoing packets (requests) from the same host. Even if we consider routing outgoing traffic to the scrubber to overcome this issue, a network-wide complex *tagging mechanism* [2] is required so that the requests and the corresponding responses are processed by the same single VM of the scrubber (otherwise, proper accounting of traffic cannot be achieved).

In addition to rerouting of traffic, a scrubbing service causes additional latency and higher cost for the operator (due to the provisioning of resources to handle the attack). Also, there is a risk of leaking user-related private information, when scrubbing is deployed in the cloud. In general, these disadvantages are likely to remain for any detection and mitigation service that is deployed outside the network.

In this work, we focus on detection and identification of attack traffic (i.e., point *(iii)*, leaving out the first two). Specifically, we come up with a new *distributed in-network* solution for detecting and mitigating AR-DDoS attacks, wherein there is no dependence on any external or third-party appliances. We show that by relying on recently emerged programmable and stateful forwarding appliances (e.g., [6]–[8]) deployed at the edges of an ISP network, a fully distributed solution can be built entirely in the data plane (i.e., in-network), which while being completely free of the disadvantages of a scrubbing service mentioned above, can also provide orders of magnitude faster detection and mitigation of AR-DDoS attacks.

However, to realize a distributed architecture in today's fundamentally centralized networks, we have to overcome the following challenges:

1) *At each individual switch*, to keep track of the requests and responses, we need an efficient algorithm that (i) works within the constraints of the programmable switches (e.g., limited amount of memory and instruc-

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Ltd.

978-1-7281-5684-2/20/$31.00 ©2020 IEEE

tions), and (ii) is independent of the control plane for data structure management (e.g., resetting the counters after a monitoring period) to achieve fast updates and queries.

2) *For network-wide connection tracking*, we need an efficient distributed protocol among the switches involved that only generates negligible control message overhead.
3) *In order to quickly mitigate an attack* (e.g., drop the malicious traffic), the switches at the edge of the network should automatically maintain an Access Control List (ACL), which is traditionally manageable only from the centralized control plane.

To solve *(1)-(3)*, we propose DIDA, a fully automated Distributed In-network Defense Architecture (§III). With DIDA, we show that deploying stateful networking switches by replacing only the border (i.e., peering side) and (customer-facing) access routers of an ISP network is sufficient to accurately keep track of the responses and the requests of a given protocol, say, DNS (§III-A). To keep track of the counts at each individual switch, *(1)* we use Count-Min Sketches (CMS, [9]) as a basis due to its sub-linear space requirements, and *we extend it with the notion of time* to keep all data structure management tasks (e.g., reset counts) entirely in the data plane (§III-D). To resolve *(2)*, *we develop a novel distributed protocol* between the border and the access routers that use the production traffic to share and compare the corresponding counts to reach a consensus about a possible attack (§III-B). Finally, to mitigate an attack in time *(3)*, DIDA *dynamically manages an ACL at each border router*, where the IP addresses of the abused servers are automatically added, again *in a fully distributed way* (§III-C).

We prototype DIDA in P4 [6], and our preliminary evaluations show that it is capable to detect and identify amplified reflection attacks with 99.8% accuracy entirely in the data plane (§IV). Furthermore, DIDA requires significantly less amount of control overhead than a centralized monitoring scheme, can be deployed with a limited number of devices capping the overall expected costs within a manageable range, and it automatically mitigate the attacks in-network without the need for any proprietary additional (scrubbing) device.

II. Background and Threat Model

Stateless Data Planes: The main design concept of software-defined networking (SDN) is to have a centralized controller with all network intelligence to "run" and manage the simple forwarding appliances. However, this hinders the possibility to make simple and fast (routing) decisions based on local states (e.g., mitigate TCP SYN floods, do NAT).

Recently, many approaches have been proposed to make the forwarding appliances stateful, including tracking mechanisms in OpenFlow [10], adapting finite state machines [11], building special purpose co-processing units [12], and proposing new programming languages, compilers and target systems as clean-slate redesigns [6]. Among all these, P4 [6] has become the most attractive approach, since its programmable packet parser enables arbitrarily parsing packet headers and contents, and doing parallel match/action processing giving more freedom for an operator to manage the network.

Traffic Monitoring: Detecting and identifying attack traffic require keeping track of connections; this is a challenging task in the data plane, not only due to the continuously increasing traffic demands and line rates, but also due to the limited features and resources available at the programmable devices (e.g., limited instruction set and memory [6], [13]). Existing works, thus, have focused on optimizing algorithms for space, fast update and query time *at a single switch* [9], [14]. But information gathered at a single location is insufficient for many (security) applications with network-wide demands. Not only AR-DDoS attacks, but others (e.g., super-spreaders, port-scanners) can go undetected if only a minimal subset of a malicious traffic goes through a specific point.

Though network-wide measurement is also studied extensively [15], current approaches rely on the centralized controller, which summarizes and verifies the data collected from the individual switches, and makes the final decisions accordingly. Such a design imposes several challenges including a huge control-plane overhead due to the excess amount of communication between the control and the data plane, highly increased (processing) latency due to distributed sampling, and inaccurate counting (e.g., the same packet may traverse multiple devices causing double counting in the control plane [16]).

In this paper, we show that by relying on a fully distributed approach running *entirely* in the data plane, we can resolve all these issues while also providing much more accurate and faster detection and mitigation against AR-DDoS attacks.

A. Threat Model

There are many misconfigured or mismanaged publicly available servers on the Internet which are often exploited for DDoS attacks [17]. We consider such a threat model. For brevity, we use a DNS-based AR-DDoS attack as a running example, but the proposed defense mechanism can be used against any other similar attack (e.g., those based on NTP, SSDP, Memcached). The attacker's aim is to saturate the network bandwidth of the victim. The attacker resides outside of the targeted network, and is capable of either coordinating globally distributed hosts or using her own system (e.g., high-end server) to send huge amounts of small legitimate but spoofed DNS queries to multiple misconfigured DNS servers on the Internet, which in return will send amplified DNS replies to the victim. Therefore, the victim's network resources will be saturated with an enormous amount of unsolicited DNS responses, eventually resulting in a denial-of-service. For brevity, attack against DIDA itself is considered out of scope.

III. DIDA: Design and development

To access a public service, a client first queries a DNS server to resolve the domain name to an IP address. Depending on the request type (e.g., querying for the `A` records only or all records by `ANY` query), the answer can be orders of magnitude larger in size than the request itself (this amplification factor is exploited in AR-DDoS attacks). Ideally, even if a large

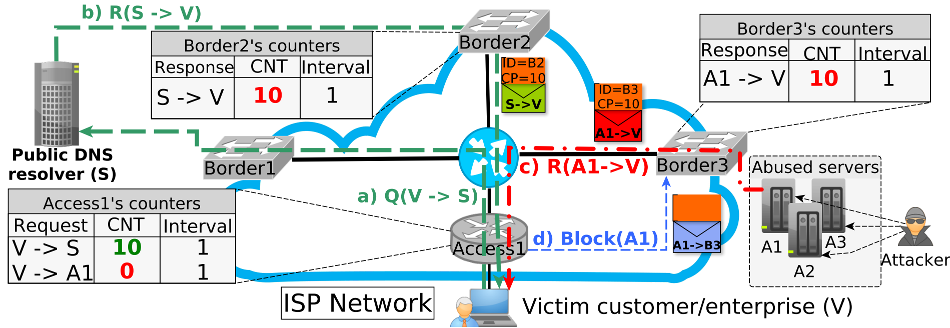

Fig. 1: The detection and mitigation mechanisms in benign and malicious use cases.

response is spread across multiple packets (i.e., due to the EDNS0 extension and DNSSEC, a response can be even 179-times bigger in size than the query [18]) there is only one response for each request and they can be identified as two separate unidirectional flows (§III-B). By counting these flows, we can easily detect whether unsolicited responses are hitting the ISP network. To realize such an architecture, however, we first need to decide which switches across the network should count which flows, and whenever any of the switches encounters a significant number of unsolicited responses how should the communication between the relevant switches be carried out without involving the control plane irrespective of the topology. In DIDA, the way we *detect* an attack, we also *identify* the corresponding packets; henceforth, when we mention detection it inherently includes identification (§III-B).

A. Detection: Where to Count

Naturally, the closer we are to the victim the easier the detection is, while the closer we are to the attack sources, the more efficient the mitigation can be. Thus, we consider the border (i.e., peer-facing) and the access routers (i.e., customer-facing) within an ISP for implementing our solution. There are multiple ways to decide which routers should be involved in which task, with each approach striking a balance between having less routers with more resource needs (e.g., memory), or relying on more routers with less resource constraints.

A requesting (DNS) packet may leave an ISP network at one border router, while the corresponding response packet might enter the network via another border router. This means, counting both the requests and the responses only along the border would require all border routers to constantly share their counts among each other. And this would introduce large number of control messages, which is not desirable during an attack period; besides it can also potentially result in lower accuracy due to asynchronous updates.

On the other hand, counting everything at the access routers does not require extra control messages to be exchanged with others. But once an attack is detected, *all* border routers have to be notified to block the malicious sources as access routers do not know through which border router did the attack packets enter the network. Besides the extra control messages, the redundancy would increase the overall memory footprint as the same malicious sources (i.e., their IP addresses) will occupy valuable slots in each border router's ACL unnecessarily.

Therefore, to make DIDA scalable, e.g., spread the load and minimize the memory footprint as well as the control overhead, the responses are only counted at the border routers, while the access routers keep track of the requests. The algorithms at the border and access routers are given in Alg. 1 and Alg. 2, respectively, and we provide the details below.

B. Detection: How to Count

We describe our solution using an example illustrated in Fig. 1. The figure depicts an ISP network with border and access routers, which we assume to be programmable. The victim (`V`) sends queries to the public DNS server `S` to resolve 20 domains (step *a*), for which it receives responses from server `S` (step *b*). However, since border router `Border2` finds 20 replies within a short time interval to be suspicious (based on the value of the pre-defined threshold τ), it asks the corresponding access router `Access1` to confirm the counts. Since `Access1` has counted 20 requests, the responses are not considered unsolicited, thus `V` is considered not under attack. However, when the responses come from the abused server `A1`, `Access1` would not have the counts of the corresponding requests, therefore it will instruct the corresponding border router (`Border3`) to block any further traffic coming from that malicious source (see details in §III-C).

Detailed mechanism: As an example, a request message from a client with the IP address `1.2.3.4` sent to Google's DNS resolver (located at `8.8.8.8`) can be identified by the following 5-tuple `<ip_proto,src_ip,dst_ip,src_port,dst_port>` as a flow ID: `<17, 1.2.3.4, 8.8.8.8, 54321, 53>`. Similarly, the corresponding response's flow ID looks like: `<17, 8.8.8.8, 1.2.3.4, 53, 54321>`. This means that, for each request (response) the corresponding response (request) can easily be identified as their headers contain the same data in a different order (source and destination fields swapped). Accordingly, in a network-wide setting, by capturing the DNS packets (e.g., via filtering on port 53) one can identify each request (`dst_port=53`) and response (`src_port=53`). While requests are usually an order of magnitude smaller in size than the largest packet size, an amplified response, on the other hand, may span across multiple packets (cf. Sec. III) making a simple packet level-based counting mechanism inefficient. Hence, to accurately identify and count each response only once, in

Algorithm 1 `Border`, *Input:* Ingress packet pkt, ACL $\mathcal{L}$, Count-Min Sketch cms, ID borderId, Suspicious threshold τ

```
1:  flowId ← hash(pkt.src_ip, pkt.dst_ip)
2:  if pkt.ctrl then
3:      L.insert(pkt.src_ip)                     # block source
4:      drop(pkt)
5:  end if
6:  if pkt.src_ip ∈ L then                       # ACL
7:      DROP(pkt)
8:  end if
9:  if pkt.src_port = 53 AND pkt.frag_offset = 0 then
10:     cms.increment_count(flowId, 1)           # count response
11:     if cms.get_count(flowId) > τ then
12:         ctrl.c_count ← cms.get_cnt(flowId)
13:         ctrl.borderId ← borderId
14:         push(ctrl, pkt)                      # add ctrl tag
15:     end if
16: end if
17: FORWARD_PACKET(pkt)
```

Algorithm 2 `Access`, *Input:* Ingress packet pkt, Count-Min Sketch cms, Tolerable range σ

```
1:  flowId ← hash(pkt.src_ip, pkt.dst_ip)
2:  if pkt.ctrl then                             # control message
3:      cCount ← pkt.ctrl.c_count
4:      if (cCount - cms.get_count(flowId)) > σ then
5:          pkt.ctrl ← 0                         # prepare alert
6:          pkt.dst_ip ← GET_BORDER_IP(pkt.ctrl.borderId)
7:      else
8:          pop(ctrl, pkt)                       # remove ctrl tag
9:      end if
10: end if
11: if pkt.dst_port = 53 then
12:     cms.increment_count(flowId, 1)           # count request
13: end if
14: FORWARD_PACKET(pkt)
```

DIDA, after capturing DNS packets, the IP header's `offset` field is also parsed[1], and we only count a response packet if its `offset` value is 0. Then, since the attack detection only requires the true counts of each request and response without identifying whether a flow is a response for a given request, they can be easily identified (and counted) using the source and destination IP addresses only.

Implementation-wise (cf. Alg. 1), when the number of responses at `Border2` hits $\tau = 10$ (line 11), it appends the information as a control header (lines 12-14 in Alg. 1) to the original triggering packet, and forwards that packet in the normal way (line 16). Upon receiving the tagged packet by the access router `Access1` (cf. Alg. 2), it parses out the relevant information (i.e., source and destination IP addresses, and counts) and checks the counts of the corresponding request (line 3-4, step *b* in Fig. 1). If the difference is within a range (σ), the suspicious packet is not considered as part of an attack; hence after removing the tag (line 8), the specific response packet will be forwarded to `V` (line 14). Observe, with σ we tolerate some bias in the counts caused by packet losses, etc.

C. Mitigation

When the number of unsolicited responses from an abused server `A1` (step *c* in Fig. 1) is greater than σ (line 4 in Alg. 2), then `Access1` considers the suspicious responses to be indeed part of an attack and it instructs `Border3` to block the corresponding traffic (line 6). This instruction is implemented by reusing the tagged packet received from `Border3` with the control tag zeroed out, the destination set to `Border3`, while the IP address to be blocked is encoded in the IP header's source IP address field (lines 4-6 in Alg. 2); accordingly, the control tag is used for identification only.

Upon receiving the control packet, `Border3` knows that this packet has to be treated and parsed differently. Accordingly, `Border3` extracts the source IP address, appends it to its in-network ACL, and drop the packet (lines 2-5 in Alg. 1). An adversary outside the ISP could attempt to abuse the algorithm by sending many forged DNS responses making them appear from a legitimate service, e.g., a web service, leading the benign source to be blacklisted. But DIDA can defeat such attacks by using the {source IP address, source port} pair for blocking. Besides, to foil the case where the legitimate service is a DNS service (e.g., 8.8.8.8), we propose to deploy a white list for known trusted recursive resolvers.

Observe that DIDA provides protection even if the abused servers are distributed as the whole operation is only based on the counts of the requests and the corresponding responses irrespective of the number of sources.

D. Overcoming Limitation of the Data Plane

1) Data Structure for Counting: Choosing the right data structure for keeping track of the traversing packets in the data plane heavily depends on the application's needs and the resource constraints of the programmable switches. Considering DIDA's demands and space efficiency, we rely on Count-Min Sketch (CMS) [9] as a basis. For detection, the counts are only relevant for a short interval; therefore, the data structure has to be reset in between. While recent proposals rely on the control plane to reach this end [14], [19], in order to perform traffic monitoring entirely in the data plane, we extend CMS with the notion of time. Briefly, the absolute time within a switch (accessible via the timestamps [7]) is divided into equally long monitoring intervals (e.g., 10 seconds). Each time an element is updated in CMS, the actual monitoring interval is stored next to the counts; whenever the current interval is greater than the stored one, that slot is overwritten (i.e., reset).

2) In-network ACL for Mitigation: Maintaining an ACL in a flow table is a straightforward task from the control plane, but as yet infeasible entirely in today's data plane (lest the main centralized design concept of SDN is violated). Similar to the monitoring task, thus, we have to rely on memory efficient data structures for ACL implementation as packets can also be matched on entries beyond the flow table (e.g., in registers [7]). While probabilistic data structures (e.g. bloom filters) that allow false positives can provide the required

[1] During fragmentation, the first packet has an offset field set to 0, while the consecutive packets for the same message have different values consequently.

accuracy for filtering malicious traffic, they cannot be used for an ACL as it may result in blocking benign traffic.

Therefore, data structures with *false negatives* only, such as hash tables, are appropriate. But traditional hash tables with no collision resolution and low average load factor (i.e., ∼50%) can have high false negative rates resulting in an overall inefficient traffic filter. Hence, to implement an ACL, we adapt Cuckoo Hash Tables (CHT) with four hash functions and thus four logical stages, which is known to achieve ∼91% load factor [20]. Note, a traditional hash table cannot guarantee a constant worst-case lookup time; with CHT, only constant number of buckets are accessed even in the worst case.

IV. Preliminary Evaluations

We prototype DIDA in P4 and emulate a DNS amplification attack in a test environment of bare metal machines similar to Fig. 1. Particularly, `Border3` and `Access1` are realized by two software switches (BMv2, [6]) running on two workstations (Ubuntu 18.04, Intel i7-7700K, 16 GB mem.). The victim (`V`) host (Intel i7-8565U, 16 GB mem.) is connected to `Access1`, communicates with a similar host (say, `U`) outside of the network through `Border2` (not shown in Fig. 1). Lastly, a similar host machine is used to send the malicious traffic towards the victim through `Border3`. We generated an AR-DDoS traffic trace that contains $7,000$ different reflectors (going by real-world statistics [17]) spread out in 25 seconds. For brevity, according to [21] the threshold (τ) along the border is set to 10. Note, however, τ and σ can be dynamically configured by the controller during the operation.

In this environment, we evaluated DIDA and the results are depicted in Fig. 2. First, we run an `iperf` session between `V` and `U` almost fully saturating the available bandwidth (93.7%), then after 5 seconds the attack is launched with half the bandwidth our border router has. As expected, it can be observed that right after the attack is launched, the throughput between `V` and `U` quickly drops to ∼45% of its baseline. Once τ is reached, `Border3` and `Access1` quickly confirm the presence of an attack and initiate the mitigation. The control message overhead of the mitigation is negligible (∼550 kbps at most; see the dashed line in Fig. 2). Within seconds, most of the malicious sources are blocked at `Border3` hindering the attack to further consume the resources. In particular, with a low false negative rate of CHT, 99.8% ($6,985$ out of the $7,000$ IP addresses) of the abused servers are placed in the ACL. Following the attack scenario, the ACL was build up as new sources were identified during the 25s of attack period. Consequently, the useful throughput goes back to its baseline.

Eventually, to handle an attack having $7,000$ different sources, DIDA requires less than 100 KB of memory for counting the responses and the requests at the border and the access switches, respectively, while an additional 50 KB is needed at the border switches for maintaining the ACL (∼1% of the memory of recent programmable switches [22]).

Conclusion. We develop the first distributed in-network defense architecture against AR-DDoS attacks. We show that DIDA detects and mitigates AR-DDoS attacks with very high accuracy while requiring only limited resources. Next, we plan to deploy DIDA on physical devices and perform large-scale experiments against further DDoS scenarios.

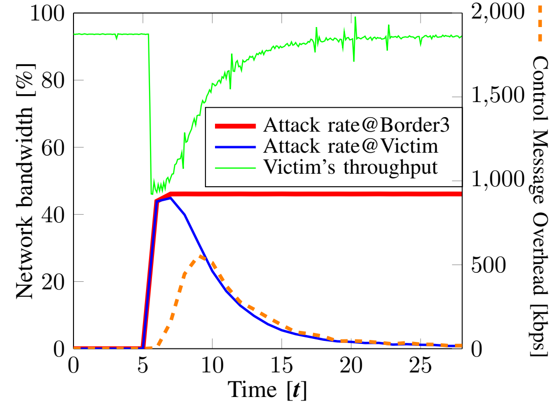

Fig. 2: DIDA in action (mind the secondary y axis)

References

[1] N. Martin, "Overprovisioning VMs may be safe, but it isn't sound," Blog post, https://bit.ly/37oCELJ, May 2014.
[2] S. K. Fayaz *et al.*, "Bohatei: Flexible and elastic ddos defense," in *Proc. of the USENIX Security*, pp. 817–832.
[3] Cisco, "Effective DDoS Mitigation in Distributed Peering Environments," White paper, https://bit.ly/2taak0P, 2018.
[4] N. Woolf, "DDoS attack that disrupted internet was largest of its kind in history, experts say," The Guardian, https://bit.ly/2ZA4Fgm, Oct 2016.
[5] Lily Hay Newman, "GitHub Survived the Biggest DDoS Attack Ever Recorded," Wired, https://bit.ly/2ZBWIY0, Jan 2018.
[6] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 87–95, 2014.
[7] The P4 Architecture Working Group, "Portable Switch Architecture," https://bit.ly/2SNGrhF, Mar 2018.
[8] Broadcom, "Network Programming Language Specification v1.3," https://bit.ly/37jOeYo, Jun 2019.
[9] G. Cormode *et al.*, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, 2005.
[10] Open vSwitch, "OVS Conntrack Tutorial," Tutorial, https://bit.ly/39qeq5j, [Accessed: 2019].
[11] G. Bianchi *et al.*, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM CCR*, vol. 44, no. 2, pp. 44–51, 2014.
[12] S. Zhu *et al.*, "Sdpa: Enhancing stateful forwarding for software-defined networking," in *Proc. of IEEE ICNP*, 2015, pp. 323–333.
[13] H. Wang *et al.*, "Dram-based statistics counter array architecture with performance guarantee," *IEEE ToN*, vol. 20, no. 4, pp. 1040–1053, 2012.
[14] V. Sivaraman *et al.*, "Heavy-hitter detection entirely in the data plane," in *Proc. of SOSR*, 2017, pp. 164–176.
[15] R. Harrison *et al.*, "Network-wide heavy hitter detection with commodity switches," in *SOSR*, March 2018.
[16] B. Ben *et al.*, "Network-wide routing-oblivious heavy hitters," in *Proc. of ANCS*, 2018, pp. 66–73.
[17] Marek Majkowski, "Reflections on reflection (attacks)," CloudFlare Blogpost, https://bit.ly/2SVgUDb, 2017.
[18] R. van Rijswijk-Deij *et al.*, "Dnssec and its potential for ddos attacks: A comprehensive measurement study," in *Proc. of IMC'14*.
[19] M. Yu *et al.*, "Software defined traffic measurement with opensketch," in *Proc. of USENIX NSDI*, ser. nsdi'13, 2013, pp. 29–42.
[20] N. L. Scouarnec, "Cuckoo++ hash tables: High-performance hash tables for networking applications," in *Proc. of ANCS*, 2018, pp. 41–54.
[21] K. Cho, "MAWI Working Group Traffic Archive," https://mawi.wide.ad.jp/mawi/, [Accessed: 2020].
[22] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proc. of ACM SIGCOMM*, 2013, pp. 99–110.