Adaptive Deterrence of DNS Cache Poisoning

Sze Yiu Chau[1], Omar Chowdhury[2], Victor Gonsalves[1], Huangyi Ge[1], Weining Yang[3], Sonia Fahmy[1], and Ninghui Li[1]

[1] {schau,vgonsalv,geh,fahmy,ninghui}@cs.purdue.edu, Purdue University
[2] omar-chowdhury@uiowa.edu, The University of Iowa
[3] weiningy@google.com, Google Inc.

Abstract. Many long-lived network protocols were not designed with adversarial environments in mind; security is often an afterthought. Developing security mechanisms for protecting such systems is often very challenging as they are required to maintain compatibility with existing implementations, minimize deployment cost and performance overhead. The Domain Name System (DNS) is one such noteworthy example; the lack of source authentication has made DNS susceptible to cache poisoning. Existing countermeasures often suffer from at least one of the following limitations: insufficient protection; modest deployment; complex configuration; dependent on domain owners' participation. We propose CGuard which is an adaptive defense framework for caching DNS resolvers: CGuard actively tries to detect cache poisoning attempts and protect the cache entries under attack by only updating them through available high confidence channels. CGuard's effective defense is immediately deployable by the caching resolvers without having to rely on domain owners' assistance and is compatible with existing and future solutions. We have empirically demonstrated the efficacy of CGuard. We envision that by taking away the attacker's incentive to launch DNS cache poisoning attacks, CGuard essentially turns the existence of high confidence channels into a deterrence. Deterrence-based defense mechanisms can be applicable to other systems beyond DNS.

1 Introduction

At the inception of network protocol design and system development, designers were oftentimes more focused on attaining scalability, instead of robustness in adversarial environments. Security mechanisms were thus only introduced retrospectively after suffering damaging attacks. This requires security mechanisms to be compatible with existing installations, manage overhead and deployment cost, and remain incentive compatible at the same time. Such design restrictions induce security mechanisms that are often ineffective in many corner cases or require major infrastructural overhaul that risks widespread adoption. One pragmatic approach to remedy this often hopeless situation, is to aim for deterrence. The key idea behind practical deterrence-based defense mechanisms is to ensure that the attacker has to invest a substantial amount of resources to carry out a successful attack, hence removing the incentives for attackers to launch attacks.

Such a principle is reminiscent of the classic deterrence theory [59]. In this paper, we *apply the principle of deterrence-based defense for the case of DNS*.

DNS is a critical part of the core Internet infrastructure. From the outset, DNS lacked a robust mechanism to authenticate DNS responses which enabled attackers to poison a caching resolver's cache of DNS entries by response spoofing—violating the integrity guarantees expected from DNS caches. Despite years of patching, DNS cache poisoning attacks still plague the DNS infrastructure [5, 6, 30, 33]. As shown by recent reports, successful cache poisoning can further enable a variety of other attacks; *e.g.*, mail handling hijacks [53, 58], drive-by downloads [13], and phishing [20, 48, 50].

The revelation of the *Kaminsky attack* in 2008 [35] was a wake-up call for the DNS community. Many software vendors started to implement source port randomization [15]—the effectiveness of which has been shown to be limited, particularly if the resolver is behind a Port Address Translator (PAT) that uses a deterministic port allocation scheme [2, 28, 34]. Efforts have also been made in further increasing the entropy of DNS packets [22, 44]. This line of defense, however, faces a dichotomy of challenges: each proposal has its own corner cases that limit robustness; and using such mechanisms while remaining compatible with entities that do not support them requires significant management effort [7].

An alternative is to run the DNS protocol on top of TCP [RFC5966] instead of the connectionless UDP. TCP provides better DNS response authentication than UDP. However, as reported in previous studies [10,19,31,32,60,62] and also observed in our own experiments, DNS over TCP, if not deployed with carefully chosen optimizations (recommended but not mandated by [RFC7766]), incur a noticeable overhead and negatively impact overall DNS performance.

Another line of cache poisoning defenses (*e.g.*, DNSSEC, DNSCurve), employs cryptographic primitives to provide authenticity guarantees to DNS response. DNSSEC in particular has been considered to be the future of DNS. These solutions, however, have not seen prevalent adoption. The deployment of DNSSEC is currently very limited [54,57], and ICANN will not deploy DNSCurve in the root zone due to key distribution and management issues [17].

The central research question we seek to answer in this paper, is *whether it is possible to design a robust defense mechanism for resolvers—without cooperation from the domain owners—that is applicable irrespective of the deployment rate of new defenses (e.g., DNSSEC)?* We focus our discussion on recursive resolvers, as they are higher-valued attack targets than stub resolvers (*i.e.*, resolvers running on a client machine) due to impact on more victims, and we argue that operators of recursive resolvers have an incentive in deploying reliable DNS services for their customers. We particularly focus on racing cache poisoning attacks carried out by off-path/blind attackers.

To this end, we propose an adaptive defense framework against DNS cache poisoning that we refer to as CGuard. In short, CGuard actively tries to detect attack attempts on cache entries and switches to a higher confidence channel for cache updates. Though mechanisms that switch to TCP during spoofing attacks have been described before [29, 41], developing a robust but flexible adaptive defense involves subtle design decisions that, as we show through a case study, if not chosen carefully, can make the resolver vulnerable to an adaptation of Kaminsky attack.

CGuard provides strong guarantees and is readily deployable by operators of recursive resolvers. As a flexible framework, CGuard can be instantiated by configuring its detection sensitivity and providing a list of usable channels, ordered in preference. Since the various high-confidence channels are used only when CGuard detects an attack, it greatly limits any attacker's success probability while maintaining a good overall performance. This also allows the various proposed high confidence channels to potentially cover for each other in terms of both corner cases and availability.

We envision that by ensuring attacks have a low probability of success, the incentives for rational attackers to launch poisoning attacks could be removed, effectively turning CGuard into a deterrence, without having to always pay for the high overhead associated with the various high confidence channels.

Contributions. In summary, this paper makes the following two contributions. **First**, we show how previously proposed cache poisoning defenses, though well-designed, fall short in practice due to different reasons. **Second**, based on the lesson learned from an adaptive defense case study, we design the CGuard adaptive deterrence framework against racing cache poisoning attacks, and empirically evaluate its effectiveness based on a particular instantiation of CGuard that we implemented.

2 Background

We now give a brief primer on DNS, and establish some of the terminology and notations that are used throughout the rest of the paper. For a detailed taxonomy of DNS cache poisoning attacks, we refer the readers to [52].

DNS queries from users are typically sent to an upstream *recursive resolver*, which will fully answer the query (or give an error) by traversing the DNS domain tree and querying other name servers. When a valid response is received, it is used to answer the query and cached for future queries. DNS queries and responses typically go over UDP, though the standard also supports message exchange over TCP. A response over UDP is considered valid if the query information, including the transaction ID (TXID), query name, and query type, matches that of the query. As such matching heuristic is not strongly authenticated, this presents an opportunity for cache poisoning attacks [49].

Depending on their capabilities, cache poisoning attackers can be classified as *in-path*, *on-path*, and *off-path*. On-path attackers have the ability to observe DNS query packets, and therefore can easily create forged response packets that will be accepted. In-path attackers have the additional capability to delay and drop packets. These are usually powerful nation-state adversaries, often used in implementing censorship [23, 40]. For DNS resolvers that operate outside the jurisdiction of such censors, however, connection controlling in-path and on-path attackers are much less likely. One is mostly concerned about *off-path* attackers who cannot observe but can query resolvers with domain names of their choosing. Protecting DNS resolvers against such off-path attackers is extremely important, as the number of parties who can potentially carry out off-path attacks could be very large. In addition, once a cache entry is poisoned, it can affect other clients that are configured to use the same resolver.

3 Assessing Proposed Defenses

We now discuss previously proposed defenses against cache poisoning, with a focus on their deployment challenges, availability, and corner cases.

3.1 Increasing Entropy

One school of thought on hardening DNS is to introduce more entropy on top of the 16-bit entropy provided by TXID.

Source port randomization. One possibility is to use random source ports for DNS UDP queries [2, 15]. This defense is adopted by several major DNS implementations. Ideally, close to 16 bits of entropy would be added. However, network middleboxes (*e.g.*, the likes of firewalls, proxies and routers) that perform Port Address Translation (PAT), depending on their configurations, might reduce the randomness of UDP source ports used by resolvers behind them [2,34], and such resolver-behind-NAT scenario is reported to be quite common [26, 28]. It has also been shown that if a DNS server and an attacker-controlled machine are behind the same NAT, then the attacker can force the NAT to make highly predictable choices of UDP ports, possibly removing any extra entropy [28].

0x20 Encoding. This mechanism rewrites the domain name in a DNS query by randomly using upper/lowercase letters [22]. If a domain name contains k alphabetic characters, the entropy gain is k bits. The method is less effective for domain names with few letters. To poison the entries for name servers of `.com`, attackers can send queries with domain names such as `853211.com` in Kaminsky attacks [28]. Another deployment hurdle is that some name servers always respond with names in lowercase [7]. Some others, in violation of the DNS standards [RFC4343], try to match the exact case of the name in the query, hence fail to resolve. Google Public DNS's solution is to create a whitelist of name servers which is compatible with 0x20 encoding. Name servers in the whitelist constitute about 70% of all traffic [7].

WSEC DNS. Another proposal is to prepend a random nonce label to query QNAME [44]. This is possible because, in most cases, requests to the root or top-level domain (TLD) name servers will result in a referral to a name server lower in the hierarchy, instead of an actual answer with IP addresses. For example, `asdf.www.msn.com` should yield the same resource record (RR) as `www.msn.com` when querying the root or `.com` name servers. It has been argued that WSEC DNS is ineffective against Kaminsky attacks [28]. This is because the total number of characters in a domain name cannot exceed 255, thus attackers can query near 255-byte-long domain names to circumvent the mechanism. Furthermore, this defense applies only to requests where referrals are expected. The Google DNS team faced challenges in deciding when is such defense applicable [7].

Randomizing destination and source IP addresses. Destination IP address can be randomized if there exists a pool of possible server addresses, and source IP address can be randomized at an NAT that can inspect and rewrite IP addresses. The actual entropy gain of these two proposals, however, are logarithmic to the number of servers and size of a network, hence often quite limited.

Summary. Proposals on increasing entropy are generally opportunistic, and there exist corner cases that would yield limited gains. Some mechanisms like WSEC DNS and 0x20 encoding require significant manual effort on tracking incompatible servers. Consequently, when we develop our adaptive approach, we do not use these mechanisms.

3.2 DNSSEC

DNSSEC (Domain Name System Security Extensions) digitally signs DNS RRs using public-key cryptography [RFC4033–4035]. Although DNSSEC was proposed back in 1997 [RFC2065] its adoption has been slow. The number of DNSSEC validating clients is growing, albeit slowly [4, 12, 38]. Meanwhile, the adoption rate on the domain side remains low. It has been shown that only around 1% of all the `.com` and `.net` domains are secured by DNSSEC [27,54,57]. The measurement of our experiment below shows similar findings. To enjoy the assurances of DNSSEC, domain owners are often required to take the initiative in configuring it. Misconfiguration can be used by DDoS reflection attacks [18,47], and can lead to loss of users [38]. A recent study showed that many DNSSEC-signed domains are also plagued by poor key generation practices [51]. There are no real technical reasons why DNSSEC should not be used, though cost and management issues exist that are deterring adoption.

DNSSEC support. To test whether DNSSEC is deployed, for each authoritative name server address[4] we request the `DNSKEY` type record of the domains for which it is authoritative. If a domain has DNSSEC correctly deployed, the authoritative name servers should return a response with `DNSKEY` type and a `RRSIG` type RR. We then consider the authoritative name server as supporting DNSSEC if the signature validates. A domain is considered to support DNSSEC if all its authoritative name servers support DNSSEC. We observe that *among the top* $15,000$ *domains, only* **1.1**% *have DNSSEC support.*

Summary: DNSSEC availability is currently very limited but we will use it in our adaptive defense mechanism whenever applicable, as it has been standardized and the Internet community has been promoting its adoption [8, 9, 39].

[4] We obtained 18,075 unique IP addresses from 19,669 authoritative name servers of the top $20,000$ domains as ranked by Alexa. Many of our subsequent experiments are also based on this data.

3.3 DNS over TCP

Although TCP support is mandated by the standard, it is typically only used by resolvers as a fall-back mechanism when packets are long, or if a TCP connection has already been established and is open [RFC5966]. DNS over TCP enjoys both reliable transport and extra entropy. Specifically, the combined entropy from TXID and TCP sequence number is high enough to make off-path attack unappealing. We would like to quantify the overhead if all the resolutions are done over TCP.

TCP support. For each authoritative name server address, using TCP as the transport protocol, we ask for the `A` records of the domains it is authoritative for. If a valid response is returned, then this authoritative name server address is considered to support TCP. If not, we send the same query again but through UDP, to verify that the server is responsive. In the end, 636 addresses did not respond to any TCP or UDP queries. *Out of the* 17,439 *authoritative name server addresses that responded,* 15,774 (90.4%) *support TCP. About* **85**% *of the top* 15,000 *domains have TCP support on all of their authoritative name servers.*

TCP overhead in recursive resolvers. We empirically determine the overhead a recursive resolver incurs due to resolving queries iteratively through TCP. We extract domains whose authoritative name servers support TCP, and query for `A` records using the `drill` utility. For each domain, after measuring the latency with UDP, we clear the cache, reset the resolver software, and then measure the latency with TCP. This guarantees that the recursive resolver will perform iterative queries from the root for each measurement instance. In the end, the average time for UDP was 423 ms/domain and TCP was 834 ms/domain, over 17,340 domains. *On average, the total communication overhead for TCP is roughly twice of UDP, as shown in Figure 1.* This result is consistent with the number reported in a recent work [62] (Fig. 7(b), with full TCP handshake and no connection reuse), which is unsurprising as each such DNS over TCP instance needs two round-trip times whereas UDP needs only one [62].

We note that various optimizations like connection reuse, pipelining and out-of-order processing that can improve the performance of DNS over TCP are also discussed in [62]. For the latter two, as noted in [62], major software have no/partial support, so we do not consider them here. For connection reuse, its effect depends on actual traffic pattern and server configurations. Also note that in [RFC7766], connection reuse and pipelining are recommended but not mandated for clients. Our experiments here can be thought of as stressing servers at a worst-case scenario.

TCP overhead in authoritative name servers. We attempt to find empirically, from the point of view of an authoritative name server, how much overhead it will incur if all the resolvers use TCP for queries, without connection reuse.

We did an emulation study using a machine with Intel Core i5 2.5 GHz CPU and 8 GB RAM, running Unbound 1.5.4 configured as an authoritative name server (of a local zone). Client is another machine with Intel Core i7 2.2 GHz CPU and 16 GB RAM running a modified version of queryperf++ [56].

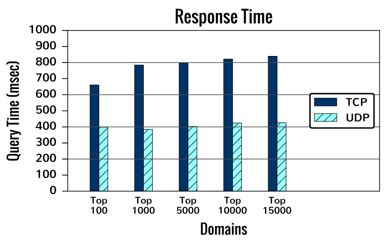

Fig. 1. Resolution latency between recursive and authoritative servers

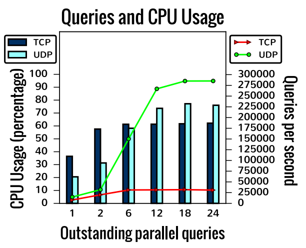

Fig. 2. Service rate and CPU usage of an authoritative server

For both TCP and UDP, we vary the query window size (*i.e.*, number of parallel outstanding queries on the client side) to produce different volumes of traffic. For each window size, the average CPU usage of the authoritative name server for {TCP, UDP} is obtained by taking the mean of 600 data points measured over 1,200 seconds of queries. The results can be found in Figure 2. As we allow more outstanding parallel queries, the service rate (queries responded per second) also increases, up to a point where it saturates. *Comparing to TCP, UDP yields a higher peak service rate.* When the window size is 2 for UDP and 6 for TCP, the service rate becomes comparable, and UDP exhibits a much lower (roughly half) CPU usage than that of TCP.

Summary: TCP support is widely available, though without the recommended optimizations, it negatively impacts overall DNS performance, which motivates the benefits of having an adaptive deterrence that uses the TCP channel. Given its standardized status and good general availability, we will use TCP in our adaptive defense mechanism whenever applicable.

3.4 Other Defenses

Other proposals including CoDNS [43], DoX [61], ConfiDNS [45] and Anax [11] require significant resources and changes to the DNS infrastructure in order to be deployed at DNS resolvers, making their adoption unlikely.

Another cryptographic mechanism is DNSCurve [14]. Though promising, it has not received wide deployment on authoritative name servers. Overhead is one of its discussed drawbacks, particularly because embedding cryptographic keys in names would render existing resolver caching mechanisms ineffective [36], increasing query traffic at authoritative name servers. Moreover, due to key distribution and management issues in potentially hostile regions, it has been said that ICANN will not deploy DNSCurve in the root zone [17].

4 Adaptive Defense: Challenges

We now discuss the challenges involved in making the adaptive defense paradigm (i.e., attack detection and protection) effective against Kaminsky-style racing cache poisoning attacks. Though adaptively switching to TCP during spoofing attack has been mentioned as a possible countermeasure before [29, 41], to make such a defense robust and widely applicable is actually non-trivial. The intricacies are hidden in (1) the decision logic of when to use which channel, (2) the granularity of detection and protection, and (3) the availability of high confidence channels and a reasonable preference. Using a real world instantiation from Nominum [41] as a case study, we will explain the associated challenges and show how a simplistic implementation falls short in effectiveness. Our design decisions will be discussed in Section 5.

4.1 Preliminary

Threat Model: We consider an off-path or a blind attacker targeting a DNS caching resolver. The attacker can neither observe any outgoing query posed by the resolver nor has the capability to stop it. We also consider the attacker to have the knowledge of the deployed mechanism enabling him to adapt his attack strategy. We consider that the `TXID`s are randomly generated and are unpredictable to the attacker. Finally, we assume that the attacker may control many edge hosts (e.g., a botnet) and have coordination capabilities resulting in a substantial amount of attack bandwidth.
Mismatched DNS Response and Attack: If a DNS resolver sends a DNS query q and receives a response r such that r agrees with q on all the deterministic fields but disagrees on one or more of the randomized fields (e.g., `TXID`, source port), then we call the response r a **mismatched DNS response**.

At a first glance, it seems very enticing to consider receiving any mismatched DNS response to be a sign of active cache poisoning attack; we want to point out that a mismatched response can also occur in a benign setting. Suppose a resolver does not get a response to a pending DNS query within a threshold amount of time, and it sends out a new query with a new random `TXID`. While waiting for the response to the new query, the old response arrives, and the old response's `TXID` does not match the current pending query's `TXID`. This is plausible as UDP does not provide reliability guarantees.
Kaminsky Attack: Suppose an off-path attacker Adv intends to poison caching resolver R's DNS cache entry for `google.com`'s authoritative name server (e.g., `ns1.google.com`). Adv starts off by querying R with an inevitably non-existant domain of the form `dshmik.google.com` which is unlikely to be stored in R's cache, resulting in a cache miss. Without loss of generality, suppose the IP address of `google.com`'s name server `ns1.google.com` (e.g., 1.2.3.4) is stored in R's cache and random `TXID` is the only form of available entropy. In which case, R will query `ns1.google.com` to resolve the query.

Adv then races the legitimate name server's response by flooding R with fake DNS responses with spoofed source IP address 1.2.3.4 where it tries different values of `TXID`. Adv generated fake DNS responses contain a glue record which provides the IP address of `ns1.google.com` to be one of Adv's choice. If one of the fake responses submitted by Adv contains the correct `TXID` and it arrives before the legitimate response, R will cache the malicious glue record; poisoning it. Once poisoned, any future resolution of any subdomain of `google.com` (e.g., `docs.google.com`), Adv's name server is going to be consulted. If Adv cannot

guess the correct `TXID`, he can restart the same attack with a different non-existant domain (e.g., `rksjtw.google.com`). This attack has the following two desirable characteristics over regular cache poisoning: (1) The attacker can start the attack without having to wait for a cache entry to expire; (2) The attacker does not have to pay any penalty (i.e., no waiting), even if he cannot submit a fake response with correct `TXID` before the legitimate response arrives.
Nominum's Adaptive Defense: In Nominum's approach implemented in their proprietary Vantio CacheServe product, whenever a resolver receives a mismatched DNS response for an outstanding DNS query, the resolver considers the query to be under attack and immediately switches to TCP for responding that pending query. For spoofing the DNS response, the attacker additionally has to guess the TCP sequence number increasing the overall entropy to around 48 bits (i.e., 32 bit TCP sequence number + 16 bits for `TXID`).

4.2 Attack Detection: Challenges

We now discuss the challenges a resolver has to overcome for effectively detecting an active cache poisoning attempt.
(1) Attack Detection Settings: A resolver can possibly attempt to detect an attack in a **local setting** or in a **global setting**. In the local setting, the resolver checks to see whether a particular unit is under attack. In case of a detected attack, only that unit is protected. In this setting, different protection units are considered independent. *Nominum's attack detection operates in the local setting.* Depending on the attack detection granularity (discussed next), it may be possible to bypass the detection mechanism in the local setting by slightly modifying the original Kaminsky attack.

In the global setting, however, when the resolver detects an attempted attack, it proactively starts protecting all the cache entries. Under the assumption that cache entry protection incurs some amount of additional overhead (e.g., network bandwidth, memory), even if only one query (or cache entry) is under attack, other queries will suffer, thus it is undesirable to always use this setting.
(2) Granularity of Detection and Protection: A fundamental challenge in successfully detecting cache poisoning attacks is to be able to correlate different attack instances. In the above example of Kaminsky attack, both `dshmik.google.com` and `rksjtw.google.com` queries are part of the attacker's goal of poisoning the name server (e.g., `ns1.google.com`) of `google.com`. This induces the critical design decision of the granularity of attack detection. The three alternatives are: (i) **per-query**, (ii) **per-domain**, and (iii) both **per-query** and **per-domain**.

In the per-query attack detection, whenever the resolver receives one or more mismatched responses for a pending DNS query, the query (answered in the Answer Section of the responses) is considered to be under attack. When a resolver, however, is detecting attack at per-domain granularity, it considers its zone to be under attack if the domain (and its subdomain) was in the records of the Additional Section of one or more mismatched DNS responses. Detecting attacks at both the per-query and per-domain granularity offers better protection.

Nominum's implementation detects attack at per-query granularity and it is susceptible to the following attack.

Bypassing Per-Query Attack Detection: Detecting attacks at per-query level, with mismatch threshold of one, only limits an off-path, Kaminsky-style attacker to send one mismatched response for each attack query. After the first mismatched response, the resolver will start protecting the query (e.g., using TCP to resolve the query as in Nominum's implementation) thwarting all subsequent forged responses by the attacker. **However, it does not stop the attacker to start a new attack instance right away**. This is due to the fact that the resolver cannot correlate two attack instances `dshmik.google.com` and `rksjtw.google.com`, by only keeping track of queries. One can thus modify the original Kaminsky attack, which we dubb as **Parallel Kaminsky Attack (PKA)**, to send only one forged response (instead of many forged responses) for each non-existant DNS query.

Each PKA instance has a $\frac{1}{2^r}$ success probability when r-bit entropy is available to the resolver. For the traditional Kaminsky attack, on the contrary, if the attacker can send n forged responses before the legitimate response arrives, the attacker's success probability is $\frac{n}{2^r}$. A PKA instance succeeds when the *single forged response* for the query happens to match. The attacker, however, does not need to wait until one attack has failed to carry out another attack. If the resolver has only a 16-bit entropy, either because source port randomization is not available or is severely limited, and the attacker can cause the server to send, *e.g.*, 500 queries/second, then it takes just a couple of minutes for PKA to succeed. With 30-bit entropy, it takes the attacker about 2 days to succeed. *Note that imposing rate limiting on outstanding queries of the same question (i.e., Birthday attack [42] protection) does not prevent PKA as the query for each PKA instance is different.*

(3) Storing Attack History: Irrespective of the attack detection setting and granularity, the affected domains, queries, or cache entries should be stored in some data structure so that the resolver can protect them. One of the main challenges is the resolver-side overhead and accuracy trade-off for storing attack information. If the resolver stores fully accurate attack information, the attacker can strategically make the overhead prohibitive whereas if the resolver stores attack information in a probabilistic data structure (e.g., bloom filter) its attack detection accuracy may deteriorate due to false positives. One possibility is to maintain a fixed-size cache which contains fully accurate attack information. This, however, signifies that past information may have to be evicted to accommodate new attack information, exposing an attack vector where an attacker can strategically try to remove entries from the attack history cache.

(4) Lifetime of an Attack Entry: The next natural question is how long does the resolver protect a query, a domain, or a cache entry under attack. This is similar to the time-to-live (TTL) field for DNS cache entries. Suppose the TTL of an attack entry is 2 minutes. In this case, a greedy attacker with the goal of poisoning one of the top $100,000$ domains can possibly carry out a **spray attack** on different domains. Such an attack could be carried out by posing the following queries in succession: $\langle \text{random}_1 \rangle$`.google.com`, $\langle \text{random}_2 \rangle$`.facebook.com`, $\langle \text{random}_3 \rangle$`.yahoo.com`, ... By the time attacker comes back to attacking `google.com` again, hopefully the 2 minutes are up; forcing the resolver to forget about the attack on `google.com`.

4.3 Protection: Challenges

Under the assumption that an effective attack detection mechanism is in place, the resolver has to establish a mechanism to protect a domain, a query, or a cache entry. The following are few possibilities with which a resolver can protect a domain, a query, or a cache entry.
(1) Employ a High-Confidence Channel: Although DNS was designed to run on top of UDP for scalability, some authoritative name servers expose high confidence channels such as TCP and DNSSEC. When a resolver detects an attack on a cache entry (or, a query), it can employ one of these high confidence channels to update the cache entry under attack. Along with the additional overhead imposed by these high confidence channels, one of the major obstacles of employing high confidence channels is their low adoption rate. *Nominum's implementation employ TCP as the high confidence channel.* Based on our measurements in Section 3.2 and 3.3, among Alexa's top 15, 000 domains, TCP availability is about 85% and DNSSEC availability is about 1%. This means more high-confidence channels are needed to provide protection to more domains.
(2) No Caching: When a resolver detects an attack on a cache entry, it may decide not to update the entry even if it is part of a possibly legitimate response (e.g., glue record). This, however, leads to significant performance degradation due to the need to traverse the domain hierarchy even for benign cases.
Summary: Why Nominum's adaptive defense approach is inadequate? Nominum's implementation does not achieve the full defense potential due to the following two limitations: (I) Due to their query level attack detection, they are susceptible to PKA; (II) They do not take into consideration the possibility of an authoritative name server not supporting TCP.

5 CGuard: An Effective Adaptive Defense

We now present our instantiation of the adaptive defense paradigm which we refer to as by **CGuard**. CGuard *proactively tries to detect racing cache poisoning attack attempts and switches to one of the available high confidence channels to update those cache entries that are under attack.* We detail CGuard's attack detection, the high confidence channels we currently consider, and the overall defense mechanism.

5.1 Attack Detection of CGuard

CGuard conservatively detects attack at both global and local setting. To prevent spray attacks across a large variety of different domains, if CGuard observes a total of mm_thresh (default value 10, configurable) mismatched responses over the last 10 minutes, it considers a system-wide attack and starts protecting all cache entries. In other case, CGuard detects attack at local setting using both per-domain and per-query granularities for attack detection. For storing attacks CGuard uses a set-associative cache with least-recently used (LRU) replacement policy. We use a non-cryptographic hash function to map a domain to its set in the attack cache. The life-time of each attack entry is attack_TTL with the default value being 15 minutes.

For a pending DNS query, *if CGuard receives a mismatched response, we consider the query, the additional* RR*s, and* ***their associated zones*** *to be under attack*. A DNS response can have additional RRs that the resolver did not *explicitly* ask for. These additional RRs can be viewed as prefetching of NS records (authoritative name server records) signifying that the authoritative name server of a domain has changed. For each RR in the additional section of a mismatched response, we consider that RR's domain (and if it is an NS record, then also its zone), to be under attack. This is based on the intuition that if the TXID would have matched, then the RR would have been cached, and hence it is safe to consider it under attack. Without considering the zone of the RRs to be under attack, the threat of PKA lingers as demonstrated by the following example.

Example attack scenario. Suppose an attacker Adv started a PKA instance by posing an `A`-type query for a non-existent domain `xdRfggh.google.com`. Obviously, there is no `A` record in the resolver R's cache for that domain. Suppose R has not observed any attack so far. R starts traversing the domain hierarchy and observes that it has the following authoritative name server for the zone `google.com` in its cache: `ns1.google.com`. Suppose that R also has the name servers' `A` records in its cache and sends the query to it. Adv then sends a mismatched, forged response for `xdRfggh.google.com`; trying to poison the cache entry for `ns1.google.com`. A hypothetical attack detection mechanism that does not add the name servers' zone, will add the following domains to the attach cache: `xdRfggh.google.com` and `ns1.google.com`. In the next PKA instance, Adv sends an `A` record request for `YUrrpom.google.com` to R. R will ask `ns1.google.com` for the `A` record of the domain `YUrrpom.google.com`. However, this time around Adv gets lucky and correctly guesses the TXID. In this forged response, Adv also adds a new name server for `google.com`, say `ns2.google.com`, and provides a glue record for it. The glue records for `ns1.google.com` can be ignored as it is under attack. However, the glue record for `ns2.google.com` will be cached resulting in a cache poisoning. If we monitor whether a name server's zone is under attack, in the above example, we will check whether `google.com` is under attack—which it is—before we add an NS record for `google.com` and update the `A` record for `ns2.google.com`, hence thwarting the attack.

5.2 High Confidence Channels

CGuard uses high confidence channels to update a cache entry when that entry is affected by a detected attack. As an abstract framework, CGuard can be instantiated to use any desirable high confidence channels in a preferred order in actual deployment. As discussed in Section 3, various previously proposed defense proposals currently suffer availability issues and a lack of robustness given certain corner cases. We envision that by combining them in the CGuard way, the high confidence channels that stem from the various proposals can effectively cover for each other.

In our particular instantiation, we choose to use plain UDP as the base channel, as it has the best availability and performance. Alternative, one can choose to always use a base channel that is of higher confidence (*e.g.*, TCP, 0x20 Encoding [22], etc.) at additional performance and management costs.

We now present the high confidence channels that we consider in our implementation of CGuard for the resolver to resolve DNS queries under active cache poisoning attack. For ease of management, we focus on leveraging existing standardized channels (*i.e.*, UDP, TCP, and DNSSEC), though this is not an inherent restriction of the CGuard framework. We leave the incorporation and the subsequent evaluation of other high confidence channels (*e.g.*, the likes discussed in Section 3, as well as the new proposed standards of DNS over TLS/DTLS [RFC7858, RFC8094]) as future work.

Having multiple such channels is vital as some of these channels are not supported by certain name servers. Based on our measurements in Section 3, 14.79% of Alexa's top 15,000 domains have authoritative name servers that support neither TCP nor DNSSEC. Consequently, we propose to leverage three high confidence channels in addition to TCP and DNSSEC. The first two are both based on the fact that the mappings between domain names and IP addresses of a large number of domains remain fairly stable over time; we call this the **domain name-IP stability** observation. One may question the mapping stability given the prevalence of CDNs (Content Delivery Networks). We note that not all CDNs perform DNS-based redirection. In fact, a large portion of CDNs leverage anycast-based services [1, 21, 24, 37, 46, 55], where one IP address is shared by various hosts distributed in different locations. Comparing to DNS-based CDNs, anycast-based setups are generally considered to be more resilient to DDoS attacks and more compatible with public resolvers [16].

UDP double query (UDQ). One possible high confidence channel that leverages the *domain name-IP stability* observation, is for the resolver to simultaneously send out two queries with the same question over UDP. If answers to both queries match and there is no attack detected then we can accept the answers with high confidence. We call this channel "UDQ".

UDQ support. This experiment is to determine when an authoritative name server is queried two consecutive times over UDP about the same domain, would the addresses it returns be the same. For each of the name servers we found, we send two `A` queries about a domain for which it is authoritative. Once the responses are received, we compare the IP addresses in the two responses. If the sets of IP addresses match exactly, the domain is said to support double query. We found **89.57**% *of Alexa's top* 15,000 *domains exhibit double query stability.*

Long-term stability (LTS). This channel is applicable for a query whose answer is already in the cache but the entry's TTL has expired. In such a case, one can send out a DNS query over UDP and if the answer (*i.e.*, IP address for A query) in the DNS response matches the value in the cache, then we can take that answer to be correct with high confidence. This is safe under the assumption that the DNS cache was *not* poisoned to begin with.

LTS support. For measuring this, we keep querying name servers about domains for which they are authoritative over a period of 30 days (from Sept. 11, 2015 to Oct. 11, 2015). Our goal is to validate that the mappings between a domain and its addresses remain mostly stable over time. *We observed that for the top* 15,000 *domains,* **94.3**%*,* **92.7**%*, and* **88.8**% *of them have the exact same list of addresses after* 7*,* 14*, and* 28 *days, respectively.*

In addition to the main domains (*e.g.*, `nih.gov`), we also correlated with the data from DNSCensus 2013 [3]. We found 25502 existing subdomains (*e.g.*, `dpcpsi.nih.gov`), and measured the stability of their address over a period of two weeks (from May 4, 2016 to May 18, 2016). *About* **95**% *of them have the exact same list of addresses after* 12 *days.*

Public resolver over TCP (PR-TCP) as a last resort. According to our experiments, 5.41% of Alexa top 15,000 domains do not support any of the aforementioned high confidence channels. In such cases, one possibility is to ask a somewhat trusted public resolver to resolve the query over TCP. We call this channel "PR-TCP". The rationale for not always using PR–TCP is that (1) simply delegating all queries to PR-TCP has additional privacy implications; (2) there may be conflict of interests between the recursive resolver provider and companies behind public resolvers like Google and Cisco; (3) always relying on public resolvers may negatively impact the optimality of DNS-based redirections in CDNs. We thus consider using PR–TCP only as a last resort.

High confidence channel preference. Our preference criteria is jointly based on network delay and confidence level. LTS is our first choice as it has the lowest cost (the resolver has to send only one UDP query) while delivering a fairly high confidence. We then prefer DNSSEC as it can run on top of UDP and does not incur the connection establishment cost of TCP while delivering a higher confidence. Finally, TCP has a higher confidence than UDQ. Hence, we use the following preference in our mechanism: LTS > DNSSEC > TCP > UDQ.

If desired, CGuard can be flexibly instantiated with **other preferences** on the high confidence channels. For example, one might prioritize confidence level and use the channels in this order: DNSSEC > TCP > LTS > UDQ.

5.3 CGuard's Adaptive Defense: Putting it Altogether

We now briefly describe how CGuard's active, fine-grained attack detection and use of high confidence channels achieve a robust defense against racing cache poisoning attacks; removing the incentive of a rational attacker to attack a CGuard resolver and consequently achieving deterrence. Note that when we refer to the attack state of a CGuard resolver we mean CGuard's attack cache and global attack state (timestamps of most recent mm_thresh attacks).

When a CGuard resolver receives a client DNS query and the answer to the query is in the cache, CGuard responds with its cached result. In case of a cache miss, CGuard consults the attack state to see whether the system is either under global attack, or a particular query, domain or zone is under attack. In case of an attack, CGuard uses one of the high confidence channels to resolve this query and update its cache. If a domain's authoritative name server `ns` does not support a particular high confidence channel hc_1, it goes to the next available high confidence channel according to our priority (see Section 5.2) and in the process stores the information that `ns` does not support hc_1. When resolving for the same domain (or, its zone) in future, this information can be used to avoid walking the priority chain of high confidence channels. When such information expires in the cache, CGuard will once again walk through the chain of possible high confidence channels, which can be thought of as a passive probing of the domain for newly added high confidence channels. In the case that neither the query (domain or its zone) nor the whole system is under attack CGuard resumes its benign behavior by using UDP.

When CGuard receives a matched DNS response through a high confidence channel, it caches it and uses it to possibly respond to pending client queries. In case the response arrives through UDP (as in normal DNS operation), it consults the attack state to see whether to cache it. If the attack state does not point to attack to the query (or, its zone) or the system, it caches it; otherwise, it drops the packet and sends the query with a high confidence channel. In case of a mismatched response through UDP (as in normal DNS operation), CGuard updates the system's attack state.

6 Evaluation

Setup. For evaluation purposes, we use Unbound 1.5.4 as a recursive resolver, and also use it as an authoritative name server (of a local zone). Our version of CGuard was also implemented based on Unbound 1.5.4. Here we evaluate CGuard along two dimensions: (1) the overhead it incurs, and (2) how resistant it is to attacks. We emphasize that this is just a proof-of-concept implementation; it does not mean CGuard can only be implemented in Unbound. In fact, one should easily be able to implement CGuard in other software.

We use `pidstat` from `sysstat` [25] to capture statistics about CPU usage, and `pmap` for memory measurements. All readings are taken after waiting an initial short period of time for the system to stabilize.

CPU Overhead. We evaluate the overhead of CGuard under normal operations by setting up both the original and modified Unbound as recursive resolvers, sharing the same configurations. We then issue queries using queryperf++ asking the resolver about `A` records of Alexa's top domains. We let this experiment run for 450 seconds separately for each setup. We collected the CPU usage and the service rate that the resolver delivered. The results are depicted in Figure 3. As can be seen, CGuard does not incur significant overhead comparing to its unhardened counterpart in the benign case. We note that the actual service

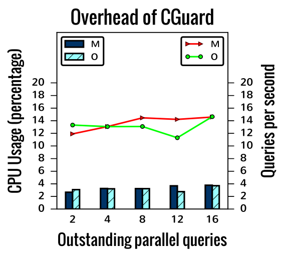

Fig. 3. Service rate and CPU usage of {Modified, Original} Unbound

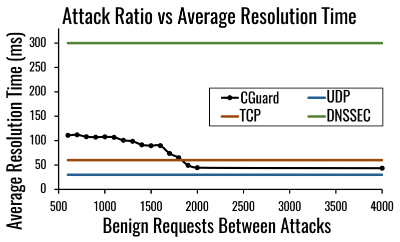

Fig. 4. Probabilistic modeling of CGuard's average resolution time

rate delivered by the resolver fluctuates due to environmental reasons including but not limited to non-deterministic network delay.

Memory Overhead. We also measured the memory usage of CGuard for benign cases and when under attack. We used the same setup as above and ran the 450-second query blasting experiment six times. We measured the memory usage of CGuard and the original Unbound. We use RSS to denote resident set size and VIRT to denote virtual memory. On average, after system initialization, the original Unbound consumes **22**,**727** kB RSS (**547**,**696** kB VIRT) while CGuard consumes a comparable **22**,**278** kB RSS (**547**,**638** kB VIRT). After handling all the queries, on average, the original Unbound consumes **43**,**000** kB RSS (**548**,**519** kB VIRT) and CGuard consumes **42**,**887** kB RSS (**548**,**570** kB VIRT), which shows that the memory overhead, if any, is negligible. We additionally ran 7 attacks on CGuard targeting different domains. When under attack, the memory usage of CGuard peaked at **44**,**180** kB RSS (**548**,**572** kB VIRT). We note that OS-level memory measurements can vary due to various environmental factors. The results here is not a proof of CGuard being more memory efficient but they show that the memory overhead is not heavy.

Probabilistic Modeling. To determine the average latency CGuard introduces, we probabilistically model its operation under varying volumes of attack traffic. The result in Figure 4 depicts the average of 5 separate simulation runs. For simplicity, we assume the following key parameters: (a) latency (ms) for each channel: 30 for UDP, 60 for TCP, 300 for DNSSEC, 40 for UDP double query, and 55 for asking Google; (b) the probability of a domain supporting each channel: 100% for UDP, 85% for TCP , 85% for long-term stability, 85% for UDP double query, 1% for DNSSEC, and 100% for Google Public Resolver. As shown in Figure 4, on average, CGuard has a much lower resolution time than always using DNSSEC, and when attacks occur only infrequently, it will also perform better than always using TCP without optimizations.

Attack Resistance. As Nominum's Vantio CacheServe is a proprietary product, we do not attempt to evaluate it. Instead, we launch PKAs on the original and modified Unbound, both configured as recursive resolvers, running on a single thread, and only 10 bit entropy is used in the TXID. Note that **not** using the full 16-bit gives an attacker an advantage in launching a successful attack since it is easier to match the TXID. To mitigate the influence of source port

Table 1. Six runs of Parallel Kaminsky attack on (Original and Modified) Unbound 1.5.4 recursive resolver

	Turn #	1	2	3	4	5	6
O	Instances	2266	1331	**3072**	1884	2519	1674
	Result	Poisoned	Poisoned	Failed	Poisoned	Poisoned	Poisoned
M	Instances	**3072**	**3072**	**3072**	**3072**	**3072**	**3072**
	Result	Failed	Failed	Failed	Failed	Failed	Failed

randomization, we forced Unbound to use a fixed source port. This resembles a scenario where the recursive resolver is behind a NAT.

We repeat the attack 6 times. Each time we stop the attack after at most $3 \times 2^{10} = 3,072$ instances. For each attack instance, we randomly generate a likely non-existent sub-domain of a victim domain and send a query about it to the resolver. Consequently, the recursive resolver is going to perform iterative queries in attempt to find an answer, which gives an attack window. We then send one forged DNS response to the resolver by spoofing the sender address to be one of the authoritative name servers. With the unprotected original Unbound, each attack instance has an independent probability of $\frac{1}{2^{10}}$ of being successful and hence the expected number of instances needed to have one success is simply 2^{10}. The results of the 6 attack attempts can be found in Table 1. Across the 5 successful attempts, on average, it took $1,934$ instances for the attack to be successful, which is within 2 standard deviations from the theoretical expected value. When Unbound was hardened with CGuard, it detected the attack and did not allow the cache to be poisoned. Hence, all attack instances in all 6 runs failed. This shows the efficacy of CGuard in detecting and defending against cache poisoning.

Threat of Performance Attack. One might suspect that given an adaptive attacker, CGuard might be forced to continuously stay on system-wide attack mode. This is true, however, the outcome of such a case should be close to that of always using the heaviest high-confidence channel (say DNSSEC), with a small manageable detection and bookkeeping overhead. CGuard would then opportunistically fall back to low-confidence channels once the attacks are over.

7 Conclusion

We present CGuard that proactively detects whether cache entries are under active attack and protects them by using high confidence channels. CGuard complements existing cache poisoning solutions; unlike many existing solutions, DNS resolvers can enjoy CGuard's protection for a minimal deployment cost without having to rely on any substantial effort from domain owners. By incorporating multiple high confidence channels, CGuard also enables them to cover for each other in terms of availability and against some tricky corner cases. Our evaluation of CGuard—implemented in Unbound DNS resolver 1.5.4—demonstrates that CGuard is effective at thwarting DNS cache poisoning attacks and incurs minimal overhead under normal operation. We envision that by taking away the attacker's incentive to launch DNS cache poisoning attacks, CGuard essentially turns the existence of high confidence channels into a deterrence. Such a deterrence-based defense mechanism can be relevant to other applications.

Being a critical infrastructure of the Internet, DNS continues to attract efforts on making it more robust and trustworthy. Recent developments include the various documents produced by the DNS Private Exchange (dprive) workgroup. Motivated by the need for privacy and confidentiality against eavesdroppers, dprive recently proposed new standards on DNS over TLS/DTLS [RFC7858, RFC8094], which also adds resilience against cache poisoning attacks. As for future work, we intend to investigate the adoption of the TLS/DTLS-based high confidence channels in CGuard instantiations, and possibly evaluate their performance when compared with other channels.

References

1. 5 Myths about Content Delivery Networks and the truths you should know, `https://www.thatwhitepaperguy.com/downloads/5-CDN-Myths.pdf`
2. Vulnerability Note VU 800113: Multiple DNS implementations vulnerable to cache poisoning. Tech. rep., US CERT Vulnerability Notes Database (2008)
3. DNS Census 2013 (2013), `https://dnscensus2013.neocities.org`
4. DNS, DNSSEC and Google's Public DNS Service (2013), `http://www.circleid.com/posts/20130717_dns_dnssec_and_googles_public_dns_service/`
5. Google's Malaysian Domains Hit with DNS Cache Poisoning Attack (2013), `http://www.tripwire.com/state-of-security/latest-security-news/googles-malaysian-domains-hit-dns-cache-poisoning-attack/`
6. DNS poisoning slams web traffic from millions in China into the wrong hole (2014), `http://www.theregister.co.uk/2014/01/21/china_dns_poisoning_attack/`
7. Google Public DNS – Security Benefits (2014), `https://developers.google.com/speed/public-dns/docs/security`
8. CloudFlare Enables Universal DNSSEC for Its Millions of Customers for Free (2015), `http://www.marketwired.com/press-release/cloudflare-enables-universal-dnssec-for-its-millions-of-customers-for-free-2072174.htm`
9. DNSSEC name and shame! (2015), `https://dnssec-name-and-shame.com/`
10. Ager, B., Dreger, H., Feldmann, A.: Predicting the DNSSEC overhead using DNS traces. In: 40th IEEE CISS (2006)
11. Antonakakis, M., Dagon, D., Luo, X., Perdisci, R., Lee, W., Bellmor, J.: A centralized monitoring infrastructure for improving DNS security. In: Recent Advances in Intrusion Detection. pp. 18–37. Springer (2010)
12. APNIC Labs: Use of DNSSEC Validation for World (2015), `http://stats.labs.apnic.net/dnssec/XA`
13. Assolini, F.: Attacks against Boletos (2014), `https://securelist.com/attacks-against-boletos/66591/`
14. Bernstein, D.J.: DNSCurve: Usable security for dns (2009), `http://dnscurve.org/`
15. Bernstein, D.J.: DNS forgery (2002), `http://cr.yp.to/djbdns/forgery.html`
16. Calder, M., Flavel, A., Katz-Bassett, E., Mahajan, R., Padhye, J.: Analyzing the performance of an anycast CDN. In: Proceedings of ACM IMC. pp. 531–537 (2015)
17. CCCen: An overview of secure name resolution [29c3] (2013), `https://www.youtube.com/watch?v=eOGezLjlzFU`

18. Cimpanu, C.: Around four in five dnssec servers can be hijacked for ddos attacks (2016), http://news.softpedia.com/news/around-four-in-five-dnssec-servers-can-be-used-in-ddos-attacks-507503.shtml
19. CommunityDNS: Performance testing of BIND, NSD and CDNS platforms on identical hardware. (2010), http://communitydns.net/DNSSEC-Performance.pdf
20. Constantin, L.: DNS Cache Poisoning Used in Brazilian Phishing Attack (2011), http://news.softpedia.com/news/DNS-Cache-Poisoning-Used-in-Brazilian-Phishing-Attack-212328.shtml
21. Czarny, M.: How Anycast IP Routing Is Used at MaxCDN (2013), https://www.maxcdn.com/blog/anycast-ip-routing-used-maxcdn/
22. Dagon, D., Antonakakis, M., Vixie, P., Jinmei, T., Lee, W.: Increased DNS forgery resistance through 0x20-bit encoding: security via leet queries. In: Proceedings of the 15th ACM CCS. pp. 211–222 (2008)
23. Duan, H., Weaver, N., Zhao, Z., Hu, M., Liang, J., Jiang, J., Li, K., Paxson, V.: Hold-on: Protecting against on-path DNS poisoning. In: Securing and Trusting Internet Names (SATIN) (2012)
24. Flavel, A., Mani, P., Maltz, D., Holt, N., Liu, J., Chen, Y., Surmachev, O.: Fastroute: A scalable load-aware anycast routing architecture for modern cdns. In: 12th USENIX NSDI. pp. 381–394 (2015)
25. Godard, S.: sysstat - System Performance tools for the Linux operating system... (2015), https://github.com/sysstat/sysstat
26. Guðmundsson, Ó., Crocker, S.D.: Observing dnssec validation in the wild. Securing and Trusting Internet Names (SATIN) (2011)
27. Herzberg, A., Shulman, H.: Retrofitting security into network protocols: The case of dnssec. IEEE Internet Computing 18(1), 66–71 (Jan 2014)
28. Herzberg, A., Shulman, H.: Security of patched DNS. In: Computer Security–ESORICS 2012, pp. 271–288. Springer (2012)
29. Hubert, A., van Mook, R.: Measures for Making DNS More Resilient against Forged Answers (Jan 2009), https://www.rfc-editor.org/rfc/rfc5452.txt
30. Hussain, I.: Google.com.bd down (2016), http://www.dhakatribune.com/feature/2016/12/20/google-com-bd/
31. Huston, G., Michaelson, G.: Measuring DNSSEC use. In: IEPG Meeting. vol. 87
32. Huston, G., Michaelson, G.: Measuring DNSSEC performance (2013), http://impossible.rand.apnic.net/ispcol/2013-05/dnssec-performance.pdf
33. Infoblox: INFOBLOX DNS THREAT INDEX (2015), https://www.infoblox.com/sites/infobloxcom/files/resources/infoblox-white-paper-dns-threat-index-q2-2015-report.pdf
34. JUNIPER TechLibrary: Network Address Translation Feature Guide for Security Devices - Disabling Port Randomization for Source NAT (CLI Procedure) (2016), https://www.juniper.net/documentation/en_US/junos/topics/task/configuration/nat-security-source-port-randomization-disabiling-cli.html
35. Kaminsky, D.: Black Ops 2008: It's The End Of The Cache As We Know It (2008)
36. Kaminsky, D.: DNSSEC Interlude 2: DJB@CCC | Dan Kaminsky's Blog (2011), http://dankaminsky.com/2011/01/05/djb-ccc/
37. Levine, M.: Measuring Throughput Performance: DNS vs. TCP Anycast Routing (2014), http://www.cachefly.com/2014/07/11/measuring-throughput-performance-dns-vs-tcp-anycast-routing/
38. Lian, W., Rescorla, E., Shacham, H., Savage, S.: Measuring the practical impact of DNSSEC deployment. In: USENIX Security. pp. 573–588 (2013)
39. Lindstrom, A.: DNSSEC implementation in Sweden (2012), https://www.antonlindstrom.com/2012/01/02/dnssec-implementation-in-sweden.html

40. Lowe, G., Winters, P., Marcus, M.L.: The great DNS wall of china (Dec 2007)
41. Nice, B.V.: High Performance DNS Needs High Performance Security (2012), http://nominum.com/high-performance-dns-needs-high-performance-security/
42. NIST National Vulnerability Database: CVE-2002-2211 (2002), http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2002-2211
43. Park, K., Pai, V.S., Peterson, L.L., Wang, Z.: CoDNS: Improving DNS performance and reliability via cooperative lookups. In: OSDI. vol. 4, pp. 14–14 (2004)
44. Perdisci, R., Antonakakis, M., Luo, X., Lee, W.: WSEC DNS: Protecting recursive DNS resolvers from poisoning attacks. In: Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on. pp. 3–12. IEEE (2009)
45. Poole, L., Pai, V.S.: ConfiDNS: Leveraging scale and history to improve DNS security. In: WORLDS (2006)
46. Prince, M.: A Brief Primer on Anycast (2011), https://blog.cloudflare.com/a-brief-anycast-primer/
47. Rashid, F.Y.: Poorly configured DNSSEC servers at root of DDoS attacks (2016), http://www.infoworld.com/article/3109581/security/poorly-configured-dnssec-servers-at-root-of-ddos-attacks.html
48. Raywood, D.: Irish ISP Eircom hit by multiple attacks that restrict service for users (2009), http://www.scmagazineuk.com/irish-isp-eircom-hit-by-multiple-attacks-that-restrict-service-for-users/article/140243/
49. Schuba, C.: Addressing weaknesses in the domain name system protocol. Ph.D. thesis, Purdue University (1993)
50. Seltzer, L.: Report Claims DNS Cache Poisoning Attack Against Brazilian Bank and ISP (2009), http://www.eweek.com/c/a/Security/Report-Claims-DNS-Cache-Poisoning-Attack-Against-Brazilian-Bank-and-ISP-761709
51. Shulman, H., Waidner, M.: One key to sign them all considered vulnerable: Evaluation of dnssec in the internet. In: NSDI. pp. 131–144 (2017)
52. Son, S., Shmatikov, V.: The hitchhiker's guide to dns cache poisoning. Security and Privacy in Communication Networks pp. 466–483 (2010)
53. Spring, J.: Probable Cache Poisoning of Mail Handling Domains (2014), https://insights.sei.cmu.edu/cert/2014/09/-probable-cache-poisoning-of-mail-handling-domains.html
54. StatDNS: TLD Zone File Statistics (2016), http://www.statdns.com/
55. Support, K.: Anycast (2016), https://www.keycdn.com/support/anycast/
56. Tatuya, J.: queryperf++ (2014), https://github.com/jinmei/queryperfpp
57. Verisign Labs: DNSSEC Scoreboard, http://scoreboard.verisignlabs.com/
58. Virus Bulletin: DNS cache poisoning used to steal emails (2014), https://www.virusbtn.com/blog/2014/09_12.xml
59. Wikipedia: Deterrence theory — Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Deterrence_theory
60. Yao, Y., He, L., Xiong, G.: Trustworthy Computing and Services: International Conference, ISCTCS 2012, Revised Selected Papers, chap. Security and Cost Analyses of DNSSEC Protocol, pp. 429–435. Springer Berlin Heidelberg (2013)
61. Yuan, L., Kant, K., Mohapatra, P., Chuah, C.N.: DoX: A peer-to-peer antidote for DNS cache poisoning attacks. In: IEEE ICC'06. vol. 5 (2006)
62. Zhu, L., Hu, Z., Heidemann, J., Wessels, D., Mankin, A., Somaiya, N.: Connection-oriented DNS to improve privacy and security (extended). Tech. Rep. ISI-TR-2015-695 (Feb 2015), http://www.isi.edu/~johnh/PAPERS/Zhu15c.html