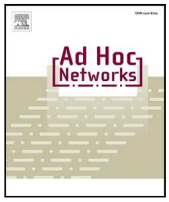
Ad Hoc Networks 151 (2023) 103259

Contents lists available at ScienceDirect

Ad Hoc Networks

journal homepage: www.elsevier.com/locate/adhoc



Defending Root DNS Servers against DDoS Using Layered Defenses (Extended)

ASM Rizvi *, Jelena Mirkovic, John Heidemann, Wesley Hardaker, Robert Story

University of Southern California/Information Sciences Institute, United States of America

ARTICLE INFO

Keywords:
DDoS
DNS
Filtering

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 show our system can successfully mitigate resource exhaustion using replay of a real-world attack. We can handle millions of filtering rules without noticeable operational overhead.

1. Introduction

Distributed-Denial-of-Service (DDoS) attacks remain a serious problem [1–4], 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% [5]. While some large-volume DDoS attacks make front page news (for example, the 1.35 Tb/s [6] attack on Github in Feb. 2018, or 2021 17.2 M requests per second attack, detected by CloudFlare [7]), many more attacks occur daily and disrupt operations of thousands of targets [8,9].

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 [10–14]. 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 [15].

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 [16–19]. Other approaches aim to separate legitimate from attack clients, e.g., via CAPTCHAs [20], or by using models of typical client behavior [21,22]. 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 [15]. Second, DNS traffic to roots is generated by recursive resolvers. Since there is neither direct

* Corresponding author.
E-mail address: asmrizvi@usc.edu (ASM Rizvi).

https://doi.org/10.1016/j.adhoc.2023.103259
Received 13 May 2023; Accepted 19 July 2023
Available online 4 August 2023

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 [23] 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 [24–27]. While prior work quantified performance of some individual filters for general DDoS attacks [25–27], and other work qualitatively described commercial deployments (such as Akamai's [24]), 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 and our DDIDD tool that we use in this paper [28,29].

An earlier version of this work was previously published at a conference [30]. This journal version extends that work with additional background (Section 2), new description of how we detect DDoS by monitoring resources for exhaustion (Section 4.2.1), additional validation of filter parameters (Section 4.3), and evaluation of attacks on resource consumption (Section 5.4).

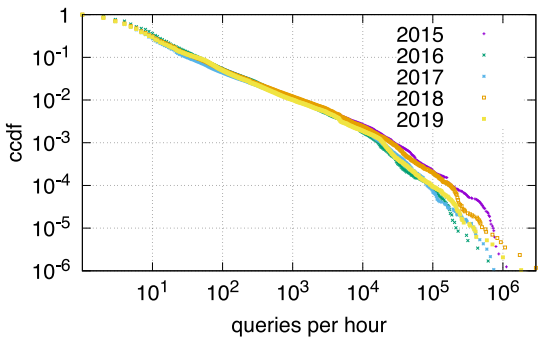


Fig. 1. Complementary cumulative distribution function of the number of requests per hour sent to *B-root* on five random dates between 2015 and 2019.

2. Background: DNS and DDoS

Domain Name System (DNS) is part of the critical Internet infrastructure. It maps between resource names and IP addresses, using a hierarchical database distributed across *authoritative DNS nameservers* (*authoritatives* for short). The DNS root is on top of the hierarchy, followed by top-level domain (TLD) servers and subdomain servers. Each authoritative nameserver is responsible for maintaining mapping of some portion of the DNS namespace, and for replying to queries about that portion to any DNS client.

Users usually do not directly query the DNS, but instead use *recursive resolvers* ("recursives" for short) that resolve names on their behalf. There are many recursives, and each serves some local users by proxying for them the translation of DNS names to IP addresses, and caching any new data learned from the authoritative servers. Users' computers are clients of recursives, and recursives are clients of the authoritatives.

Each DNS name consists of multiple components, separated by periods, such as `www.example.com`. The rightmost segment denotes a top-level domain or TLD, such as *.com* or *.us*. When a recursive resolver looks up a name, it parses each component, querying authoritative nameservers, if it does not have a resolution for that given suffix in its cache. Components and their resolutions are cached for durations specified by their owner, and can be overridden by the recursive's configuration.

2.1. DNS root traffic

Because recursives cache responses from DNS root, and there are only a few thousand TLDs that can be cached for 24 h or more, one expects that recursives query the authoritatives for the root infrequently. The actual traffic from resolvers, however, defies this expectation by a large margin. Fig. 1 illustrates the complementary cumulative distribution function (ccdf) of the number of queries to *B-root* per hour at a random hour in each of the years 2015, 2016, 2017, 2018 and 2019. There is a wide range of rates across 5 degrees of magnitude. While majority of resolvers exhibit the behavior we expect – 95% send fewer than 62 queries per hour and 99% send fewer than 1,500 queries per hour – a small number of resolvers sends excessive numbers of queries, up to 100,000 queries per hour!

Why are roots receiving so much more traffic than one would expect? There are several possible explanations. First, roots receive many queries (36%–39% in our dataset) that do not have a valid TLD [31–33]. Chrome browsers make such random DNS queries to detect DNS hijacking [34]. Roots will return a no-such-domain (NXDomain) reply to these queries, but such replies are not cached by resolvers. Second, some recursives may not cache properly and so reissue queries that could be cached. Finally, some resolvers query the roots directly, perhaps to monitor it. More research is needed to establish root causes of this excessive traffic.

Root servers operate as a service to the Internet and are committed to serving the root DNS zone as defined by IANA to all queriers (for example, see [35]). Due to this policy, root server operators prioritize responding to all queries, with the exception of obvious attacks and operational threats.

2.2. The DNS root and DDoS

Historically there have been several large attacks on DNS root servers. In 2002 [36], a large volumetric attack hit all 13 DNS root servers for an hour, with nine of 13 root servers largely inaccessible. In 2007 [37], a volumetric attack hit six DNS root servers, and lasted 3 h and 5 h. Two servers were noticeably affected. In November/December 2015 [13], most of the root name servers were hit by two volumetric attacks containing millions of spoofed queries per second. While some root servers were lightly hit, others saw severe traffic loss of 95% or more. Analysis showed the attacks inflicted collateral damage to services collocated with root servers [13]. Although caching of root contents at recursives reduces the end-user impact of these attacks [38, 39], DNS outages at CDNs have impacted prominent user-facing services [8]. Effective DDoS defense for the DNS root is thus necessary. We use data from eight attacks in the years following 2015.

While DDoS attacks have been studied for decades, DDoS on DNS root servers pose some unique challenges. Unlike web traffic, most DNS queries use UDP and UDP support is required, so DNS is vulnerable to *IP spoofing*, making filtering by source IP address ineffective. Second, root severs see a huge *diversity of query rates*—legitimate traffic spans five orders of magnitude, complicating traffic modeling and filtering big senders. Third, *root DNS* uses a small number of fixed IP addresses, so shifting traffic to other anycast rings is not feasible. Finally, the DNS root has a very high commitment to serving all queries, so *collateral damage* is a large concern; good queries should be answered.

3. 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.

3.1. Flash-crowd DDoS defenses

CAPTCHAs [40,41] 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 [23] 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 [42–46] 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 [26], 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 [24, 47], Verizon [48], and Cloudflare [49,50], 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.

3.2. Spoofed traffic filtering

Several filters to remove spoofed traffic have been proposed: hop-count filtering [25–27], route traceback [51], route-based filtering [52], path identifier [53], unknown client filtering [43,44], and client legitimacy based on network, transport and application layer information [54]. 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 [25] 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 [27]. 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.

3.3. DDoS on DNS

BIND pioneered Response Rate Limiting (RRL) to avoid excessive replies [55] 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 [24,47]. 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. [24] 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 [13]. Recent work has taken this idea further, suggesting that a network playbook can pre-evaluate routing options to shift traffic across anycast sites [19]. 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 [56–58]. 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.

4. *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* (Section 5) mostly drops queries from sources causing the legitimate flash crowd.

4.1. 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.

4.2. 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 Section 4.3, 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 [59]. 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.

4.2.1. Attack detection

DDoS attacks cause problems because they exhaust some resource at the target. As an example, in 2015 DDoS attacks on the root DNS some operators could reply to all queries, but some failed to receive queries or could not respond to them typically because of bandwidth limitations [13,60]. *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*. Our system detects attacks considering different resources:

Ingress network bandwidth: Volumetric attacks like UDP flooding can saturate the ingress network bandwidth [61]. The memcached attack did not even send DNS queries, but exhausted channel capacity [6]. If a root server has a capacity of I Gb/s, then attack traffic of rate I_A (where $I_A > I$) will result in user loss of approximately $(I_A - I)/I_A$ legitimate queries.

Physical memory: Several types of attacks target server memory, forcing kernels to buffer IP fragments [62,63], or TCP connections from a TCP-SYN flood attack [64]. Today's operating systems are generally hardened to these attacks and drop partially-complete information when resources are limited.

CPU usage: CPU usage increases in proportion to query rates, and while non-DNS traffic may be filtered at a firewall, DNS queries require some application-level processing. Query processing can incur an asymmetric cost; they are cheap for zombies to generate but much more expensive for servers to detect and discard or handle.

Egress network bandwidth: It is typical in DNS deployments that the responses are larger than the corresponding query size [63,65]. This amplification attack can result in exhaustion of the egress network bandwidth before the ingress network is exhausted. Moreover, if the source IP address is spoofed, the reply can more easily exhaust the target victim. According to Arbor Network, DNS based amplification is the most common form of DDoS attack [2].

The wide adoption of DNSSEC also enables a way to get a larger DNS response, and DNSSEC might be used to make potential attacks [66]. If the server has E Gb/s egress capacity, then any outgoing traffic which is greater than E *Gbps*, can overwhelm the egress network bandwidth.

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 [67].

Table 1
Filter parameters.

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
$w_i, \ldots, w_N$.	observ. windows	$2^0, 2^1, \ldots, 2^8$
t_{WR}	deviance threshold	0.5

4.2.2. 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.

4.3. 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 1. For each filter, we measure the performance and operational cost.

4.3.1. 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

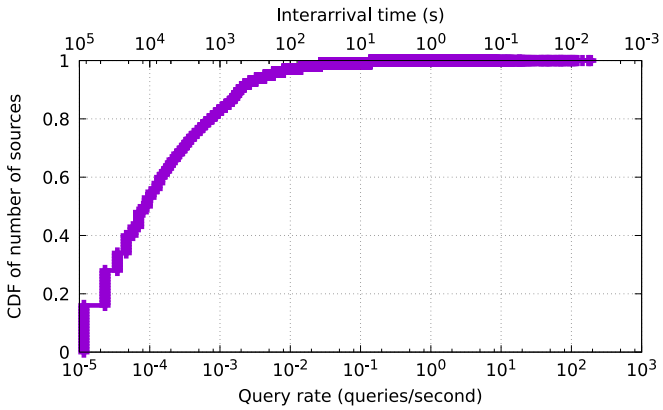


Fig. 2. CDF of source query rates, showing a wide range of rates. Data: 2015-11-29.

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).

4.3.2. 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 min (capture around 90% of traffic sources) to 6 h (capture 99% of traffic sources). We also tested values of U_{UR} of up to 1 day, and the allow-lists were very stable. We selected 2 h for L_{UR}, and 1 day for U_{UR}.

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

The unknown resolver has a number of parameters that we study in Section 4.4.

4.3.3. 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 [25,27] or TTL values for filtering. TTL values are better choice, since they have larger value space, which

```
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. 3. Pseudocode for filter selection.

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 [25,27,68] 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).

4.3.4. Wild recursive filter (WR)

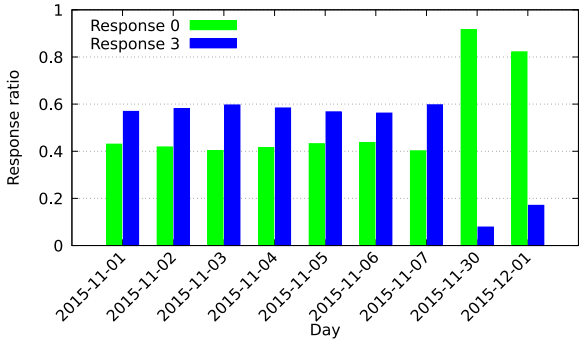
While query rate of different DNS recursives towards a DNS root server varies widely, individual recursives' behaviors are mostly consistent over short time periods (e.g., several hours). We leverage this observation to build models of each individual recursive's behavior. The model for a given recursive, 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, $w_1, w_2, w_3, \ldots, w_N$, 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.

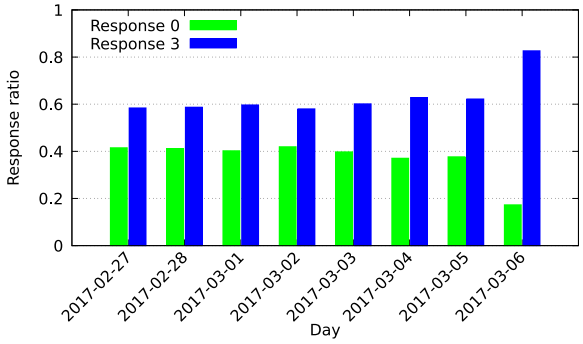
When the attack is detected, the filter measures the current query rates over the same windows. It then calculates the difference between the current rate r_{cw_i} in the window w_i and the rate expected by the model: $mean_{w_i} + 3 \times std_{w_i}$. We then calculate a smoothed, normalized deviance 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 min and 6 h. 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 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.



(a) Rcode ratio changed during November 30, 2015 and Decembe events



(b) Rcode ratio changed during attack event of March 6, 2017

Fig. 4. Rcode trend during normal and attack traffic in root A.

4.3.5. 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 [24] replies, while attackers often query for random query names.

We therefore considered a filter based on response codes that discards NXDomain responses (Response code 3 as shown on 2017-03-06 in Fig. 4(b)). Unfortunately, more than 60% of *root* DNS traffic involves non-existing TLDs as shown in the seven days of 2015 and 2017 (Fig. 4). We also find that sometimes the attack query names have an "NoError" response in the replies (Response code 0 in Fig. 4(a)), and we cannot filter out "NoError" replies. Thus for root DNS traffic, a response code filter will have large collateral damage, and we do not currently include it in *DDiDD*.

4.3.6. 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

Table 2
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 s.

PoP	Date	Start	dur	ULQ	DNS	FQ		UR		HC		WR		$DDiDD_F$		$DDiDD_P$	
		(UTC)	(sec)		mon	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**

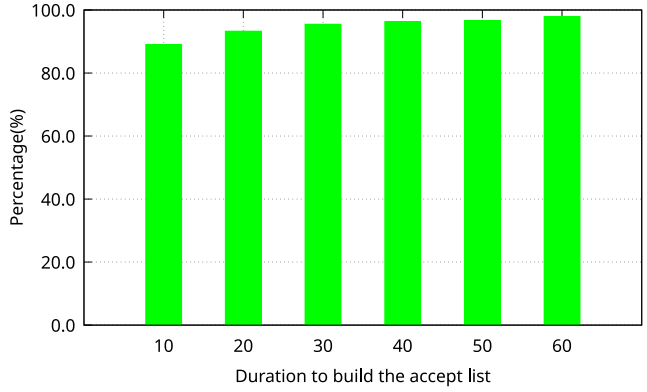


Fig. 5. Impact of the duration to build the resolver list.

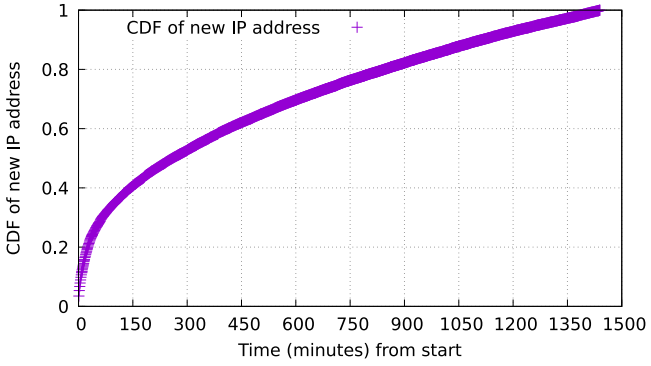


Fig. 6. CDF of new IP address with time.

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*.

4.4. Parameter validation for the unknown resolver filter

Next, we validate our choice of the parameters for unknown resolver filter. We will show how we choose observation period, L_{UR}, and observation frequency, U_{UR}.

How long to observe? We want to observe long enough to get most legitimate sources, so we first consider how often each source makes queries. Fig. 2 shows that sources place queries at many different rates with a long tail, with about 80% having inter-arrivals of 1000 s or more, while a few place thousands of queries per second. This wide range of rates is also visible in earlier studies [33,69].

We are certain to capture all frequent queriers in our accept list, as a relatively short observation period (2 h) provides a list that covers the majority of queries. However, a short observation will miss the infrequent queriers. We next quantify how much queries and unique sources we can cover based on an observation duration.

We expect shorter observations for hitlists to observe only the most frequent carriers, hence, most legitimate queries. To evaluate how long we must observe to get the most queries, we examine normal traffic of the day 2015-11-29 and build a list from data lasting from 10 min to 60 min starting from 00:00:00 UTC. We evaluate how many future queries these lists can identify from 02:00:00 UTC to 23:59:59 UTC traffic. Fig. 5 shows only 20 min of traffic can cover over 90% of future queries. Hence, only a small duration of traffic can cover the sources which will make most of the future queries of the day.

Although small duration *covers* frequent queriers which make most queries, we still *miss* many non-frequent queriers. From Fig. 6, we can see that within a ~1400 min time-frame, we get 50% of unique sources within ~250 min. This implies that even if we create the accept list with 250 min of traffic, we still miss 50% of the future unique legitimate sources (not the number of queries). However, these infrequent sources send very few queries.

We choose to take 2 h of traffic to build the known resolver list. According to Fig. 6, we expect to miss around 60% of the unique sources of the day (though attacks do not persist the whole day). We choose this value because this value is sufficient to cover most of the legitimate future queries (Fig. 5). Also, non-frequent queriers mostly have well-configured cache, and most of the time they get their query response from their own cache.

How often to build? We next consider how often we need to build a known resolver list, confirming that once a day is sufficient (Section 4.3.2).

How often the list is built can influence its success. We must build the list frequently enough that it reflects current queriers, yet list generation has some cost so we cannot build it continuously. To evaluate how list age changes accuracy, we create lists from 2 h of data starting 0 to 960 min before the attack. We compute the confusion matrix to see how much malicious traffic we can block along with the collateral damage for 2017-03-06 event. From Fig. 7, we can see little change in the confusion matrix even if we build the known resolver list 960 min before the attack event. Sensitivity and specificity values differ by 1% to 2% based on the creation time (sensitivity means how much malicious traffic we can detect and specificity means how much legitimate traffic we can detect).

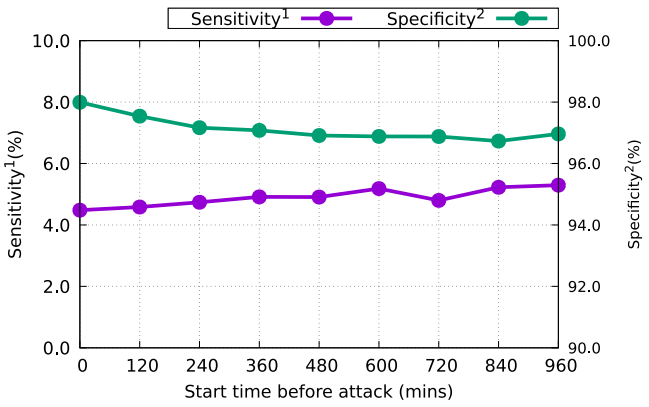


Fig. 7. Impacts of the accept list creation time based on confusion matrix for 2017-03-06 event considering all queries.

Since time of list construction has relatively little effect on effectiveness, we build the resolver accept list once a day.

4.5. 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 Fig. 3. 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 Section 4.2) 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.

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 h, and then keep updating WR entries each 20 min 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 (Fig. 8).

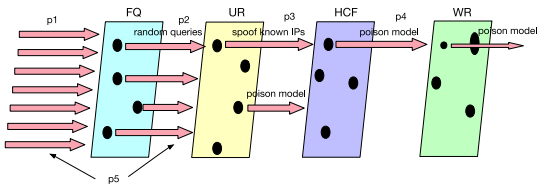


Fig. 8. Swiss cheese model of defense.

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 (Fig. 8) 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 WR filter (thin red arrow at the top right of Fig. 8), 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.

5. Evaluation

We use datasets containing real DNS root traffic and attacks (Section 5.1) to calculate success metrics (Section 5.2) that characterize *DDiDD* performance (Section 5.3).

5.1. Datasets

We use datasets collected at *B-root*, one of 13 root identifiers. These datasets are publicly available [28]. 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 2). 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 2. DNSmon reports the fraction of responses received by many (about 100) physically distributed probers, which query each DNS root every 10 min. In Table 2, the first three attack events had a large impact, showing 99%–100% of unanswered queries, as publicly reported [11–13]. The other seven events had smaller impacts (1%–7% unanswered queries), because they were shorter (5 min 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 [70,71]. 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.

5.2. 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%.

5.3. DDiDD Performance

Table 2 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 s.

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 2. 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 min 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 s.

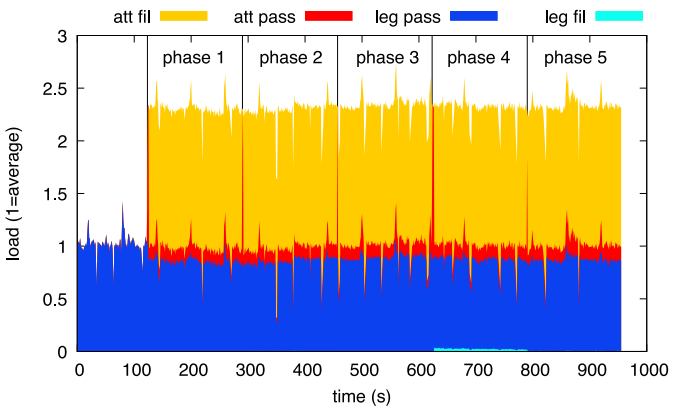


Fig. 9. *DDiDD* evaluation for a synthetic polymorphic attack.

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 Fig. 9. We create a synthetic attack by mixing legitimate traffic from February 2017 with five synthetic attacks, which correspond to p1–p5 labels in Fig. 8: (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. Fig. 9 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.

5.4. Impacts on resource consumption

We next look in detail about how *DDiDD* handles a DDoS attack. To evaluate the performance, we conduct controlled experiments in the DETER testbed (Section 5.4.1), evaluating resource consumption at a simulated target.

5.4.1. Experimental setup

We replay traffic with LDplayer [72], using 22 clients to reproduce the full, original bitrate. Fig. 10 shows the experimental setup to replay an attack event to the server. Fig. 10 also illustrates how the different components of our system interact each other to mitigate the attack.

The attack target is an emulated DNS root server. We implement it with BIND, using the LocalRoot method of providing root service [73]

Reproducing viable events: To show the effects of attacks that drive a production system to resource exhaustion requires many servers to attack and to emulate the service. It is also difficult to perfectly reproduce attacks since the stored traces are often unable to capture the entire attack because of limitations of the capture system. We therefore

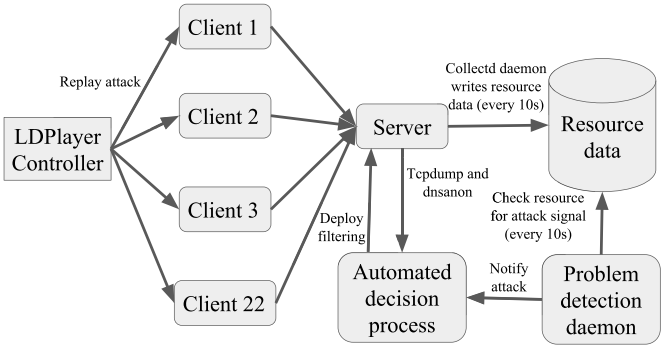


Fig. 10. Experimental setup and the interaction with our automated system.

scale down the server capacity to match the stored traces. We measure the regular traffic resource consumption, and trigger a problem when the resource is double than the regular consumption.

After filtering, can we reduce the resource consumption?: We consider the resource consumption before and after deploying our system.

From Fig. 11, we can see the comparison of resource consumption for the 2017-03-06 event. CPU usage reduces from ~62% (top row, second graph from left) to ~48% (bottom row, second graph from left) using our system. In case of egress network bandwidth, we can reduce the bandwidth from ~0.8 Gb/s (top row, third graph from left) to ~0.3 Gb/s (bottom row, third graph from left). We cannot reduce ingress network bandwidth as we have to give access before making any filtering. We do not find any effect over memory during the attack and after deploying the system (we ignore that in Fig. 11).

Responding to polymorphic attacks: Our system periodically evaluates the traffic to address polymorphic attacks that change attack methods during the event. We next look at the 2017-03-06 event to see how our system copes with changing attacks.

The top-leftmost graph of Fig. 11 shows the polymorphic nature of 2017-03-06 event—attack starts (first red line), pause (green line), and then starts again with a new query name (last red line). From the bottom row-leftmost graph of Fig. 11, we can see that our system deploys the best filter quickly (first blue line from the left), keeps the best filter until a temporary stop in attack at ~89 minutes, reacts accordingly to stop filtering (middle blue line), and deploys the best filter again when a different attack starts again (last blue line from the left). This shows our system is adaptive to the polymorphic attack events.

6. 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 polymorphic attack events, which change attack pattern during an ongoing attack event, and nimbly makes new filter selection in up to 4 s. 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 show our system successfully reduces resource consumption during replay of a real-world attack. We release *DDiDD* as open source.

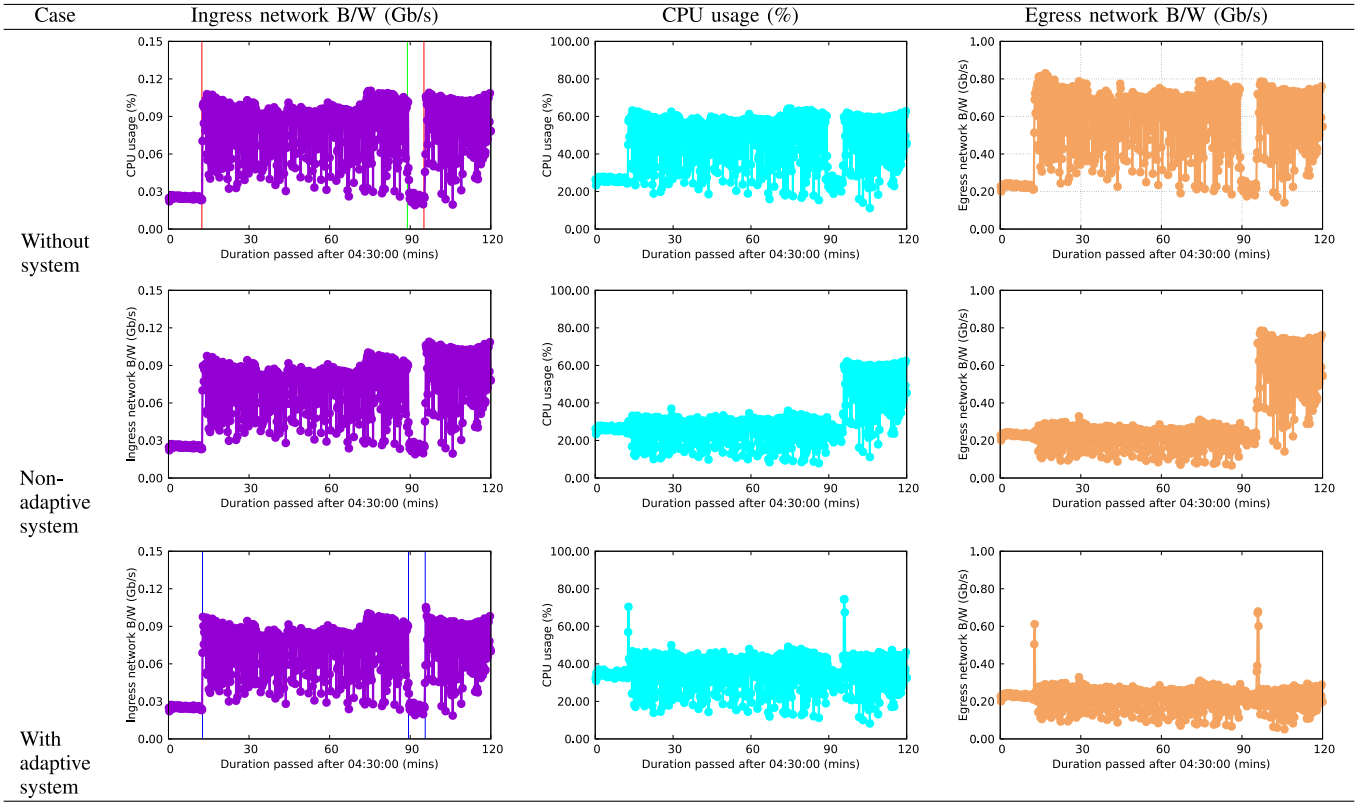


Fig. 11. Resource consumption comparison for 2017-03-06 event: top row shows resources when we do not deploy the automated system, middle row shows when our system is not adaptive to the changes during an attack, bottom row shows resources when we deploy the automated adaptive system. We ignore memory graph as memory remains consistent over experiment.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: John Heidemann reports financial support was provided by National Science Foundation. John Heidemann reports financial support was provided by US Department of Homeland Security.

Data availability

Data will be made available on request.

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] Akamai InfoSec, A look back at the DDoS trends of 2018, 2017, [Online]. https://blogs.akamai.com/2019/01/a-look-back-at-the-ddos-trends-of-2018.html. (Accessed 31 May 2019).
[2] A. Network, NETSCOUT Arbor's 13th annual worldwide infrastructure security report, 2019, [Online]. https://resources.netscout.com/threat-report-archives/13th-worldwide-infrastructure-security-report. (Accessed 31 May 2019).
[3] A. Toh, Azure DDoS protection—2021 Q1 and Q2 DDoS attack trends, 2021, [Online]. https://azure.microsoft.com/en-us/blog/azure-ddos-protection-2021-q1-and-q2-ddos-attack-trends/. (Accessed 23 October 2021).
[4] T. Emmons, 2021: Volumetric DDoS attacks rising fast, 2021, [Online]. https://www.akamai.com/blog/security/2021-volumetric-ddos-attacks-rising-fast/. (Accessed 23 October 2021).
[5] F. David Warbuton, Ddos attack trends for 2020, 2020, [Online]. https://www.f5.com/labs/articles/threat-intelligence/ddos-attack-trends-for-2020. (Accessed 7 October 2021).
[6] L.H. Newman, Github servived the biggest DDoS attack ever recorded, 2018, [Online]. https://www.wired.com/story/github-ddos-memcached/. (Accessed 19 March 2018).
[7] ZD Net, Cloudflare says it stopped the largest DDoS attack ever reported, 2020, [Online]. https://www.zdnet.com/article/cloudflare-says-it-stopped-the-largest-ddos-attack-ever-reported/, (Accessed 7 October 2021).
[8] The Guardian, DDoS attack that disrupted internet was largest of its kind in history, experts say, 2016, [Online]. https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet. (Accessed 24 October 2021).
[9] L. Adriano, Canadian comms company suffers DDoS attack, 2021, https://www.insurancebusinessmag.com/ca/news/breaking-news/canadian-comms-company-suffers-ddos-attack-310819.aspx.
[10] P. Vixie, G. Sneeringer, M. Schleifer, Events of 21-oct-2002, 2002, web page http://c.root-servers.org/october21.txt.
[11] Root server operations, Events of 2015-11-30, 2018, [Online]. http://root-servers.org/media/news/events-of-20151130.txt. (Accessed 17 January 2018).
[12] Root server operations, Events of 2016-06-25, 2018, [Online]. http://root-servers.org/media/news/events-of-20160625.txt. (Accessed 17 January 2018).
[13] G.C. Moura, R. d. O. Schmidt, J. Heidemann, W.B. de Vries, M. Muller, L. Wei, C. Hesselman, Anycast vs. DDoS: Evaluating the November 2015 root DNS event, in: Proceedings of the 2016 Internet Measurement Conference, IMC '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 255–270.
[14] B. Schneier, Lessons from the Dyn DDoS attack, 2016, [Online]. https://www.schneier.com/blog/archives/2016/11/lessons_from_th_5.html. (Accessed 21 June 2018).
[15] M. Calder, A. Flavel, E. Katz-Bassett, R. Mahajan, J. Padhye, Analyzing the performance of an anycast CDN, in: Proc. of ACM IMC, ACM, Tokyo, Japan, 2015, pp. 531–537.
[16] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, B. Weihl, Globally distributed content delivery, IEEE Internet Comput. 6 (5) (2002) 50–58.
[17] J. Pang, J. Hendricks, A. Akella, R.D. Prisco, B. Maggs, S. Seshan, Availability, usage, and deployment characteristics of the Domain Name system, in: Proc. of ACM IMC, ACM, Taormina, Sicily, Italy, 2004, pp. 123–137.
[18] G.C.M. Moura, J. Heidemann, M. Müller, R. de O. Schmidt, M. Davids, When the dike breaks: Dissecting DNS defenses during DDoS, in: Proc. of ACM IMC, 2018.
[19] A. Rizvi, L. Bertholdo, J. Ceron, J. Heidemann, Anycast agility: Network playbooks to fight DDoS, in: 31st USENIX Security Symposium, USENIX Security 22, USENIX Association, Boston, MA, 2022, pp. 4201–4218.
[20] G. Oikonomou, J. Mirkovic, Modeling human behavior for defense against flash-crowd attacks, in: 2009 IEEE International Conference on Communications, IEEE, 2009, pp. 1–6.
[21] S. Ramanathan, J. Mirkovic, M. Yu, Blag: Improving the accuracy of blacklists, in: NDSS, 2020.
[22] R. Tandon, J. Mirkovic, P. Charnsethikul, Quantifying cloud misbehavior, in: 2020 IEEE 9th International Conference on Cloud Networking, CloudNet, IEEE, 2020, pp. 1–8.
[23] R. Tandon, A. Palia, J. Ramani, B. Paulsen, G. Bartlett, J. Mirkovic, Defending web servers against flash crowd attacks, in: International Conference on Applied Cryptography and Network Security, Springer, 2021, pp. 338–361.
[24] K. Schomp, O. Bhardwaj, E. Kurdoglu, M. Muhaimen, R.K. Sitaraman, Akamai DNS: providing authoritative answers to the world's queries, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, 2020, pp. 465–478.
[25] H. Wang, C. Jin, K.G. Shin, Defense against spoofed IP traffic using hop-count filtering, IEEE/ACM Trans. Netw. 15 (1) (2007) 40–53.
[26] C. Jin, H. Wang, K.G. Shin, Hop-count filtering: an effective defense against spoofed DDoS traffic, in: Proceedings of the 10th ACM Conference on Computer and Communications Security, ACM, 2003, pp. 30–41.
[27] A. Mukaddam, I. Elhajj, A. Kayssi, A. Chehab, IP spoofing detection using modified hop count, in: 2014 IEEE 28th International Conference on Advanced Information Networking and Applications, 2014, pp. 512–516.
[28] Analysis of network traffic (ANT) group, ANT datasets, 2022, [Online]. https://ant.isi.edu/datasets/all.html. (Accessed 19 February 2022). Datasets with Anomaly keywords.
[29] USC/ISI, Ddos defense in depth for dns (ddidd) tools, 2022, [Online]. https://ant.isi.edu/software/ddidd/index.html. (Accessed 24 November 2022).
[30] A. Rizvi, J. Mirkovic, J. Heidemann, W. Hardaker, R. Story, Defending root DNS servers against DDoS using layered defenses, in: Proc. of IEEE International Conference on Communications Systems and Networks, COMSNETS, IEEE, Bengaluru, India, 2023 (in press). Awarded best paper.
[31] A. root, rcode-volume, 2022, [Online]. https://a.root-servers.org/rssac-metrics/raw/2022/01/rcode-volume/. (Accessed 24 January 2022).
[32] B. root, rcode-volume, 2022, [Online]. https://b.root-servers.org/rssac/2022/01/rcode-volume/. (Accessed 24 January 2022).
[33] S. Castro, D. Wessels, M. Fomenkov, K. Claffy, A day at the root of the Internet, ACM SIGCOMM Comput. Commun. Rev. 38 (5) (2008) 41–46.
[34] C. Duckett, Chromium DNS hijacking detection accused of being around half of all root queries, 2020, [Online]. https://www.zdnet.com/article/chromium-dns-hijacking-detection-accused-of-being-around-half-of-all-root-queries/. (Accessed 24 January 2022).
[35] B.-R. Operators, B-root statement of operational principles, 2008, Web page https://b.root-servers.org/statements/operation.html.
[36] Bill Slater, President of Chicago ISOC, The internet outage and attacks of october 2002, 2002, https://billslater.com/writing/2002_1107_Internet_Outage_and_Attacks_in_october_2002_by_William_Slater.pdf.
[37] ICANN, FACTSHEET: Root server attack on 6 february 2007, 2007, https://www.icann.org/en/system/files/files/factsheet-dns-attack-08mar07-en.pdf.
[38] G.C. Moura, J. Heidemann, R. d. O. Schmidt, W. Hardaker, Cache me if you can: Effects of DNS time-to-live, in: Proceedings of the Internet Measurement Conference, 2019, pp. 101–115.
[39] T. Koch, K. Li, C. Ardi, E. Katz-Bassett, M. Calder, J. Heidemann, Anycast in context: A tale of two systems, in: Proc. of ACM SIGCOMM, ACM, 2021, Virtual.
[40] C. Barna, M. Shtern, M. Smit, V. Tzerpos, M. Litoiu, Model-based adaptive DoS attack mitigation, in: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 119–128.
[41] S. Kandula, D. Katabi, M. Jacob, A. Berger, Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds, in: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2, USENIX Association, 2005, pp. 287–300.
[42] E.Y. Chen, M. Itoh, A whitelist approach to protect SIP servers from flooding attacks, in: Communications Quality and Reliability (CQR), 2010 IEEE International Workshop Technical Committee on, IEEE, 2010, pp. 1–6.
[43] M. Yoon, Using whitelisting to mitigate DDoS attacks on critical internet sites, IEEE Commun. Mag. 48 (7) (2010) 110–115.
[44] T. Peng, C. Leckie, K. Ramamohanarao, Proactively detecting distributed denial of service attacks using source IP address monitoring, in: International Conference on Research in Networking, Springer, 2004, pp. 771–782.
[45] P. Ferguson, D. Senie, Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing, 2000, RFC 2267, internet-rfc. also BCP-38.
[46] J. Levine, DNS blacklists and whitelists, 2010, RFC 5782, IETF.
[47] D. Gillman, Y. Lin, B. Maggs, R.K. Sitaraman, Protecting websites from attack with secure delivery networks, Computer 48 (4) (2015) 26–34.
[48] J. Blazina, Stonefish—automating DDoS mitigation at the edge, 2019, [Online]. https://medium.com/@verizondigital/stonefish-automating-ddos-mitigation-at-the-edge-6a2650aeb6af. (Accessed 30 May 2019).

[49] Yoachimik. O., Who DDoS'd Austin?, 2019, [Online]. https://blog.cloudflare.com/who-ddosd-austin/. (Accessed 02 December 2019).

[50] A. Fabre, L4Drop: XDP DDoS mitigations, 2018, [Online]. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/. (Accessed 01 December 2019).

[51] M. Sung, J. Xu, IP traceback-based intelligent packet filtering: a novel technique for defending against internet DDoS attacks, IEEE Trans. Parallel Distrib. Syst. 14 (9) (2003) 861–872.

[52] Z. Duan, X. Yuan, J. Chandrashekar, Controlling IP spoofing through interdomain packet filters, IEEE Trans. Dependable Secur. Comput. 5 (1) (2008) 22–36.

[53] A. Yaar, A. Perrig, D. Song, StackPi: New packet marking and filtering mechanisms for DDoS and IP spoofing defense, IEEE J. Sel. Areas Commun. 24 (10) (2006) 1853–1863.

[54] R. Thomas, B. Mark, T. Johnson, J. Croall, Netbouncer: client-legitimacy-based high-performance DDoS filtering, in: DARPA Information Survivability Conference and Exposition, 2003. Proceedings, volume 1, IEEE, 2003, pp. 14–25.

[55] I. S. C. (ISC), Using the response rate limiting feature, 2018, [Online]. https://kb.isc.org/docs/aa-00994. (Accessed 05 May 2019).

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

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

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

[59] M. Jonker, A. King, J. Krupp, C. Rossow, A. Sperotto, A. Dainotti, Millions of targets under attack: a macroscopic characterization of the DoS ecosystem, in: Proceedings of the 2017 Internet Measurement Conference, 2017, pp. 100–113.

[60] K. McCarthy, Internet's root servers take hit in DDoS attack, 2015, [Online]. https://www.theregister.co.uk/2015/12/08/internet_root_servers_ddos/. (Accessed 29 Janurary 2019).

[61] Imperva, Different attack description, 2015, [Online]. https://www.imperva.com/docs/DS_Incapsula_The_Top_10_DDoS_Attack_Trends_ebook.pdf. (Accessed 19 September 2017).

[62] C. Kaufman, R. Perlman, B. Sommerfeld, DoS protection for UDP-based protocols, in: Proceedings of the 10th ACM Conference on Computer and Communications Security, ACM, 2003, pp. 2–7.

[63] L. Zhu, Z. Hu, J. Heidemann, D. Wessels, A. Mankin, N. Somaiya, T-DNS: Connection-oriented DNS to improve privacy and security, ACM SIGCOMM Comput. Commun. Rev. 44 (4) (2015) 379–380.

[64] W. Eddy, et al., TCP SYN Flooding Attacks and Common Mitigations, Technical Report, 2007, RFC 4987.

[65] K. Tzvetanov, DDoS mitigation tutorial NANOG 69, 2017, [Online]. https://www.nanog.org/sites/default/files/DDoSTutorial-NANOG69-v3.pdf. (Accessed 31 January 2018).

[66] R. van Rijswijk-Deij, A. Sperotto, A. Pras, DNSSEC and its potential for DDoS attacks: a comprehensive measurement study, in: Proceedings of the 2014 Conference on Internet Measurement Conference, ACM, 2014, pp. 449–460.

[67] A. Rizvi, J. Heidemann, J. Mirkovic, Dynamically Selecting Defenses to DDoS for DNS (Extended), Technical Report ISI-TR-736, USC/Information Sciences Institute, 2019.

[68] M. Backes, T. Holz, C. Rossow, T. Rytilahti, M. Simeonovski, B. Stock, On the feasibility of TTL-based filtering for DRDoS mitigation, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Vol. 9854, RAID, 2016, pp. 303–322.

[69] D. Wessels, M. Fomenkov, Wow, That's a lot of packets, in: Passive and Active Network Measurement Workshop, PAM, PAM, San Diego, CA, 2003.

[70] J. Xu, J. Fan, M.H. Ammar, S.B. Moon, Prefix-preserving IP address anonymization: Measurement-based security evaluation and a new cryptography-based scheme, in: 10th IEEE International Conference on Network Protocols, 2002. Proceedings, IEEE, 2002, pp. 280–289.

[71] L. Zhu, J. Heidemann, Dnsanon: extract DNS traffic from pcap to text with optionally anonymization, 2017, [Online]. https://ant.isi.edu/software/dnsanon/index.html. (Accessed 20 January 2018).

[72] L. Zhu, J. Heidemann, LDplayer: DNS experimentation at scale, in: Proceedings of the Internet Measurement Conference 2018, ACM, 2018, pp. 119–132.

[73] W. Hardaker, LocalRoot: Serve yourself, 2018, [Online]. https://localroot.isi.edu/. (Accessed 11 January 2019).

A.S.M. Rizvi is a Ph.D. student in Computer Science at the University of Southern California Information Sciences Institute. He works in the Analysis of Network Traffic (ANT) lab, which Prof. John Heidemann leads. His research interests include Networking, Security, and Internet Measurement. He received his B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET).

Jelena Mirkovic is Project Leader at USC/ISI and research faculty at USC. She received her MS and Ph.D. from UCLA, working in the LASR group, lead by Prof. Peter Reiher. She received BS in Computer Science and Engineering from School of Electrical Engineering, University of Belgrade, Serbia. Jelena's research interests span networking and security fields. Her current research is focused on several network security problems: botnets, denial-of-service attacks, and IP spoofing. Additionally, she is interested in methodologies for conducting security experiments and she is working with colleagues at USC/ISI on improving DeterLab testbed.

John Heidemann is a principal scientist at the University of Southern California/Information Sciences Institute (USC/ISI) and a research professor at USC in Computer Science. At ISI he leads the ANT (Analysis of Network Traffic) Lab, studying how to observe and analyze Internet topology and traffic to improve network reliability, security, protocols, and critical services. He received his Ph.D. from UCLA in 1995, and his BS from the University of Nebraska-Lincoln in 1989. He is a senior member of ACM and fellow of IEEE.

Wes Hardaker is Senior Computer Scientist at USC's Information Sciences Institute. His research focuses on developing research programs to enhance the security of the DNS and other Internet protocols. He is principally responsible for the operation and management of the Internet's b.root-servers.net DNS critical infrastructure. He actively participates in the Internet Engineering Task Force and is a member of the Internet Architecture Board (IAB) and on the board of directors for the Internet Corporation for Assigned Names and Numbers (ICANN).

Robert Story is a Lead Research Engineer in the Networking and Cyber Security Division at the University of Southern California's Information Sciences Institute. He improves and maintains the B-Root authoritative server infrastructure, one of the 13 DNS root name servers. He also provides development and IT support to various research projects at ISI. He can often be found at DNS, Network Operator, and Internet protocol conferences.