Defending Root DNS Servers Against DDoS Using Layered Defenses

A S M Rizvi, Jelena Mirkovic, John Heidemann, Wesley Hardaker and Robert Story
University of Southern California / Information Sciences Institute

***Abstract*—Distributed Denial-of-Service (DDoS) attacks exhaust resources, leaving a server unavailable to legitimate clients. The Domain Name System (DNS) is a frequent target of DDoS attacks. Since DNS is a critical infrastructure service, protecting it from DoS is imperative. Many prior approaches have focused on specific filters or anti-spoofing techniques to protect generic services. DNS root nameservers are more challenging to protect, since they use fixed IP addresses, serve very diverse clients and requests, receive predominantly UDP traffic that can be spoofed, and must guarantee high quality of service. In this paper we propose a layered DDoS defense for DNS root nameservers. Our defense uses a *library* of defensive filters, which can be optimized for different attack types, with different levels of selectivity. We further propose a method that *automatically and continuously evaluates and selects* the best combination of filters throughout the attack. We show that this layered defense approach provides exceptional protection against all attack types using traces of ten real attacks from a DNS root nameserver. Our automated system can select the best defense within seconds and quickly reduces traffic to the server within a manageable range, while keeping collateral damage lower than 2%. We can handle millions of filtering rules without noticeable operational overhead.**

I. Introduction

Distributed-Denial-of-Service (DDoS) attacks remain a serious problem [5], [34], [49], [16], in spite of decades of research and commercial efforts to curb them. Ongoing Covid-19 pandemic and increased reliance of our society on network services, have further increased opportunities for DDoS attacks. According to the security company F5 Labs, between January 2020 and March 2021, DDoS attacks have increased by 55% [13]. While some large-volume DDoS attacks make front page news (for example, the 1.35 Tb/s [35] attack on Github in Feb. 2018, or 2021 17.2 M requests per second attack, detected by CloudFlare [56]), many more attacks occur daily and disrupt operations of thousands of targets [47], [4].

This paper focuses on protecting the Domain Name System (DNS) root servers against DDoS attacks. The root-DNS service is a high-profile, critical service, and it has been subject to repeated DDoS attacks in the past [50], [1], [2], [31], [42]. In addition, because the DNS root "bootstraps" DNS, it is served on specific IP addresses that cannot be easily modified, thus precluding use of many traditional DDoS defenses that redirect traffic to clouds to distribute load [11].

There are many types of DDoS attacks. Some attacks are conceptually easy to mitigate with firewalls, assuming upstream capacity is sufficient, such as volumetric attacks using junk traffic. Others, such as exploit-based attacks, remain pernicious, but automated patching and safer coding practices offer promise. Most challenging are attacks using legitimate-seeming application traffic, since a *flash-crowd attack* from millions of compromised hosts (also known as layer-7 or application-layer attacks) can resemble a legitimate *flash crowd*, when many legitimate clients access popular content. At DNS root servers, flash crowd attacks would generate excessive DNS queries. Because legitimate clients also generate DNS queries, it is challenging to filter out attack traffic. We focus on mitigation of flash-crowd attacks on DNS root servers.

In flash-crowd attacks, attack traffic often appears identical in content to legitimate traffic. Approaches to handle flash-crowd attacks thus focus on withstanding the attack using cloud-based services [14], [37], [32], [40]. Other approaches aim to separate legitimate from attack clients, e.g., via CAPTCHAs [36], or by using models of typical client behavior [39], [45]. These defenses work poorly for DNS root servers. First, the DNS root operates at small number of fixed IP addresses that cannot be easily changed. This restriction precludes use of traditional defenses that redirect traffic to clouds [11]. Second, DNS traffic to roots is generated by recursive resolvers. Since there is neither direct interaction with a human nor a web-based user interface, CAPTCHAs cannot be interposed. Third, aggressive client identification requires modeling a typical legitimate client. Building a typical client model at roots is challenging, because client request rates vary by five orders of magnitude, from a few queries per day to thousands of queries per second. A model that spans all types of clients can be too permissive, while a model that captures a majority of clients may drop legitimate traffic from large senders. Since most DNS traffic is currently UDP-based, spoofing also is a challenge and spoofers can masquerade as legitimate clients.

In this paper we propose a multi-layer approach to DNS root server defense against DDoS attacks, called *DDiDD* – DDoS Defense in Depth for DNS. Our first contribution is to *propose an automated approach to select the best combination of filters for a given attack*. Selecting from a library of possible filters is important, since different filters are effective against different attacks, and each filter has a different false positive rate, and different operational cost, which precludes its continuous use. *DDiDD* selects the best combination of filters quickly (within 3 s) and continuously re-evaluates filtering effectiveness. When attack traffic changes (e.g., in case of polymorphic attacks), *DDiDD* quickly detects decrease in the filtering effectiveness and re-selects a new, better combination, thereby adjusting to dynamic attacks.

Our second contribution is to propose a novel *wild client filter for DNS*. We provide the first open description and evaluation of a filter that models per-client behavior for DNS clients. Client modeling is widely used to protect web servers [46] where a single model for a "typical" web client suffices. DNS shows a huge range of rates (over 5 orders of magnitude) across clients, so any model that captures this entire range will be too permissive. Instead, we model each client separately during pre-attack periods, and identify as attackers the clients that become more aggressive during attacks. In deployment we combine this filter with anti-spoofing filters to establish trust in client identities.

Our final contribution is to perform *evaluation of each candidate filter*, including our wild resolver filter and six other filters proposed in prior work [43], [51], [23], [33]. While prior work quantified performance of some individual filters for general DDoS attacks [51], [23], [33], and other work qualitatively described commercial deployments (such as Akamai's [43]), we are the first to evaluate each filter quantitatively against real DDoS attacks on a DNS root. We are also the first to propose and evaluate a dynamic multi-filter system for protection of DNS roots against DDoS. Our evaluation uses *real-world attacks and normal traffic taken over 6 years from* B-root, as well as an *adversarial, polymorphic attack* we have synthesized. Our evaluation confirms that no single filter outperforms the others, but together they provide a stable defense against different attack types converging in 3 s or less, with low collateral damage (at most 2%). Our analysis provides evidence for the DNS operators about the importance of having an automated system, and it provides insights about individual filter performance against different types of attacks.

We focus our work on the DNS root server system to meet its unique challenges, but our results also apply to other self-hosted, authoritative DNS servers.

We release the DDoS datasets that we use in this paper [3].

II. Background: DNS and DDoS

The Domain Name System (DNS) is critical Internet infrastructure that maps between human-readable names and resources such as IP addresses. DNS names are hierarchical, with the root, top-level domains (TLDs), like `.com` and `.uk`, and subdomains, like `example.com`. This hierarchy is distributed across many *authoritative nameservers* ("authoritatives" for short). Users usually do not directly query the DNS, but instead use *recursive resolvers* ("recursives" for short) that resolve names on their behalf. Each recursive usually provides service for many users, caching responses to speed access.

For resilience, root zone is served by 13 identifiers, each at a unique, anycasted IPv4 and IPv6 address, served by multiple authoritative servers at multiple geographical points of presence (PoPs). Three aspects make the authoritatives for the DNS root challenging to defend from flash-crowd DDoS attacks. First, most DNS queries use connectionless UDP (not TCP), so it is trivial for an attacker to spoof source IP addresses, making defenses that model client behavior unreliable. Second, root authoritative servers see a huge range of query rates from different recursives—over five orders of magnitude, and huge query content diversity. This variation makes it impossible to produce a single, tight model for a "typical recursive behavior". Third, the DNS root is used to bootstrap the DNS system, and so it operates at fixed IP addresses. Although resolvers refresh this list on startup [26], the list is expected to be mostly static. Deploying new root servers takes months of careful planning. Thus defenses typically used by Content Delivery Networks (CDNs) to shift traffic to different servers (such as [11]) cannot be used to protect DNS root.

Because of its visibility and defensive challenges, the DNS root has been the target of several DDoS attacks. During large, volumetric attacks in 2002 [9], 2007 [21], and 2015 [31], several of the 13 root identifiers showed service degradation (we show other events in §V). Although caching of root contents at recursives reduces the end-user impact of these attacks [30], [27], DNS outages at CDNs have impacted prominent user-facing services [47]. Effective DDoS defense for the DNS root is thus necessary.

III. Related Work

DDoS attacks have been a problem for more than two decades, and many research and commercial defenses have been proposed. This section reviews only those solutions that are closely related to our approaches and to protecting DNS servers against DDoS.

A. Flash-Crowd DDoS Defenses

CAPTCHAs [8], [25] are a popular defense against flash-crowd attacks. They can be used together with other indicators of human user presence, to differentiate between humans and bots. However, DNS queries come from recursives, not directly from human users, so there is no opportunity for a CAPTCHAs intervention. FRADE [46] is a flash-crowd DDoS defense, which builds models of how human users interact with a Web server, including query rates and query content, and uses them to detect bot-generated traffic. FRADE models a *typical client's* behavior. While this works for Web servers, which are browsed by humans, request rates and contents of DNS recursives vary widely. FRADE thus cannot protect DNS servers against DDoS.

Creating an allow-list of known-good clients is suggested in several studies and RFCs [12], [55], [38], [18], [29] for general protection from unwanted traffic. However, the approaches to create a list of known-good recursives for DNS roots have not been described nor evaluated. We evaluate this idea in this paper under the name "unknown recursive filter," in conjunction with hop-count filtering [23], and show that it works well to filter out spoofed attack traffic, but cannot handle attacks that do not use IP spoofing.

Many companies provide DDoS solutions, which may combine signature-based filtering, rate limiting, and traffic distribution using cloud resources and anycast. Such solutions are offered by Akamai [43], [19], Verizon [10], and Cloudflare [54], [17], for example. Since these solutions are proprietary, we cannot compare against them directly. In addition, they often collect traffic with DNS-based redirection

or route announcement (friendly hijacking). Neither of these redirections are possible for root DNS service, which must operate at a fixed IP address, and cannot easily be re-routed.

B. Spoofed Traffic Filtering

Several filters to remove spoofed traffic have been proposed: hop-count filtering [51], [23], [33], route traceback [44], route-based filtering [15], path identifier [53], unknown client filtering [55], [38], and client legitimacy based on network, transport and application layer information [48]. Of these approaches, only hop-count filtering and unknown client filtering can be deployed on or close to the target, and thus show promise for protection of DNS root servers. In hop-count filtering, the filter learns which IP TTL values are used in packets from a given source IP address, and uses this to filter out spoofed packets. The original approach [51] advocates for storing one expected hop-count per source. Mukaddam et al. show that recording a list of possible hop-counts improves the precision of TTL filters [33]. These studies are performed on 10–20 years old traceroute measurements, and they assume reliable inference of TTL filters from established TCP connections. Both Internet topology and application dynamics have since evolved, and DNS traffic is predominantly UDP. Our paper fills this gap, by evaluating hop-count filtering against DDoS with real attack and legitimate traffic, spanning six years and ten attack events.

C. DDoS on DNS

BIND pioneered Response Rate Limiting (RRL) to avoid excessive replies [22] and conserve outgoing network capacity during a volumetric query DDoS. RRL addresses a few misbehaving clients and outgoing amplification attacks, but it does not address well-distributed, volumetric attacks from large botnets.

Akamai uses sophisticated scoring and priority queuing to protect their authoritative DNS servers from floods [19], [43]. Akamai scores queries with the source's expected rate, if the resolver participated in prior attacks, the source's NXDomain fraction, query similarity from that source, and an evaluation of TTL consistency. While two of these scoring approaches are similar to our unknown resolver and wild resolver filters, there are three major differences. First, Akamai provides no quantitative data about how various scoring approaches perform against real attack events. We contribute a careful *quantitative evaluation* of how well different filters work against playback of real attacks. Second, we propose a specific mechanism to select filter combinations, and reevaluate them when necessary. Akamai's approach uses all filters at once to calculate each query score, and Schomp et al. [43] do not describe how the filters interact. Finally, key parts of Akamai's scoring system run inline with processing, requiring high-speed packet handling. Our approach operates in parallel with packet processing, evaluating resolvers to identify potential attackers (or known-good resolvers), simplifying deployment, particularly for lower-end hardware.

Prior work has studied real DDoS events, inferring operator responses using anycast, and suggesting possible anycast options in DNS roots [31]. Recent work has taken this idea further, suggesting that a network playbook can pre-evaluate routing options to shift traffic across anycast sites [40]. Our work complements this line of research, by studying how filters can reduce load at each anycast site.

Finally, several groups have suggested fully distributing the root to all recursives [20], [6], [28]. Such wide replication would greatly reduce the threat of DDoS on the root, but not on other DNS authoritative servers. As a result, on-site defense is still necessary to mitigate DDoS attacks on DNS.

IV. *DDiDD* Design

Our goal is to design an automated system, which continuously evaluates suitability of multiple filters to handle an ongoing DDoS attack on a DNS root server. Our system needs to quickly select the best filter or the combination of filters, reasoning about the projected impact on the attack, the collateral damage from the filter on legitimate recursives' traffic and the operational cost. The system should also be able to adjust its selection as attack changes. Finally, individual filters need to be configured to achieve optimal performance – high effectiveness against attacks they are designed to handle and low collateral damage.

DNS root may also experience a legitimate flash crowd, e.g., when many clients access some popular online content. Due to caching, queries for existing TLDs should not create flash crowd effect, but queries for non-existing TLDs may, since their replies are not cached. *DDiDD* will only activate when excessive queries overwhelm server resources. Unless the server can quickly draft more resources (e.g., through anycast) some queries have to be dropped. Without *DDiDD*, random legitimate queries would be dropped. *DDiDD* (§V) mostly drops queries from sources causing the legitimate flash crowd.

A. Threat Model

We assume that an attacker's goal is to exhaust some key resource at a target by sending legitimate-like requests to the server. Current authoritative servers (including root) do not store state between requests, so the attacker can target CPU resources, incoming bandwidth or outgoing bandwidth. In all cases, the attacker generates more requests than the server can process per second. The attacker may spoof these requests, or they may compromise new or rent existing bots and send non-spoofed requests.

A *spoofing attacker* may spoof at random, or they may choose specific IP addresses to spoof. In some cases, the attacker may choose to spoof addresses of existing, legitimate recursives.

A *non-spoofing attacker* compromises or rents bots to use in the attack. Drafting new bots carries non-negligible cost for the attacker.

The features of *attack requests* depend on the resource that the attack targets. If the targeted resource is CPU, the attacker may generate many requests per second. If the target is incoming bandwidth, the attacker may generate large requests to quickly consume the bandwidth. In both of these cases, the

content of the requests is not important, just their rate and size. Finally, if the target is outgoing bandwidth, the attacker may generate requests that maximize the size of replies, using the ANY query type.

Some attacks are *polymorphic* – they change their features during the attack event. Any attack features may change: how spoofing is done, which sources generate attacks, and the content of attack requests.

A *naive* attacker does not have knowledge about *DDiDD* and is focused only on overwhelming the target server. A *sophisticated* attacker may obtain information about types and parameters of the filters that our defense uses, and they may try to adjust their attack to bypass the defense, or to trick the defense into filtering a legitimate recursive's traffic.

DDiDD works well both against naive and against sophisticated attackers, and against spoofing and non-spoofing attackers, due to its layered defense approach, and multiple filters, as we show in our evaluation.

B. DDiDD *Operation*

To avoid any operational impact on a DNS root server, *DDiDD* consumes packet captures, operating offline to get required parameters, independently of the actual DNS server software. *DDiDD*'s analysis detects an attack, selects a filter or a combination of filters, then deploys filters via `iptables` and `ipset` rules on the server. We consider six filters, described in §IV-C, and implement four that perform well with DNS root traffic: frequent query filter, unknown recursive, wild recursive and hop-count filter. `iptables` work well when number of rules is small (up to 2% delay increase for 5 rules) and matching is needed on query content. We use `iptables` to implement the frequent query filter, for 1–5 frequent query names. `ipset` uses an indexed data structure and provides efficient matching of thousands or even millions of rules, without added delay. We use it when blocking attack sources, identified by unknown recursive, wild recursive and hop-count filters. `iptables`/`ipset` or their equivalents are available on all modern operating systems, thus *DDiDD* is highly deployable by any interested DNS root server. If a root is anycast over multiple points-of-presence (PoPs), *DDiDD* should be deployed at each PoP independently. No synchronization or information exchange is required across instances deployed at different PoPs.

DDiDD automatically selects filters to meet two goals. First, we prefer filters that will remove most attack traffic with low or zero collateral damage to legitimate queries. Second, we aim to select filters quickly, because most DDoS attacks are short [24]. We then revise our selection if attack changes, or if we learn that another filter combination works better. This decision process is fully automated. Further, *DDiDD* is flexible and modular, allowing addition of new filters in the future.

Attack detection. *DDiDD* detects possible attacks by monitoring the status of critical resources and recognizing when a resource is overloaded. We use *collectd* to periodically collect status information from several resources (CPU, memory, inbound and outbound network capacity). We identify possible attacks when any resource exceeds a fraction of its maximum capacity, which we denote as *critical load*.

parameter	meaning	rec. values
L_{FQ}	num. queries for learning	10 K
f_{FQ}	freq. change threshold	0.3
L_{UR}, L_{HC}, L_{WR}	learn. period	2 h (20 m for WR)
U_{UR}, U_{HC}, U_{WR}	use period	2 h
w_i, ..., w_N.	observ. windows	2^0, 2^1, ..., 2^8
t_{WR}	deviance threshold	0.5

TABLE I
FILTER PARAMETERS

We detect attack termination by monitoring the amount of traffic blocked by the deployed filters. We declare the attack over when the traffic blocked by *DDiDD* decreases significantly, and the load on the server stays low as well, for an extended period of time. More details are given in [41].

Filter priming and selection. All filters (e.g., frequent query filter, unknown recursive, wild recursive filter, hop-count filter) require information that must be learned continuously, in absence of attacks. *DDiDD* continuously learns these parameters from packet collection and uses them when the corresponding filter is deployed. Some filters (e.g., frequent query name) also require a short learning phase during an attack. *DDiDD* triggers a short learning phase for these filters when the attack is detected, and repeats it regularly to update filter parameters. After the detection, *DDiDD* uses the incoming traffic to select the filter parameters (for example, finding the frequent query name to filter). For some filters like unknown resolver filter, *DDiDD* uses known legitimate traffic (we provide more details when we describe the filters).

During attack, each filter and some filter combinations are continuously evaluated for potential deployment. We emulate the effect of each filter or their combination on a sample of captured packets. We estimate the success of each filter based on acceptable query load at the server, calculated as the server's average query load times a small multiplicative factor f_{ACC}. Because root servers operate well below their capacity, this approach guarantees that query rates below the acceptable load will also not exhaust the server's CPU or bandwidth resources, and will not trigger attack detection.

We also estimate collateral damage when the filter is parameterized using peace-time (non-attack) traffic. The collateral damage depends on the legitimate traffic's blend and we have verified that it does not change sharply over time. Thus, we can calculate it once and use this estimate for a long time (e.g, months). Based on the estimated effectiveness of the given filter or their combination, and their projected collateral damage, new filters may be selected for deployment and existing filters may be retired.

C. DDiDD *Filters*

In *DDiDD* we have implemented the following filters: (FQ) frequent query name filter, (UR) unknown recursive filter, (HC) hop-count filter and (WR) wild recursive filter. In addition to these, we have also considered (RC) response-code filter and (AR) aggressive recursive filter. Since these two filters do not perform well on root server traffic, we do not include them in *DDiDD*, but we evaluate them on our dataset and summarize results in this section. We show our

recommended filter parameters in Table I. For each filter, we measure the performance and operational cost.

Frequent query name filter (FQ). In our datasets many attacks have queries that follow a given pattern, e.g., have a common suffix. Thus, in practice it is useful to develop filters that remove frequent queries during attack periods.

Approach: We use a simple algorithm to identify frequent query names. We continuously observe L_{FQ} queries of incoming traffic and learn frequency of top-level domains, subdomains and full queries. Under attack, we repeat the calculation and look for segments (TLDs, subdomains or full queries) whose frequency has increased more than a threshold f_{FQ}. These segments are candidates for frequent query names. Segment frequency prior to the attack serves to estimate collateral damage. We evaluated a range of values for L_{FQ} and f_{FQ}. Shorter L_{FQ} than 10,000 reduced mitigation delay, but increased chances of mis-identification of frequent queries. Similarly, lower f_{FQ} than 0.3 lead to some collateral damage. These values should be calibrated for each server.

Operational cost: We can filter frequent query names directly using `iptables`, or we can identify sources that send frequent queries and block them using `ipset`. We denote these two implementation approaches as FQ$_t$ and FQ$_s$. The FQ$_t$ (`iptables`) implementation imposes added processing delay, which greatly increases once we go past five filtering rules, but it minimizes collateral damage. The FQ$_s$ (`ipset`) implementation adds no measurable delay, but it may create collateral damage if spoofing is present, and thus must be deployed together with anti-spoofing filters (UR and HC).

Unknown recursive filter (UR). An allow-list with IP addresses of recursives present prior to the attack can be an effective measure against random-spoofing attacks or those that rent bots. This filter passes traffic from recursives on allow-list to the server, and drops all other traffic.

Approach: An allow-list is built by processing incoming traffic to the DNS root server over period L_{UR} prior to an attack event. The list is then ready to be used for some time U_{UR}, and after that it can be replaced by new list.

DDiDD builds allow-lists proactively at all times, observing traffic over period L_{UR}. We experimented with L_{UR} ranging from 10 minutes (capture 93% of traffic sources) to 6 hours (capture 99% of traffic sources). We also tested values of U_{UR} of up to 1 day, and the allow-lists were very stable.

Operational cost: An allow-list can be implemented efficiently using `ipset`, which adds no processing delay.

Hop count filter (HC). A hop-count filter builds the *TTL-table*, containing source IP addresses, along with one or more TTL values seen in the incoming traffic from each given source. This kind of filter can be effective for attacks that spoof IP addresses of existing recursives. The filter drops traffic from sources that exist in the TTL-table, but whose TTL value does not match the values in the table. All other traffic is forwarded.

Approach: We build the TTL-table by processing incoming traffic to the DNS root server over period L_{HC}. The list is then ready to be used for some time U_{HC}, and after that it can be replaced by new list.

One could use hop counts [51], [33] or TTL values for filtering. TTL values are better choice, since they have larger value space, which improves filter effectiveness. *DDiDD* builds its TTL-list by using each packet in the incoming traffic to the server during the learning period. Such traffic could be spoofed. Prior approaches [51], [33], [7] rely on established TCP connections or they probe sources to reliably learn TTL-table values. These approaches do not work for DNS root servers, which serve mostly UDP traffic and whose policy forbids generation of unsolicited traffic. Hop-count filter parameter values have similar properties to known-recursive parameter values.

Operational cost: We implement this filter efficiently by adding a new `ipset` module to match on an IP address and TTL value (or range).

Wild recursive filter (WR). While query rate of different DNS recursives towards a DNS root server varies widely, individual recursives' behaviors are mostly consistent over short time periods (e.g., several hours). We leverage this observation to build models of each individual recursive's behavior. The model for a given recursive, along with the recursive's IP address is stored in the *rate-table*. During an attack, we identify those recursives that send more aggressively than their rate-table predicts as *wild recursives*. Wild recursive filter drops traffic from wild recursives, and it forwards all other traffic.

Approach: A wild-recursive filter learns the rate of a DNS recursive's interaction with the DNS root server over multiple time windows, $w_1, w_2, w_3, ..., w_N$, during learning period L_{WR}. For each window, the filter learns the mean and standard deviation of the number of queries observed and stores them in the rate-table. The rate-table can be used for some time U_{WR}, and after that it can be replaced by a new table.

When the attack is detected, the filter measures the current query rates over the same windows. It then calculates the difference between the current rate r_{cw_i} in the window w_i and the rate expected by the model: $mean_{w_i} + 3 \times std_{w_i}$. We then calculate a smoothed, normalized deviance score d_t at time t as: $d_t = (d_{t-1} \times 0.5) + 0.5 \times \sum_i \frac{r_{cw_i} - mean_{w_i} - 3 * std_{w_i}}{std_{w_i}}$. Those recursives whose deviance score exceeds threshold t_{WR} will be identified as wild recursives.

We experimented with values for L_{WR} between 10 minutes and 6 hours. While performance was relatively stable, lower values led to lower collateral damage, since they captured recent traffic trends. We experimented with uniformly distributed and exponentially distributed (powers of two) window sizes. Exponentially distributed windows led to lower mitigation delay, because they capture both aggressive and stealthy attackers. We also experimented with 1–9 windows. Higher number of windows had slightly higher collateral damage, but they significantly improved filter effectiveness, because they enabled us to identify sporadic attackers. Learned models become quickly outdated so we set $U_{WR} = L_{WR}$. We experimented with values for the threshold t_{WR} from 0.1 to 16. Values higher than 0.5 minimized collateral damage.

Operational cost: This filter is implemented by processing

```
filters: array of all possible filters
candidates: array of filters that can be deployed
deployed: array of currently deployed filters
AL: acceptable load
CL: current load

function select_filters()
1:  select_candidates()
2:  deployed=deploy_single()
3:  if not deployed:
4:      deploy_combo()

function select_candidates()
1:  for F in filters:
2:      if F can reduce load to AL:
3:          candidates.append(F)

function deploy_single()
1: current_fp = 1, best = null
2: for C in candidates:
3:     if C.fp < current_fp:
4:         best = C
5:         current_fp = C.fp
6: if best is not null:
7:     deployed.clear()
8:     deployed.append(best)
9:     return true
10: return false

function deploy_combo()
1: tofilter = CL - AL; deployed.clear()
2: for T in ur, hcf, fq, wild:
3:     for C in candidates:
4:         if C.type not T:
5:             continue
6:         if C is effective:
7:             deployed.append(C)
8:             tofilter -= C.filtered
9:             if tofilter <= 0
10:                return
```

Fig. 1. Pseudocode for filter selection

the traffic incoming to the DNS server offline. When the attack starts, the filter identifies wild recursives and inserts corresponding `ipset` rules to block their traffic.

Response code filter (RC). For some DNS servers, queries with missing names are rare. For example, at Akamai only a small fraction of legitimate queries result in NXDomain [43] replies, while attackers often query for random query names. We therefore considered a filter based on response codes that discards NXDomain responses. Unfortunately, more than 60% of *root* DNS traffic involves non-existing TLDs. Thus for root DNS traffic, a response code filter will have large collateral damage, and we do not currently include it in *DDiDD*.

Aggressive recursive filter (AR). This filter blocks the aggressive clients during an attack, starting with the client that sends the highest query rate and moving down. Filter adds addresses to the block-list until the query load reduces to acceptable levels. We evaluated this filter on our dataset. It performs well when attacks use non-spoofed traffic, but its performance is consistently worse than that of wild recursive filter. We thus do not include it in *DDiDD*.

D. Filter Selection and Synchronization

In this section we discuss how filters are selected for deployment and why their learning periods have to be synchronized. **Filter selection.** Our goal was to design effective filter selection process, which minimizes collateral damage to legitimate traffic. Our pseudocode for filter selection is given in Figure 1. At each time interval (e.g., one second), if the current query load (CL) on the server (queries per second) is higher than the acceptable load (AL), we first select candidate filters. We continuously emulate operation of all filters, thus we produce for each filter an estimate of the amount of queries they would drop. Our candidate filters are those whose drop estimates are positive. If among the candidate filters there are any that could reduce the load to AL, we will select the filter with the lowest estimated collateral damage (described in §IV-B) and deploy only this filter (function *deploy_single*).

If no such filters exist, we will consider combinations of multiple filters (function *deploy_combo*). Not all combinations are valid, which greatly reduces complexity of this step. HC filter must be deployed after an UR filter, since HC is pass-through for addresses that do not exist in TTL-table. UR filter removes queries that spoof unknown recursives, thus guaranteeing that addresses of queries that pass will be present in TTL-table. FQ_t could be deployed together with any other filter. FQ_s and WR filters must be deployed after UR and HC, because they make per-source blocking decisions, and require reliable source identities. Since both FQ_t and FQ_s filter frequent query names, only one of them should be deployed. FQ_t has zero collateral damage and is considered first. If it cannot be supported operationally (there are more than five query names, and thus there will be added processing delay), FQ_s will be considered. In addition to considering filters in a specific order for deployment, we only consider filters that are *effective* – filter at least 5% of excess traffic (function *effective*). Deployment is finalized as soon as the filter combination can reduce the load below AL.

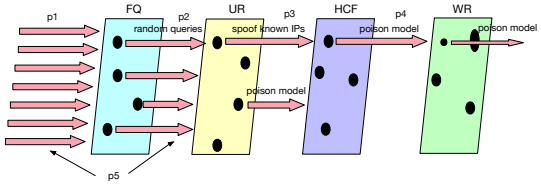

Fig. 2. Swiss cheese model of defense

Filter synchronization. *DDiDD* may engage one or multiple filters to mitigate an attack. When some filter combinations are engaged, it is important that their learning periods match, so that each filter has entries for the same recursives in their table. Because we need a shorter learning period for wild recursive filter, than for the unknown recursive and hop-count filter, we learn parameters over 2 hours, and then keep updating WR entries each 20 minutes to keep them as recent as possible.

Sophisticated adversary. Each of the filters we consider could be bypassed by a sophisticated adversary. We now discuss how their combination makes this challenging (Figure 2).

FQ filter could be bypassed by the attacker sending random queries. UR filter could be bypassed by the attacker spoofing existing (known) recursives. UR, HC and WR filters could each be bypassed by poisoning the models during learning. One way to counter poisoning attacks could be to learn over longer time periods, from random traffic samples. While this works for UR and HC, whose data is fairly stable, it would greatly diminish effectiveness of WR filter, and it would complicate filter synchronization. Our approach is to handle poisoning attacks only at WR filter, and to rely on the Swiss cheese defense model (Figure 2) to capture attackers that bypass one filter layer, but can be stopped at the other. Thus random queries may bypass FQ, but will be stopped at UR if they are from new sources, or at HCF if they are spoofed. At WR, queries sent by recursives at high rate (spoofed or not) can be detected and dropped. This leaves poisoning attacks at

PoP	date	start (UTC)	dur (sec)	ULQ	DNS mon	FQ		UR		HC		WR		*DDiDD* $_F$		*DDiDD* $_P$	
						con	cd	con	cd	con	cd	con	cd	con	cd	con	cd
LAX	2015-11-30	06:50	8,918*	98	100	**100**	**0**	**99.1**	**1.8**	0.3	1.4	0	5.5	**99.1**	**0.4**	99.3	1.7
LAX	2015-12-01	05:10	3,781*	100	100	**98.7**	**0**	**99.1**	**0**	0.6	0	0	0	**99.3**	**0**	**99.4**	**0**
LAX	2016-06-25	22:18	2,436*	52	99	0	0	**100**	**0.1**	0	0	0	0	**100**	**0.1**	**100**	**0.1**
LAX	2017-02-21	06:40	6,992*	2	1	**98.4**	**0**	0.1	1.8	0.1	1.5	98.4	0	**99**	**0**	**98.8**	**0**
LAX	2017-03-06	04:43	19,835*	6	5	**98.8**	**0**	0	1.1	0	0.4	91.6	1.5	**100**	**0**	92.3	1.5
LAX	2017-04-25	09:54	10,414*	3	4	**98.3**	**0**	0	1.1	0	0.7	94.9	2	**99.1**	**0**	95.1	2
ARI	2019-09-07	06:45	80	0	5	0	0	**93.3**	**0.6**	0	0.8	0	0.1	**93.7**	**0.6**	**93.1**	**0.6**
LAX	2019-09-07	06:45	80	23	5	0	0	**100**	**0.9**	0	0.2	0	0.2	**100**	**0.9**	**100**	**0.9**
MIA	2019-09-07	06:45	80	8	5	0	0	**100**	**0.6**	0	0	0	0.4	**100**	**0.6**	**100**	**0.6**
SIN	2020-02-13	08:05	206	14	2	**100**	**0**	0	0.3	4.8	0	38.5	0.5	**100**	**0**	97.5	0.8
ARI	2020-10-24	02:55	445	67	7	0	0	**100**	**1.3**	0	0	0	0.8	**100**	**1.3**	**100**	**1.3**
ARI	2021-05-28	02:35	70	25	3	0	0	**100**	**1.1**	0	0	0	0.1	**100**	**1.1**	**100**	**1.1**
IAD	2021-05-28	02:35	70	63	3	0	0	**100**	**0.4**	0	0	2.7	0	**100**	**0.5**	**100**	**0.5**
LAX	2021-05-28	02:35	70	3	3	0	0	**100**	**0.4**	0	0	0	0	**100**	**0.4**	**100**	**0.4**
MIA	2021-05-28	02:35	70	2	3	0	0	**100**	**1.5**	0	0	0	0	**100**	**1.7**	**100**	**1.7**
SIN	2021-05-28	02:35	61	41	3	0	0	**100**	**0**	0	0	0	0	**100**	**0**	**100**	**0**

TABLE II

DDiDD PERFORMANCE: COMPARING LOAD CONTROL (CON) AND COLLATERAL DAMAGE (CD) FOR EACH POSSIBLE FILTER AND *DDiDD* AS A WHOLE. WE HIGHLIGHT RESULTS WITHIN 1% OF THE BEST PERFORMANCE IN BOLD. FOR LONG ATTACKS (*) WE SIMULATE ONLY THE FIRST 600 SECONDS.

WR filter (thin red arrow at the top right of Figure 2), where each bot poisons the rate model for itself by sending sporadic traffic during learning, with high fluctuations. This can lead the filter to model a large expected rate for the bot in each window, due to large standard deviation. To address this attack, we learn only when load on the server is low ($avg + stdev$). This forces the attacker to engage their bots very sporadically, which becomes an outlier and is excluded from the model.

V. EVALUATION

We use datasets containing real DNS root traffic and attacks (§V-A) to calculate success metrics (§V-B) that characterize *DDiDD* performance (§V-C).

A. Datasets

We use datasets collected at *B-root*, one of 13 root identifiers. These datasets are publicly available [3] in both pcap and text format. The operators of *B-root* identify attacks based on unusual traffic rates and system load, as seen from operational monitoring. Our evaluation uses ten diverse attack events spanning six years (see Table II). During events in 2017 and later *B-root* employed anycast network with multiple points-of-presence (PoPs). Some attacks affected only one PoP (e.g., 2020-02-13), while others targeted all PoPs (e.g., 2020-05-28).

We confirm that our selected events are DDoS attacks based on DNSmon observations shown in the "DNSmon" column Table II. DNSmon reports the fraction of responses received by many (about 100) physically distributed probers, which query each DNS root every 10 minutes. In Table II, the first three attack events had a large impact, showing 99–100% of unanswered queries, as publicly reported [31], [1], [2]. The other seven events had smaller impacts (1–7% unanswered queries), because they were shorter (5 minutes and less) and sent at a lower rate, and because *B-root*'s capacity had increased. DNSmon reports reflect aggregate performance across all PoPs, so the percentage of unanswered queries at each PoP might be higher than measured by DNSmon. We include traces from all the PoPs in our analysis, and simulate running of *DDiDD* at each PoP. We use ground truth for attack start and stop times to start and stop *DDiDD*'s simulation, and use $f_{ACC} = 2.5$. During attacks, query rate at the server increases more than 10-fold, so using $f_{ACC} = 2.5$ is reasonable.

While attackers could generate any random traffic to port 53, attacks in our dataset had unique content or traffic signatures, which enabled us to establish *ground truth* during evaluation. Attacks on 2015-11-30, 2015-12-01, 2017-02-21, 2017-03-06, 2017-04-25, and 2020-02-13 had used either several specific queries or a random prefix with a common, specific, suffix. Attack on 2016-06-25 was a TCP SYN flood. Attacks on 2019-09-07 and 2020-10-24 and 2021-05-28 sent malformed UDP traffic to port 53, which consumed resources at the server, but did not parse into legitimate query format.

Ethical considerations. Our analysis is performed on packet traces incoming to and outgoing from *B-root*. Both source and destination IP addresses are anonymized using Crypto-PAn [52], [57]. Packet payloads are not anonymized, which allows us to establish ground truth in evaluation. After ground truth is established, analysis is automated and we report only aggregate results. These steps preserve resolver privacy.

B. Metrics

Our goal is to reduce load on the DNS root server, by filtering malicious traffic, to allow serving more legitimate users when under duress. We therefore consider two success metrics: (1) *controlled load*, the percent of time when server load is at or below acceptable load due to defense's actions, ideally 100%; (2) *collateral damage*, the percent of legitimate queries filtered, with an ideal of 0%.

C. DDiDD *Performance*

Table II shows *DDiDD*'s performance per each PoP affected by a given attack. We show several defense configurations: first, each filter by itself (FQ, UR, HC, or WR), then the full *DDiDD* with all four filters and a partial *DDiDD* with only UR, HC, and WR filters. Removing the FQ filter from the partial *DDiDD* simulates a smart adversary, which randomizes queries for each attack.

These experiments confirm that *no single defense does well in all attack cases*. The FQ filter does very well in attacks

that use similar queries, but has no effect otherwise. The UR filter performs well in many attacks. HC does not work well by itself, but enhances other filters. Finally, WR does well in a few attacks, where some recursives, which are present prior to the attack, modify their behavior to become more aggressive. This evaluation demonstrates that we need multiple filters to handle all attack events.

We further show that *the full* DDiDD *automatically chooses the best filter or combination of filters for each attack*, always achieving 93% or higher controlled load and at most 1.7% collateral damage. *DDiDD* selects the optimal filter combination in 1–3 seconds.

Partial *DDiDD*'s performance (the right-most column) shows how well it would handle *an adversary that randomizes queries*. *DDiDD* controls load for most of the time (92.3%–100%), with low collateral damage (2% or lower), with all filters selected in 3 s or less.

We compare collateral damage of *DDiDD* with percentage of legitimate queries at the affected PoP that fail to receive a response during the original attack, without *DDiDD*. We calculate this percentage from our datasets and show it in the fifth column (ULQ) of Table II. This is an internal measure of DoS impact and it can differ from the external measurements by DNSmon, because of several reasons. First, DNSmon averages measurements over 10 minutes and across all PoPs for a given root, while our internal-DoS measure is per PoP and it is averaged over the duration of the attack. For these reasons DNSmon will often underestimate attack impact, as is the case for many of our attacks. Second, if *B-root*'s incoming bandwidth were overloaded, DNSmon could measure higher loss rate than our internal-DoS measure. This is the case, for example, for 2019-09-07 attack.

Full *DDiDD*'s and partial *DDiDD*'s collateral damage is always lower than DNSmon (external) and ULQ (internal) measures. Thus *DDiDD* improves legitimate traffic's handling during DoS attacks. *DDiDD* is also effective, reducing resource consumption by controlling server load, 93–100% of time, after a short initial delay of 1–3 seconds.

Legitimate flash crowds. While three attacks in 2017 overloaded *B-root*, they involved a large number of recursives involved (around 50 K per event), large difference in rates per recursive, and did not spoof. Legitimate flash crowds would show similar patterns. In 2017 events, *DDiDD* dropped only traffic that was causing the overload event, and only as much as to free server resources from overload.

Polymorphic attacks. In evaluation events *DDiDD* changes defenses because the attacks change. During 2015-11-30 attack there were periods where existing clients were spoofed with incremental TTL values, traversing the entire TTL value space. Partial *DDiDD* correctly switched from UR to UR+HC combo to handle these cases. During 2020-02-13 attack, single UR, HC and WR filters could not sufficiently reduce the load. Partial *DDiDD* deployed all three filters, which managed to reduce the load.

We demonstrate how *DDiDD* can nimbly adjust filter selection by using an artificial polymorphic attack in Figure 3.

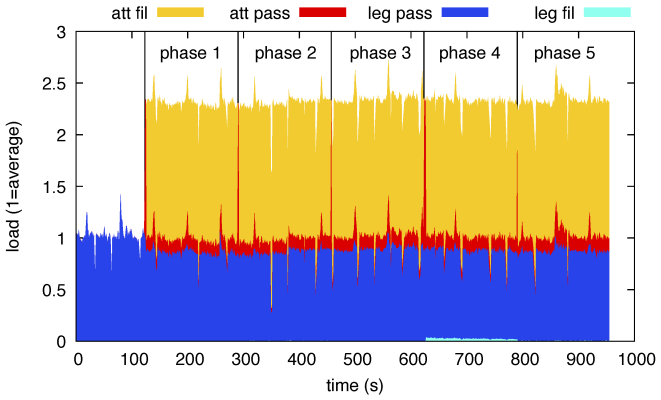

Fig. 3. *DDiDD* evaluation for a synthetic polymorphic attack.

We create a synthetic attack by mixing legitimate traffic from February 2017 with five synthetic attacks, which correspond to p1–p5 labels in Figure 2: (p1) a random-spoofed attack with a fixed query name, (p2) an attack with random query names, (p3) same as (p2) but also spoofs only known recursives using random TTL values, (p4) same as (p3) but spoofs with correct TTL values, (p5) same as (p1) but 90% of queries are random and 10% use a fixed query name. We find that *DDiDD* quickly converges to the best single filter for each attack strategy: FQ_t, UR, HC, WR and FQ_s, respectively. Figure 3 shows passed and filtered legitimate and attack traffic for our synthetic attack—overall controlled load was 99.1%, collateral damage was 0.7%, and selection delay was 1–4 s.

VI. Conclusion

This paper provides the first in-depth design and evaluation of an automated, layered approach to mitigate DDoS on DNS root. Evaluated on ten real-world DDoS attacks on *B-root*, *DDiDD* quickly selects the best filter or filter combination from a library of filters, and deploys it automatically. *DDiDD* reduces server load to acceptable levels within seconds, with collateral damage under 2%. *DDiDD* is adaptive to the polymorphic attack events, which change attack pattern during an ongoing attack event, and nimbly makes new filter selection in up to 4 seconds. It further has low operational cost, working offline to process incoming traffic at the server, and producing filtering rules, which can be implemented at no added processing delays using `ipset`. We release *DDiDD* as open source.

Acknowledgments

This work is partially supported by the National Science Foundation (grant NSF OAC-1739034) and DHS HSARPA Cyber Security Division (grant SHQDC-17-R-B0004-TTA.02-0006-I), in collaboration with NWO.

References

[1] Root server operations, Events of 2015-11-30. http://root-servers.org/media/news/events-of-20151130.txt.

[2] Root server operations, Events of 2016-06-25. http://root-servers.org/media/news/events-of-20160625.txt.

[3] Analysis of network traffic (ANT) group, ANT datasets. https://ant.isi.edu/datasets/all.html, 2022. Datasets with Anomaly keywords, [Online; accessed 19-Feb-2022].

[4] L. Adriano. Canadian comms company suffers DDoS attack. https://www.insurancebusinessmag.com/ca/news/breaking-news/canadian-comms-company-suffers-ddos-attack-310819.aspx, 2021.

[5] Akamai InfoSec. A look back at the DDoS trends of 2018. https://blogs.akamai.com/2019/01/a-look-back-at-the-ddos-trends-of-2018.html, 2017.

[6] M. Allman. On eliminating root nameservers from the DNS. In *Proc. of ACM Workshop on Hot Topics in Networks*, Princeton, NJ, USA, Nov. 2019. ACM.

[7] M. Backes, T. Holz, C. Rossow, T. Rytilahti, M. Simeonovski, and B. Stock. On the feasibility of TTL-based filtering for DRDoS mitigation. In *RAID Symposium*, volume 9854, pages 303–322, 2016.

[8] C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu. Model-based adaptive DoS attack mitigation. In *7th IEEE SEAMS*, pages 119–128, Piscataway, NJ, USA, 2012. IEEE Press.

[9] Bill Slater, President of Chicago ISOC. The Internet Outage and Attacks of October 2002. https://tinyurl.com/22hrfj4f, 2002.

[10] J. Blazina. Stonefish—automating DDoS mitigation at the edge. https://tinyurl.com/4yh5t57z, 2019. [Online; accessed 30-May-2019].

[11] M. Calder, A. Flavel, E. Katz-Bassett, R. Mahajan, and J. Padhye. Analyzing the performance of an anycast CDN. In *Proc. ACM IMC*, pages 531–537, Tokyo, Japan, Oct. 2015. ACM.

[12] E. Y. Chen and M. Itoh. A whitelist approach to protect SIP servers from flooding attacks. In *IEEE Wkshp. Comm. Qual. and Reliab.*, 2010.

[13] F. David Warbuton. Ddos attack trends for 2020. https://www.f5.com/labs/articles/threat-intelligence/ddos-attack-trends-for-2020, 2020.

[14] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, Sept. 2002.

[15] Z. Duan, X. Yuan, and J. Chandrashekar. Controlling IP spoofing through interdomain packet filters. *IEEE Trans. on Dependable and Secure Computing*, 5(1), 2008.

[16] T. Emmons. 2021: Volumetric DDoS attacks rising fast. https://www.akamai.com/blog/security/2021-volumetric-ddos-attacks-rising-fast/.

[17] A. Fabre. L4Drop: XDP DDoS mitigations. https://tinyurl.com/ycx6m7r7, 2018.

[18] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. RFC 2267, internet-rfc, May 2000. also BCP-38.

[19] D. Gillman, Y. Lin, B. Maggs, and R. K. Sitaraman. Protecting websites from attack with secure delivery networks. *Comp.*, 48(4):26–34, 2015.

[20] W. Hardaker. Analyzing and mitigating privacy with the DNS root service, 02 2018.

[21] ICANN. Root server attack on 6 February 2007. https://www.icann.org/en/system/files/files/factsheet-dns-attack-08mar07-en.pdf, 2007.

[22] I. S. C. (ISC). Using the response rate limiting feature. https://kb.isc.org/docs/aa-00994, 2018.

[23] C. Jin, H. Wang, and K. G. Shin. Hop-count filtering: an effective defense against spoofed DDoS traffic. In *Proceedings of the 10th ACM conf. on Computer and comm. security*, pages 30–41. ACM, 2003.

[24] M. Jonker, A. King, J. Krupp, C. Rossow, A. Sperotto, and A. Dainotti. Millions of targets under attack: a macroscopic characterization of the DoS ecosystem. In *Proc. of ACM IMC*, pages 100–113, 2017.

[25] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *Proc. USENIX NSDI*, pages 287–300. USENIX Association, 2005.

[26] P. Koch, M. Larson, and P. Hoffman. Initializing a DNS resolver with priming queries. RFC 8109, Internet Request For Comments, Mar. 2017. (also Internet BCP-209).

[27] T. Koch, K. Li, C. Ardi, E. Katz-Bassett, M. Calder, and J. Heidemann. Anycast in context: A tale of two systems. In *Proc. of ACM SIGCOMM*, Virtual, Aug. 2021. ACM.

[28] W. Kumari and P. Hoffman. Running a root server local to a resolver. RFC 8806, Internet Request For Comments, June 2020.

[29] J. Levine. DNS blacklists and whitelists. RFC 5782, IETF, Feb. 2010.

[30] G. C. Moura, J. Heidemann, R. d. O. Schmidt, and W. Hardaker. Cache me if you can: Effects of DNS time-to-live. In *Proc. of ACM IMC*, pages 101–115, 2019.

[31] G. C. Moura, R. d. O. Schmidt, J. Heidemann, W. B. de Vries, M. Muller, L. Wei, and C. Hesselman. Anycast vs. DDoS: Evaluating the November 2015 root DNS event. In *Proc. ACM IMC*, 2016.

[32] G. C. M. Moura, J. Heidemann, M. Müller, R. de O. Schmidt, and M. Davids. When the dike breaks: Dissecting DNS defenses during DDoS. In *Proc. of ACM IMC*, Oct. 2018.

[33] A. Mukaddam, I. Elhajj, A. Kayssi, and A. Chehab. IP spoofing detection using modified hop count. In *2014 IEEE 28th Int. Conf. on Advanced Information Networking and Apps*, pages 512–516, 2014.

[34] A. Network. NETSCOUT Arbor's 13th annual worldwide infrastructure security report. https://tinyurl.com/2p89k9ru, 2019.

[35] L. H. Newman. Github servived the biggest DDoS attack ever recorded. https://www.wired.com/story/github-ddos-memcached/, 2018. [Online; accessed 19-March-2018].

[36] G. Oikonomou and J. Mirkovic. Modeling human behavior for defense against flash-crowd attacks. In *IEEE ICC*, pages 1–6. IEEE, 2009.

[37] J. Pang, J. Hendricks, A. Akella, R. D. Prisco, B. Maggs, and S. Seshan. Availability, usage, and deployment characteristics of the Domain Name System. In *Proc. of ACM IMC*. ACM, Oct. 2004.

[38] T. Peng, C. Leckie, and K. Ramamohanarao. Proactively detecting distributed denial of service attacks using source IP address monitoring. In *Int. conf. on research in networking*. Springer, 2004.

[39] S. Ramanathan, J. Mirkovic, and M. Yu. Blag: Improving the accuracy of blacklists. In *NDSS*, 2020.

[40] A. Rizvi, L. Bertholdo, J. Ceron, and J. Heidemann. Anycast agility: Network playbooks to fight DDoS. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4201–4218, Boston, MA, Aug. 2022. USENIX Association.

[41] A. Rizvi, J. Heidemann, and J. Mirkovic. Dynamically selecting defenses to DDoS for DNS (extended). Technical Report ISI-TR-736, USC/Information Sciences Institute, May 2019.

[42] B. Schneier. Lessons from the Dyn DDoS attack. https://tinyurl.com/yc4f5ez4, 2016.

[43] K. Schomp, O. Bhardwaj, E. Kurdoglu, M. Muhaimen, and R. K. Sitaraman. Akamai DNS: providing authoritative answers to the world's queries. In *Proc. ACM SIGCOMM*, pages 465–478, 2020.

[44] M. Sung and J. Xu. IP traceback-based intelligent packet filtering: a novel technique for defending against internet DDoS attacks. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):861–872, 2003.

[45] R. Tandon, J. Mirkovic, and P. Charnsethikul. Quantifying cloud misbehavior. In *2020 IEEE 9th CloudNet*, pages 1–8. IEEE, 2020.

[46] R. Tandon, A. Palia, J. Ramani, B. Paulsen, G. Bartlett, and J. Mirkovic. Defending web servers against flash crowd attacks. In *Int. Conf. on ACNS*, pages 338–361. Springer, 2021.

[47] The Guardian. DDoS attack that disrupted internet was largest of its kind in history, experts say. https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet, 2016.

[48] R. Thomas, B. Mark, T. Johnson, and J. Croall. Netbouncer: client-legitimacy-based high-performance DDoS filtering. In *Proc. DARPA Information Survivability Conf.*, volume 1, pages 14–25. IEEE, 2003.

[49] A. Toh. Azure DDoS protection—2021 Q1 and Q2 DDoS attack trends. https://azure.microsoft.com/en-us/blog/azure-ddos-protection-2021-q1-and-q2-ddos-attack-trends/, 2021.

[50] P. Vixie, G. Sneeringer, and M. Schleifer. Events of 21-Oct-2002. web page http://c.root-servers.org/october21.txt, Nov. 2002.

[51] H. Wang, C. Jin, and K. G. Shin. Defense against spoofed IP traffic using hop-count filtering. *IEEE/ACM ToN*, 15(1):40–53, 2007.

[52] J. Xu, J. Fan, M. H. Ammar, and S. B. Moon. Prefix-preserving IP address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. In *10th IEEE ICNP, 2002*, pages 280–289. IEEE, 2002.

[53] A. Yaar, A. Perrig, and D. Song. StackPi: New packet marking and filtering mechanisms for DDoS and IP spoofing defense. *IEEE Journal on Selected Areas in Communications*, 24(10):1853–1863, 2006.

[54] O. Yoachimik. Who DDoS'd Austin? https://blog.cloudflare.com/who-ddosd-austin/, 2019.

[55] M. Yoon. Using whitelisting to mitigate DDoS attacks on critical internet sites. *IEEE Communications Magazine*, 48(7):110–115, 2010.

[56] ZD Net. Cloudflare says it stopped the largest DDoS attack ever reported. https://www.zdnet.com/article/cloudflare-says-it-stopped-the-largest-ddos-attack-ever-reported/, 2020. [Online; accessed 7-Oct-2021].

[57] L. Zhu and J. Heidemann. Dnsanon: extract DNS traffic from pcap to text with optionally anonymization. https://ant.isi.edu/software/dnsanon/index.html, 2017. [Online; accessed 20-Jan-2018].