Optimizing DNS Resolvers for High Loads

Itay Alayoff, Gil Einziger
Computer Science Department, Ben Gurion University of the Negev,Be'er Sheba, Israel.
itayala@post.bgu.ac.il, gilein@bgu.ac.il

Index Terms—**component, formatting, style, styling, insert**

***Abstract*—Handling volumetric attacks and high loads is fundamental for Domain Name System (DNS) resolvers. While one can often scale up resources to cope with high loads, this costly solution is not always viable. Thus, our work revisits the design of DNS resolvers and maps the design decisions that most impact the resolver's robustness to high loads and volumetric attacks. Our work suggests a novel resolver composed of deadline-aware queue management, optimized cache policy, and high-load handling mechanisms. An evaluation with real DNS traces shows a consistent improvement of 21% in a high load scenario and by 40% under an NXDomain attack when compared with the best alternative method.**

***Index Terms*—DNS Resolver, Volumetric Attacks, Cache Policy, Queue Management, Count-Min Sketch.**

I. Introduction

Volumetric attacks on critical Internet infrastructure are an increasing concern. *Domain Name Systems (DNS)* provides a mapping between easy-to-remember domain names and the IP addresses of the servers that host the site. DNS is perhaps the most critical Internet infrastructure since most users cannot access the Internet when denied access to a DNS server [1]. The Dyn attack in October 2016 exemplified the vulnerability of the DNS infrastructure to volumetric attacks. In this attack, millions of users in the eastern USA were denied Internet access for an extended time.

The attack was a *Nonexistent Domain (NXDomain)* distributed denial of service attack [2]. In this attack, a large number of compromised devices (also known as a botnet) launched DNS queries for nonexistent domains to Dyn's servers. Such queries require more resources from the DNS servers, and Dyn could not handle the massive attack traffic. As a result, large parts of the eastern USA lost Internet access for several hours. By taking down a single service, the attacker effectively impacted banks, electronic commerce sites such as eBay and Amazon, news sites, and many more.

In retrospect, the Dyn attack was not sophisticated and did not expose new attack vectors. It uncovered that the standard mitigation techniques often boil down to additional resources. However, Akamai failed to mitigate the attack despite their extensive resources. We are entering an era where volumetric attacks challenge even the industry giants. Thus, we focus on designing better DNS servers capable of handling more load. Numerous works address hardening DNS servers about specific attacks or inefficiencies [2]–[6]. However, these are ad-hoc solutions, and we miss the wide perspective of how design decisions impact performance.

Our Contribution: Our work suggests the *Resolver Aware Remaining Time (RART)* design for DNS resolvers based on the remaining time to serve queries. Rart is a complete overhaul of the current designs, and it focuses on queue management, caching, and high-load optimization. Through an evaluation of real datasets, we demonstrate a 21% improvement in the goodput for handling high load scenarios (of legitimate traffic) and 40% for handling NXDomain attacks when compared against leading open source DNS resolvers.

Our approach is based on informed risk management, where we see handling a query as a gambit where if we fail to finish in time, we waste the resolver's resources. As such, RART would perform all queries like a normal resolver when the load is below capacity. As the load intensifies, the queue length increases and RART may selectively drop some of the queries to maximize the number of completed queries.

Our work also provides insight into the delicate interplay of the various design decisions in resolver design, including queue management, cache management, and high-load management. We perform a step-by-step evaluation and explain the tradeoffs encountered at each decision interval. Such a systemic examination of the resolver's design is novel as the existing works each focus only on one of the aspects (e.g., queue management, cache policy, or attack mitigation).

Roadmap: Section II surveys the related work for each of the design decisions we study within this work. Section III gradually explains our solution, starting by surveying the model and assumptions, followed by explaining the design of popular DNS resolvers, and then presents our approach as a series of answers to each design point. Section IV provides an evaluation using real workloads that motivates our design and positions it within the context of the existing DNS resolvers. Finally, Section V summarizes our results and discusses their impact, as well as future directions.

II. Related Work and Motivation

Designing robust DNS resolvers is a multidisciplinary process that revolves around queue management, cache policy, and attack mitigation schemes. This section surveys important works in each field and important DNS resolvers. **Queue Management:** Queue management is crucial for effectively handling high loads and avoiding timeouts. Most server operating systems have a *First in First Out (FIFO)* queue. If a query arrives and the queue is full, the server drops that query (e.g., *TailDrop policy*). The operating system that runs the DNS resolver directly affects the order in which queries

ISBN 978-3-903176-57-7

are handled. Additionally, some DNS resolvers have a rate-limiting mechanism to prevent excessive loads [7], [8]. For instance, BIND serves the first query to arrive and has two thresholds for the number of recursive queries [9].

Some resolvers even have more sophisticated queuing systems. Akamai DNS [10] has a query scoring and prioritization mechanism as part of their nameservers. It prevents malicious queries from exhausting server resources. A nameserver gives a penalty score to each query. Low score queries receive a higher execution priority as they are more likely to be legitimate traffic. More so, queries with a penalty score above a certain threshold are dropped.

Queue management is also very common in packet-switched networks to avoid congestion. The queue management governs which packets are buffered, transmitted, and discarded. It can either be active or passive. Passive schemes take action only when the queue is full. For example, *TailDrop* rejects an incoming packet when the queue is full, while *HeadDrop* drops the oldest packet in the queue. Active schemes try to avoid problems rather than react to them. RED [11] randomly discards packets with a probability that depends on the queue length. While CoDel [12] drops packets based on the queue delay (rather than the queue length) to avoid timeouts.

Cache Policy The research of cache policies advances in two parallel tracks: Some works focus on DNS-specific challenges to the cache performance, such as handling "one-time" legitimate queries called *disposable domain names* that are not cachable. In comparison, other works focus on cache policies in general. Hao et al. [13] and Yu et al. [14] provide a fine example for DNS specific optimizations. Their works use machine learning classifiers that train on syntactical features extracted from the hostname string. The classifiers identify disposable domain names and mark these to avoid the cache. Yuchi et al. [15] suggest a segmented cache strategy to deal with disposable domain names. Their caching strategy reduces DNS traffic between the resolver and the authoritative servers.

Within a wider scope, caching is a common optimization shared by many systems. Thus, designing cache policies for general use attracts a lot of research. In a gist, cache policies decide (a) if to admit a new item to the cache and (b) which item to evict if the cache is full. The *hit ratio* measures how many queries are satisfied by the cache (at a reduced cost). The simplest cache policy is FIFO (*First in first out*), the policy always admits new items to the cache and evicts the item that entered the cache first. FIFO is considered a bit simplistic as it also evicts items that are accessed a lot. The most well-known cache policy is arguably the *Least Recently Used (LRU)* policy. LRU always admits new items and evicts the least recently *accessed* items. LRU can be implemented by keeping the items ordered by their last access time. The TinyLFU [16] policy records the number of access to items across some time period. It admits a new item to the cache if it is more frequent than the cache victim. Any cache eviction policy can select the cache victim (e.g., by LRU). Hyperbolic caching [17] assigns a score to each item based on frequency and recency. Since keeping all items ordered by arbitrary scoring is impractical, Hyperbolic caching randomly samples a small number of entries. It admits a new entry if its score is higher than that of the lowest score item in the sample and evicts the lowest score item.

Interestingly, the general community noticed a similar problem to the DNS disposable domain problem. In many workloads, some items are only accessed once in a long while, which makes them undesirable to cache in most cases. The most common pattern is a database or memory scan. Cache algorithms are called *scanning resilient* if the performance impact of large one-time scans on the algorithm is small. LRU and FIFO are not scanning resilient. For reference, LRU is usually the cache management policy in DNS systems. This weakness explains the need for work on better handling disposable domains. On the other hand, TinyLFU and Hyperbolic caching are scan resilient, which means that adapting them to DNS resolvers also addresses the disposable domain problem and mitigates simple attacks to pollute the cache with single-use nonexistent queries (such as the NXDomain attack).

Attack and high-load mitigation Previous works suggest different approaches to mitigate and cope with volumetric attacks. Some studies focus on changing a particular behavior or property of the DNS infrastructure, while others detect malicious traffic. Ballani et al. [3] suggests storing expired records in a separate "stale cache" for use in case there is no response from the nameservers. Massey et al. [4] propose to set longer time-to-live values for nameserver (NS) resource records since they change infrequently. Other studies built an external system or component to detect malicious traffic. Feibish et al. [2] identify a rise in the number of distinct subdomains of targeted domains and extract attack signatures for mitigation. Kostopoulos et al. [5] proposes a privacy-preserving schema to protect authoritative DNS servers from a flavor of distributed denial of service attacks. Their schema uses probabilistic data structures and related algorithms to classify DNS queries as legitimate or suspicious. Hasegawa et al. [6] present a whitelist filter based on fully qualified domain names for recursive DNS servers.

III. Resolver Aware Remaining Time (RART)

In this section, we gradually design the *Resolver Aware Remaining Time (RART)* resolver. RART's design goal is to maximize the goodput in high-load situations.

System model & Preliminaries: We model DNS resolvers in the following manner: Each query has a timeout (known to the resolver), defining a deadline for handling the query. Queries may arrive at any time, according to the trace. Upon arrival, we enqueue queries in a queue, and multiple workers periodically dequeue queries. The resolver has a fixed-sized cache with some management policy (e.g., LRU), and the worker first checks that cache. If the query response is cached, we have a cache hit; otherwise, it is a cache miss. In DNS resolvers, cache misses result in recursive lookups, which are more time-consuming than cache hits. Our model has some (unknown to the resolver) hit time and recursive time for each query. Such a time may vary between the different domains and subsequent queries for the same domain. The worker then

waits either hit time (for cache hits) or recursive time (for cache misses) and then completes handling the query. The query is successful if a worker finished handling it before the deadline and is unsuccessful otherwise.

The arrival rate is a Poisson process, and we can vary the number of queries entering the resolver on average per time unit. In addition, some of our evaluations simulate NXDomain attacks. In these, we differentiate between legitimate queries from the original workload and attack queries from the simulated attack. The resolver does not know which query is legitimate, but attack queries are never issued twice, and thus the resolver can never benefit from caching them.

We evaluate our resolvers using the *Response Rate* which is the portion of (legitimate) queries that were successfully handled by the resolver. A high response rate is desired. Our work aims to maximize the response rate in situations of high (legitimate) load and NXDomain attacks.

Decision points in DNS design: Our work identifies the following fundamental design decisions that distinguish resolvers.

- **Queue management** - controls how the queue is maintained. E.g., long queues can handle sudden bursts of traffic, but when they are too long, queries may wait too much time in the queue and fail due to a timeout.
- **Cache policy** - in DNS resolver, cache hits do not require a recursive lookup, and we get shorter times to handle cached queries. Thus, a better cache policy would attain more cache hits and increase the response ratio under a heavy load of queries.
- **High load mitigation** - the system may choose to revise its behavior when faced with high loads. For example, we suggest prioritizing popular queries over unpopular queries when the queue gets out of hand. The motivation is to allow the cache to function better in high-load cases.

Design Decisions of open source resolvers:

BIND has a soft and hard limit on the number of concurrent queries it performs [9]. The motivation behind the soft limit is to allow the resolver to handle legitimate queries that can complete quickly. When the number of concurrent queries reaches the soft limit, BIND drops the oldest query and begins executing the new query. Coming to the hard limit, it drops both the pending and new queries. Ideally, the soft limit is the preferred operation mode. BIND's approach is equivalent to *HeadDrop* queue policy. The default value of the soft limit is 90% of the defined resolver capacity.

PowerDNS can be configured to strip the Recursion Desired (RD) bit from any traffic that exceeds a specific rate or to drop it [8]. We focused on the latter option, similar in behavior to *TailDrop*. PowerDNS uses the LRU cache management policy but prioritizes evicting expired records (according to TTL).

Unbound limits the number of queries sent to nameservers. More queries are turned away with an error (SERVFAIL) [7]. Unbound uses the TailDrop queue policy but allows queries to check the cache before queuing them (CacheFirst). To maximize cache performance, Unbound keeps expired entries until the cache runs out of memory; it then uses the LRU policy when it has to evict queries.

In the following subsections, we gradually explain our approach by expanding its answer to each of the above-mentioned design points. For ease of reference, Table I summarizes key details on the modeling of DNS servers.

DNS Resolver	Queue Management	Cache Policy
BIND	HeadDrop	LRU
Unbound	CacheFirst + TailDrop	
PowerDNS	TailDrop	

TABLE I: Design choices in DNS resolvers

A. Queue Aware Remaining Time (QART)

During the dequeue operation, we decide what to do with the query according to its ***remaining time (rt)*** which is the time the query has until it experiences a timeout. [1] Our strategy depends on rt in the following manner:

- $rt < Cache_{th}$ - When we have less than $Cache_{th}$ remaining time, we drop the query with a rationale that there is no time to even look in the cache.
- $Cache_{th} < rt < Attack_{th}$ - The remaining time is between $Cache_{th}$ and below $Attack_{th}$. We perform a cache lookup and return a response in case of a cache hit but drop the query upon a cache miss as there is insufficient time to perform a recursive lookup.
- $Attack_{th} < rt < Recursive_{th}$ - There is enough time to try a recursive lookup with a low success ratio, we will only try a recursive lookup in specific conditions, which we later detail.
- $Recursive_{th} < rt$ - There is enough time to perform a recursive lookup; we handle the query normally. E.g., we perform a cache lookup, and upon a miss, we perform a recursive lookup. This state is the desired situation when the resolver is not overly stressed and resolves all queries.

We determine these thresholds by measuring our system's average performance, as we explain in the following section.

1) Measurement based Threshold Tuning: We adapt the threshold according to the last M measurements of the processing time of queries that returned cached responses and of the M last queries that performed a recursive lookup. Let T_C and T_R denote the last M measurements of queries resolved by cache and recursively, respectively. Upon dequeue, we set the remaining time of a query (rt) as $Timeout$ - qt. Here, qt is the time spent in the resolver's queue, and $Timeout$ is two seconds (common DNS Timeout value [18]). $Cache_{th} = Avg(T_C)$, which implies that the probability of completing a cache lookup (and returning the result) in time is at least 50% providing the distribution is symmetric. $Recursive_{th} = Avg(T_R)$ following a similar logic as $Cache_{th}$, while $Attack_{th}$ is $Min(T_R)$. E.g., we use the high load algorithm when recursive queries can be successful, but not with a very high probability.

[1] Notice that the DNS resolver does not know the exact remaining time as it depends on the RTT between the client and the resolver. However, as most resolvers are close to their clients, and such RTT is likely very short, (e.g., 30 ms) compared to the timeout (e.g., 2 seconds). Thus, we neglect the RTT between the client and resolver.

2) Cache Policy Optimization: A higher chance of hit-ratio translates to faster query handling, positively affecting the resolver's performance. Thus, our work deviates from most existing works that use LRU as their cache management scheme. We choose to use Hyperbolic caching [17] which attained the best performance out of all the cache management policies we checked and better performance than the standard LRU.

In a gist, upon eviction, Hyperbolic caching selects a random subset of the cached items as potential victims. It assigns each of these items a score according to some cost function and evicts the lowest score item. We use the standard priority function, which is the ratio of the number of access and the lifetime of an item.

3) Selective Drop Optimization: When the remaining time is above the attack, but below the recursive threshold ($Attack_{th} < rt < Recursive_{th}$), we have time to look at the cache but not necessarily to perform recursive lookups. Notice that we do not know exactly how much time each recursive lookup would take, but we expect to fail more than 50% of the time. In such conditions, it makes sense to drop some queries to increase our chances of successfully handling other queries. We define two mechanisms to decide which query to perform, and which one to drop.

Delay filter - drop a query with a probability proportional to its remaining time, similar to RED, which drops with a probability proportional to the queue length. Specifically, we use a drop probability of: $\frac{Recursive_{th}-rt}{Recursive_{th}-Attack_{th}}$.

Sketch filter - the delay filter drops packets indistinguishably regardless of their benefit to the resolver. We want to prioritize performing recursive queries for popular domains and drop queries to unpopular domains. Doing so makes sense in high-load situations, as popular queries would benefit the cache. It makes sense under a family of attacks, including the NXDomain attack, as queries to nonexistent subdomains are only issued once (and are thus unpopular).

Ideally, we would like to know exactly how many times each query was issued before. However, keeping exact statistics may be costly to maintain, take a long time to analyze, and take a lot of storage. Instead, we suggest using a (fixed size) Count-Min Sketch [19]. Count-Min-Sketches are probabilistic data structures that approximate frequency estimation using a compact space. They use multiple hash functions and shared counter arrays. Upon each query arrival, we increment the Count-Min Sketch [19] to keep an estimation of the query repetition. Each such increment increases the value of several random counters received from applying a hash function to the item. Thus, if the same item appears again, it increments the same set of counters. When we want to estimate the frequency of a domain, we read its counters and use the minimal value as our estimation.

When using the sketch, we do not need to remove old entries from the sketch (as long as the counters do not overflow). Instead, we periodically calculate the statistical mode of a random row in the Count-Min sketch. In a gist, the statistical mode is the most common value in the row. The statistical mode provides us with a measure to estimate the noise in the sketch, and when an item appears above the noise, it is likely popular. When the Sketch filter is active, we perform recursive queries for domains whose frequency is above the statistical mode of the sketch and drop queries to domains whose frequency is below the statistical mode.

Queue	Parameters
FIFO / TD	$Q_s = W \cdot (Timeout - R_{miss})/R_{miss}$
RED	$max_{th} = W \cdot (Timeout - R_{hit})/R_{miss}$ $min_{th} = W \cdot (Timeout - R_{miss})/R_{miss}$ $max_p = 0.5$ $w_q = 1 - e^{-1/W}$
CoDel	$interval = 10ms$ $target = Timeout - R_{miss}$
QART	$Cache_{th} = Avg(T_C)$ $Attack_{th} = Min(T_R)$ $Recursive_{th} = Avg(T_R)$ $M = 100$

TABLE II: Queue parameters used in our evaluation

IV. Evaluation

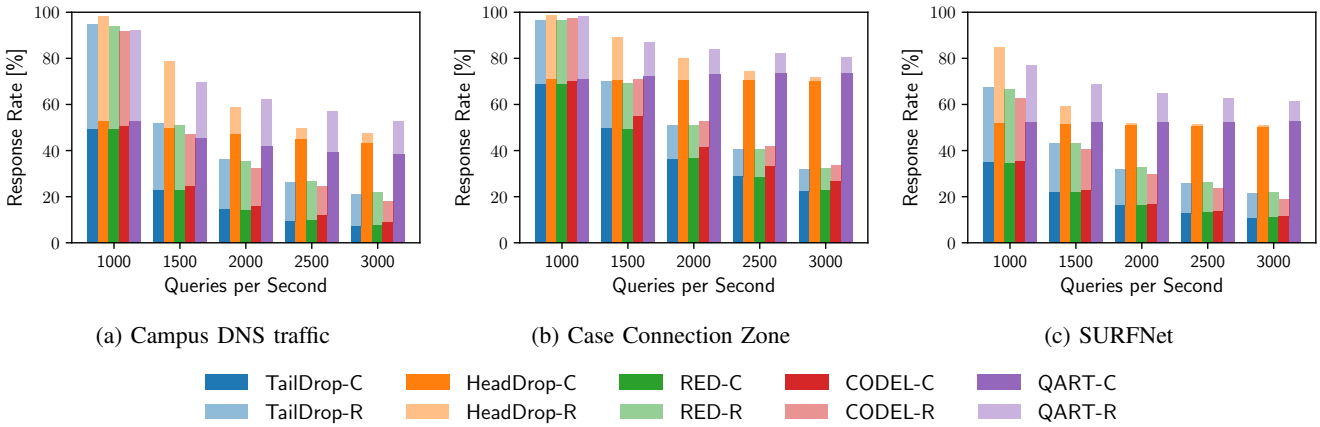
Our evaluation motivates our design when constructing RART. We begin by explaining the different decisions one at a time and comparing them to the existing alternatives. Then, we position RART with other popular DNS resolvers.

Our evaluation includes the following datasets:

- ***SURFnet's Authoritative Servers:*** The trace contains circa 3.5 billion DNS queries received at one of SURFnet's authoritative DNS servers from Google's Public DNS Resolver [20]. We used several day traces.
- ***Campus DNS network traffic:*** A dataset containing DNS queries from over four thousand users was taken on ten random days between April and May 2016 at Thapar University [21].
- ***Case Connection Zone DNS Transaction:*** DNS lookups by client devices on an experimental Fiber-To-The-Home network in Cleveland [22]. This trace contains anonymized domain names to preserves users' privacy.

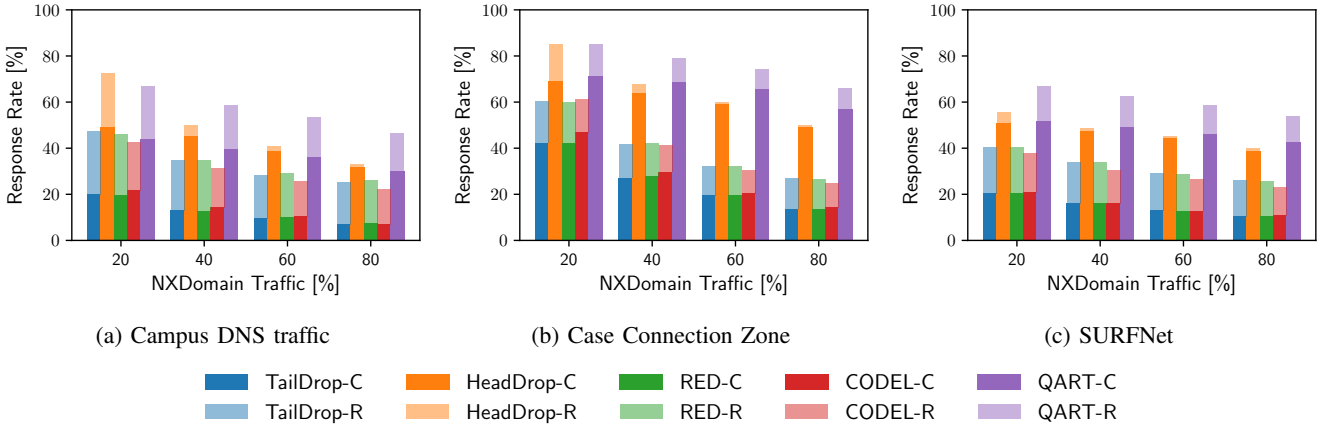
Determining hit and recursive times: Notice that the workloads we use only contain the DNS queries and do not contain the time it takes to retrieve these queries either from cache or recursively. Thus, using the BIND resolver, we measure access times to each query in case of a cache hit and miss. We repeat these measurements multiple times and assign the average access times to each query. In workloads where the queries are anonymized, we randomly select one of Alexa's Top 1 Million websites [23] to replace each anonymized query and then similarly assign the access times.

Attack trace generation: We also generate NXDomain attack traces for our evaluation by injecting NXDomain traffic randomly into the traces. We do that according to a certain percentage, e.g., when the NXDomain traffic is 80%, then 80% of the generated trace contains injected packets, and 20% contains the original packets. Notice that even the legitimate



(a) Campus DNS traffic (b) Case Connection Zone (c) SURFNet

Fig. 1: Response rate for different query rates of resolvers with different queue management. DNS workloads properties: 100K DNS transactions and 15% NXDomain queries.



(a) Campus DNS traffic (b) Case Connection Zone (c) SURFNet

Fig. 2: Response rate for different NXDomain traffic volumes of resolvers with different queue management. DNS workloads properties: 100K DNS transactions with fixed rate of 1000Qps.

DNS workload contains 85% of NOERROR traffic and 15% of NXDomain traffic [24].

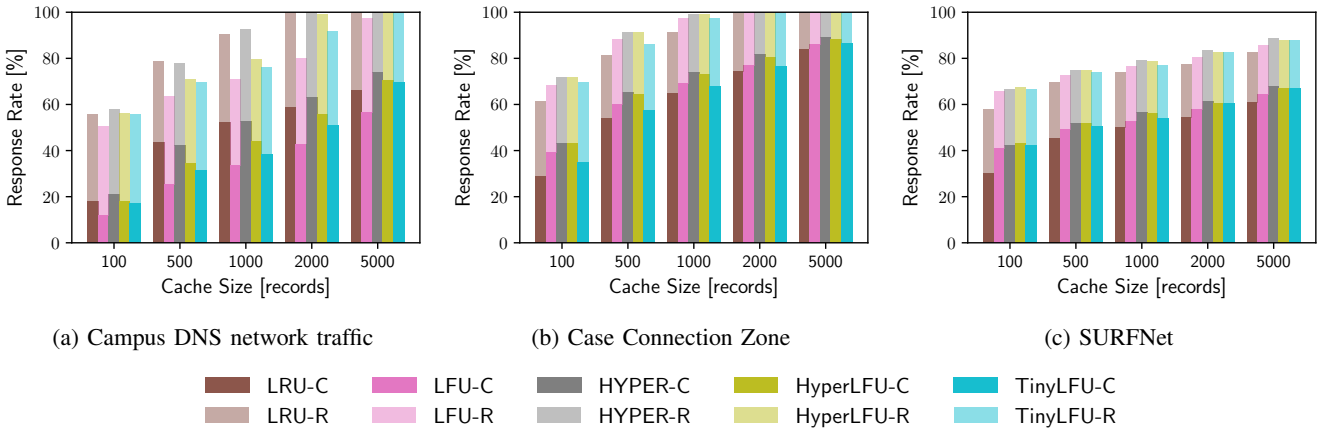
High level simulation description: In our simulations, we enqueue queries from a trace to a queue at a given rate. The DNS resolver uses W workers to dequeue queries from the queue and simulate resolving these queries. When handing a query, a worker first performs a cache lookup, and in the case of a cache hit, it waits the hit time (provided by the trace) and dequeues the next query. Otherwise, the worker waits for a recursive completion time which is included in the trace for that query. Once that time passes, it dequeues another query from the queue.

Queues configuration: Table II shows the full configuration of each queue management scheme. Below we explain the configuration we chose which set the queue parameters conservatively as the cache hit ratio varies. We denote the averages of resolution time by R_{hit} and R_{miss}, calculated by averaging Alexa's resolution times from cache and recursively.

A query that has experienced a queue delay greater than $Timeout - R_{miss}$ has a reasonable chance for a timeout. The queue delay that a query experience in case the resolver has W workers and queue length of Q_l is $Q_d = R_{miss} \cdot \frac{Q_l}{W}$. Considering that the waiting queries require recursive lookup. The queue length equivalent to a queue delay of $Timeout - R_{miss}$: $Timeout - R_{miss} = R_{miss} \cdot \frac{Q_l}{W}$
$Q_l = W \cdot \frac{Timeout - R_{miss}}{R_{miss}}$

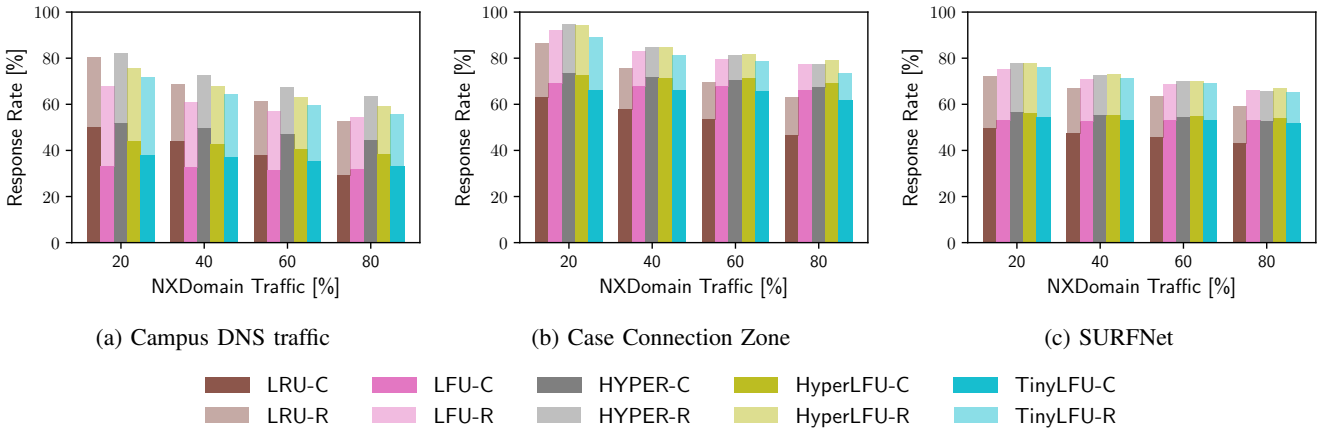
1) RED [11]: RED's max_{th} parameter controls the maximal queue size before RED drops excess queries and we set it to: $W \cdot (Timeout - R_{hit})/R_{miss}$ assuring that only few queued queries experience timeouts. We set the min_{th} parameter to $W \cdot (Timeout - R_{miss})/R_{miss}$, if the queue length is below min_{th}, RED never drops a query as there is a good chance it has time for all the queries. When the queue size is between min_{th} and max_{th}, RED randomly drops queries according to the w_q and max_p parameters; these values were selected according to the authors' recommendations.

2) CoDel [12]: In CoDel, we need to select *target* and *interval* parameters. For the interval we use the authors' recommendation, and for the target, we used $Timeout - R_{miss}$, which assures that CoDel has time to resolve a non-cached



(a) Campus DNS network traffic (b) Case Connection Zone (c) SURFNet

Fig. 3: Response rate broken to recursive (R) and cached (C) when varying the cache policy and cache size. Simulation properties: fixed rate of 1000Qps, 2M DNS transactions, 15% NXDomain traffic and QART.



(a) Campus DNS traffic (b) Case Connection Zone (c) SURFNet

Fig. 4: Response rate broken to recursive (R) and cached (C) for different NXDomain traffic volumes when varying the cache policy. Simulation properties: 2M DNS transactions, fixed rate of 1000Qps and QART.

query. Thus, timeouts are still possible, but since they are very rare, there is no need for a more conservative configuration as it forces CoDel to drop more queries and hurt its performance.

3) HeadDrop / TailDrop: We set the queue size so the chance of timeouts will be negligible and therefore the queue size is $W \cdot (Timeout - R_{miss})/R_{miss}$.

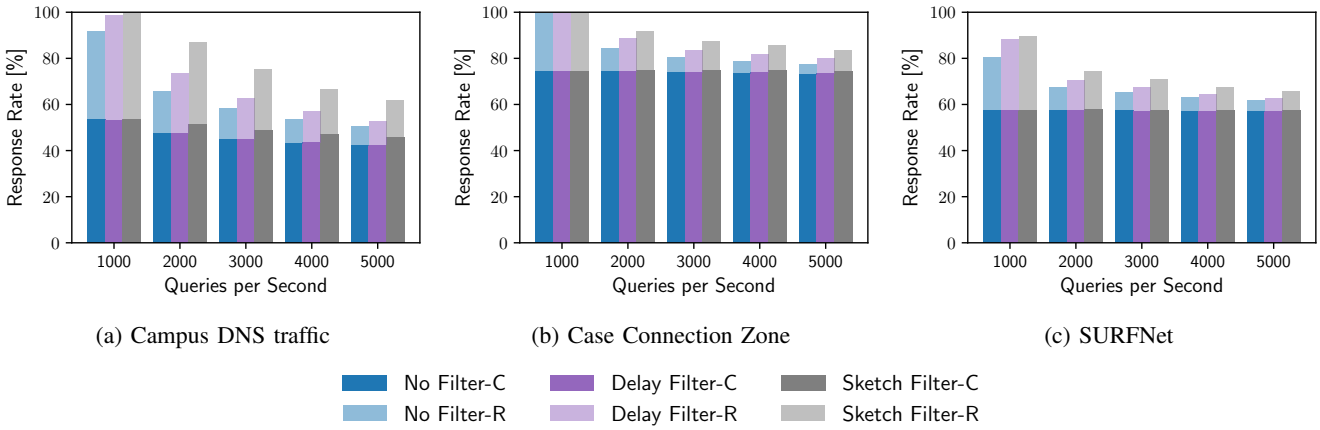
A. Queue Management

This section compares QART to CoDel, RED, HeadDrop and TailDrop under identical settings. That is standard LRU caches of the same size and the same workload.

In the first experiment, we vary the queries per second and show the effect on the Response Rate, which is the percentage of (legitimate) queries being answered. Our results are in Figure 1. As can be observed, in the Campus and Case Connection Zone traces, all queue managements can answer over 90% of the queries at 1,000 queries per second. Thus, the DNS is at the edge of its capacity. In the SURFNet trace, we can see that the servers are already struggling at 1,000 queries per second. As we increase the number of queries per second, we observe that all the schemes drop in the response rate. Yet, HeadDrop and QART behave better than the alternatives, and when the queries per second are above 1,500, QART is strictly better than HeadDrop.

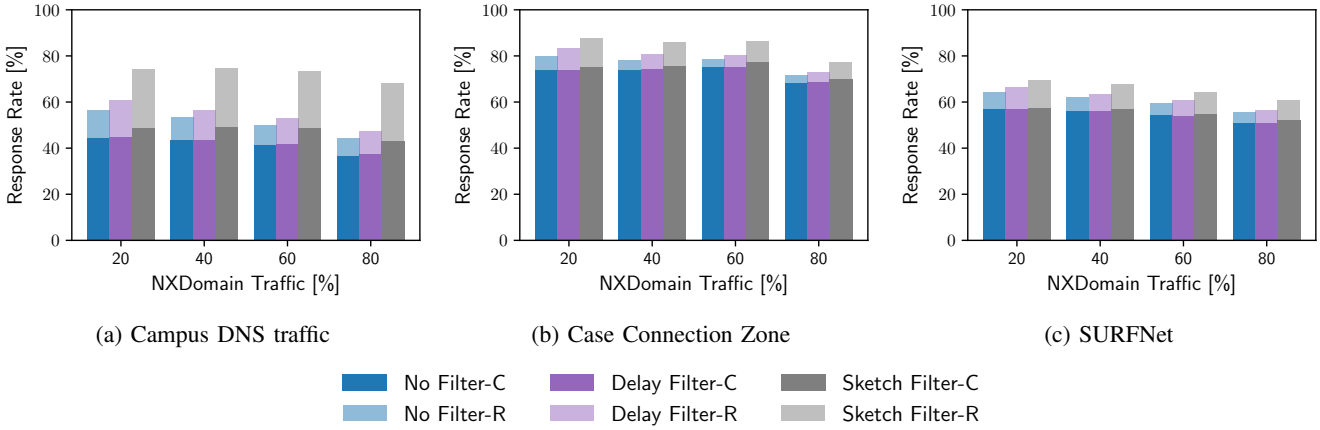
Interestingly, QART maintains a queue delay that allows performing at least a cache lookup. TailDrop and RED reach a certain queue length and drop new queries without even enqueuing them. CoDel preserves a queue delay, but its queue delay is overly conservative because it is not aware of the cache. HeadDrop drops the oldest query when the queue is full, thus avoiding timeouts. However, when the rate is higher HeadDrop fails in all its recursive lookups (e.g., see SURFNet at 2500 queries per second). The reason for such failure is that HeadDrop drops the old queries before they finish. QART, on the other hand, would only stop a recursive lookup in case of a timeout. Instead of dropping recursive lookups, QART would avoid starting them in the first place under certain conditions. Thus, it retains the highest response rate under heavy loads.

In Figure 2, we present the effect of NXDomain traffic on the (legitimate) response rate. We set a fixed rate of 1000Qps



(a) Campus DNS traffic (b) Case Connection Zone (c) SURFNet

Fig. 5: Response rate for different query rates of resolvers with different filter approaches. Simulation properties: 500K DNS transactions, 15% NXDomain traffic, Hyperbolic caching and QART.



(a) Campus DNS traffic (b) Case Connection Zone (c) SURFNet

Fig. 6: Response rate for different NXDomain traffic volumes of resolvers with different filter approaches. Simulation properties: 500K DNS transactions, fixed rate of 3000Qps, Hyperbolic caching and QART.

since we noticed in Figure 1 that the different schemes are stressed at that rate. Here, QART is better than the other queues for all workloads, starting at 40% of NXDomain traffic. As the percentage of NXDomain traffic increases, fewer queries can be satisfied from the cache, and the queue delay and length increase. Consequently, RED, TailDrop, and CoDel experience the same problem when increasing the query intensity, but this time it happens at a lower load since NXDomain traffic is not cacheable. HeadDrop handles fewer queries by cache lookups, so its benefit over other schemes decreases as the portion of NXDomain queries. The difference between it and the other queues is smaller compared the experiment with legitimate DNS traffic.

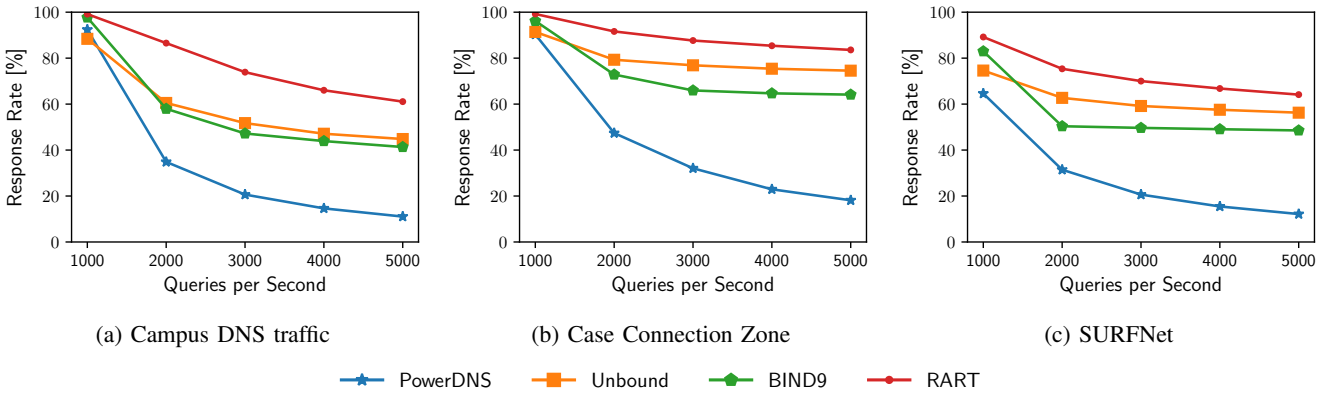
B. Cache Policy Optimization

Notice that from now on, all our evaluations use QART as queue management. This section shows the effect of modern cache policies on the effectiveness of DNS resolution. We consider five algorithms: LRU, LFU [25], Hyperbolic Caching with default score function (HYPER), and with the LFU score function(HyperLFU) [17], and the TLRU algorithm containing LRU eviction + TinyLFU admission (TinyLFU) [16].

Figure 3 demonstrates the effect of varying the cache size on the performance of the DNS. First, observe that we see an increase in all policies with the percentage of cached responses as the cache size increases. Such an increase is translated directly to the percentage of NOERROR responses as our experiment uses 1000 queries per second, which places the DNS close to its capacity.

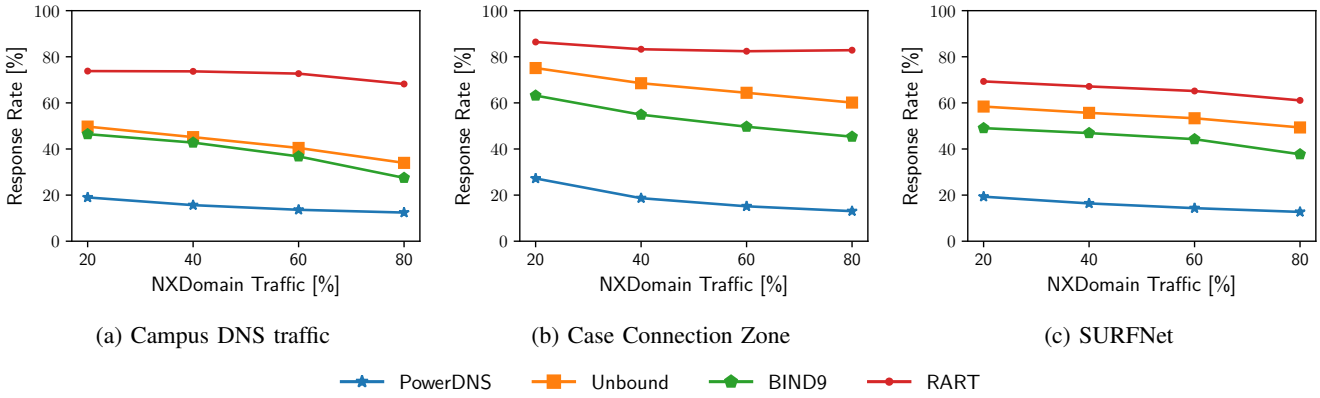
It is evident that LRU is a reasonable policy and the differences with the best policy are small. However, Hyperbolic is slightly better than the alternatives, resulting in at most a 7% increase in cached responses, which directly translates to a similar rise in successfully handled queries.

Figure 4 shows how different caches behave under an NXDomain attack. Here, we set the cache size to 1000 and vary the attack traffic percentage. First, observe that LRU behaves poorly under high NXDomain load. Such a result explains the variety of works on disposable domains which indirectly targets a weakness in LRU. Hyperbolic and TinyLFU behave



(a) Campus DNS traffic (b) Case Connection Zone (c) SURFNet

Fig. 7: Response rate for different query rates of different DNS resolvers. Simulation properties: 1M DNS transactions and 15% NXDomain traffic.



(a) Campus DNS traffic (b) Case Connection Zone (c) SURFNet

Fig. 8: Response rate for different NXDomain traffic volumes of different DNS resolvers. Simulation properties: 1M DNS transactions and fixed rate of 3000Qps.

much better under high NXDomain load, and Hyperbolic seems to be the best policy overall.

To sum up, from now on, we always use the Hyperbolic (with the original scoring metric) policy as it is competitive with the best approach in all cases and is slightly better than the best alternative under an NXDomain attack in the Campus DNS network (a margin of 2%-7% from the best alternative).

C. High Load Mitigation

Finally, we now evaluate our high load mitigation mechanisms. We compare having no mechanism (No filter) to our Delay and Sketch filter. Figure 5 shows result for increasing (legitimate) load. As can be observed, the Sketch filter is better than the alternatives across the board, even when there is no attack. Specifically, observe that a slightly higher portion of the queries is handled by the cache when using the Sketch filter. This increase is expected as the Sketch filter prioritizes completing popular queries that may benefit the cache. Figure 6 evaluates the mechanisms under an NXDomain attack. Here, the Sketch filter is dramatically better than the alternatives. Intuitively, all the attack packets are unpopular, and the Sketch filter ignores most of them when the load is too high. Thus, we use the Sketch filter in all subsequent measurements.

D. RART and Other DNS Resolvers

This section compares RART, which uses QART for queue management, hyperbolic caching, and the Sketch filter with popular open source resolvers. Details about these resolvers appear in Table I. Figure 7 presents the response rate of the resolvers under loads of legitimate DNS traffic. PowerDNS does not try to perform a cache lookup and suffers from the lowest response rate. Unbound has an LRU cache policy, while RART has Hyperbolic caching, which results in a higher cache hit ratio. BIND cannot perform a recursive lookup starting from a specific rate due to its queue management, while our resolver does not drop queries in handling. RART has a better response rate than BIND, Unbound, and PowerDNS by 17.8%, 13.2%, and 44.7% on average under legitimate DNS traffic.

In Figure 8 we evaluate the resolvers under varying NXDomain traffic rate. RART resolver is less vulnerable to NXDomain attack since it has the Sketch filter protection layer. The other resolvers' queues fill up quickly, directly translating to an increase in drop rate. Consequentially, our resolver has a greater response rate than BIND, Unbound, and PowerDNS by 28.4%, 19.2%, and 57.3%, respectively.

V. Discussion

The capacity of DNS resolvers to handle high loads and volumetric attacks is instrumental to the stability of our Internet infrastructure. We focused on three crucial design decisions and created the RART resolver by carefully optimizing each of these decisions. We showed through an extensive evaluation that RART processes on average 21% more queries for the same load, 41% more queries in an NXDomain attack.

RART performs risk management based on the remaining time to complete a query. In a gist, it only performs a resolution step (such as checking the cache or performing a recursive resolution) if we have enough time to do so. Our approach uses multiple thresholds, which we measure by measuring the current performance. Thus, the thresholds would automatically update after scale-up operations.

In addition, RART includes an adaptation of the state-of-the-art Hyperbolic cache policy into DNS resolvers and shows that the new policy behaves better under attacks when compared with the standard LRU policy (that currently appears in all open source DNS projects). We show that the performance of the cache policy is crucial in handling high loads and attacks, and thus we also suggest using the novel Sketch filter approach to maximize the cache hit rate under heavy loads.

Intuitively, when RART cannot handle all the queries, it prioritizes queries to popular domains. These allow the cache to function better under heavy load, and as a side effect, attack packets (e.g., for an NXDomain attack) tend to be unpopular and are more likely to be dropped. However, such an approach may lead to fairness issues under high legitimate loads where the high goodput might hide lower availability for unpopular domains. However, the alternative is to fail on more queries.

Looking into the future, we plan to pursue the adaptation of our approach to existing DNS resolvers. Such a deployment may require better access to managing the query queue. However, the existing resolvers seem to move in a similar direction. For example, BIND 9.18 already supports editing the queue size of the operating system [26], which removes an obstacle to realizing our work on BIND.

For reproducibility, we plan to release our code as an open source once anonymity restrictions are lifted. In the meantime, our code can be found in this anonymous repository [27].

References

[1] L. Wei-min, C. Lu-ying, and L. Zhen-ming, "Alleviating the impact of dns ddos attacks," in *2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing*, vol. 1, April 2010, pp. 240–243.

[2] S. L. Feibish, Y. Afek, A. Bremler-Barr, E. Cohen, and M. Shagam, "Mitigating dns random subdomain ddos attacks by distinct heavy hitters sketches," in *Proceedings of the Fifth ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, ser. HotWeb '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3132465.3132474

[3] H. Ballani and P. Francis, "Mitigating dns dos attacks," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 189–198. [Online]. Available: https://doi.org/10.1145/1455770.1455796

[4] V. Pappas, D. Massey, and L. Zhang, "Enhancing dns resilience against denial of service attacks," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, 2007, pp. 450–459.

[5] N. Kostopoulos, A. Pavlidis, M. Dimolianis, D. Kalogeras, and V. Maglaris, "A privacy-preserving schema for the detection and collaborative mitigation of dns water torture attacks in cloud infrastructures," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, 2019, pp. 1–6.

[6] K. Hasegawa, D. Kondo, and H. Tode, "Fqdn-based whitelist filter on a dns cache server against the dns water torture attack," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 628–632.

[7] "Unbound by nlnet labs." [Online]. Available: https://unbound.docs.nlnetlabs.nl/en/latest/

[8] "Rules for traffic exceeding qps limits." [Online]. Available: https://doc.powerdns.com/recursor/indexTOC.html

[9] "Recursive client rate limiting." [Online]. Available: https://kb.isc.org/docs/aa-01304

[10] K. Schomp, O. Bhardwaj, E. Kurdoglu, M. Muhaimen, and 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*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 465–478. [Online]. Available: https://doi.org/10.1145/3387514.3405881

[11] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, p. 397–413, aug 1993. [Online]. Available: https://doi.org/10.1109/90.251892

[12] K. M. Nichols, V. Jacobson, A. McGregor, and J. R. Iyengar, "Controlled delay active queue management," *RFC*, vol. 8289, pp. 1–25, 2018.

[13] S. Hao and H. Wang, "Exploring domain name based features on the effectiveness of dns caching," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 1, p. 36–42, jan 2017. [Online]. Available: https://doi.org/10.1145/3041027.3041032

[14] G. Yu, Y. Zhang, H. Cui, X. Yang, and Y. Li, "Mitigating negative impacts on dns caches caused by disposable domain names," in *2019 IEEE Symposium on Computers and Communications (ISCC)*, 2019, pp. 1–6.

[15] X. Yuchi, X. Lee, and L. Pan, "Dealing with temporary domain name issues in the dns," in *2016 IEEE Symposium on Computers and Communication (ISCC)*, 2016, pp. 778–783.

[16] G. Einziger, R. Friedman, and B. Manes, "Tinylfu: A highly efficient cache admission policy," *ACM Transactions on Storage (TOS)*, 2017.

[17] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017, pp. 499–511.

[18] D. C. Lawrence, W. A. Kumari, and P. Sood, "Serving Stale Data to Improve DNS Resiliency," RFC 8767, Mar. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8767

[19] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[20] (2018) DNS Queries to Authoritative DNS Server at SURFnet by Google's Public DNS Resolver. [Online]. Available: https://doi.org/10.4121/uuid:1ef815ea-cb39-4b41-8db6-c1008af6d5aa

[21] (2016) 10 Days DNS Traffic. [Online]. Available: https://data.mendeley.com/datasets/zh3wnddzxy/2

[22] (2021) Case Connection Zone. [Online]. Available: https://www.icir.org/mallman/data.html

[23] O. Himelbrand. [Online]. Available: https://github.com/himelbrand/dns-timing

[24] R. Taylor, "The top four dns response codes and what they mean." [Online]. Available: https://bluecatnetworks.com/blog/the-top-four-dns-response-codes-and-what-they-mean/

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[26] "Bind 9.18 configuration reference." [Online]. Available: https://bind9.readthedocs.io/en/v9_18_0/reference.html

[27] [Online]. Available: https://anonymous.4open.science/r/DNSSimulation-BDDC