PROJECT REPORT ON

Implementation of Recursive DNS Resolver in Python and Unbound

Report submitted to the SASTRA Deemed to be University
as the requirement for the course

CSE302: COMPUTER NETWORKS

Submitted by

NUNNA SRINIDHI

124003418, BTech CSE

SCHOOL OF COMPUTING

THANJAVUR, TAMIL NADU, INDIA – 613 401

SCHOOL OF COMPUTING

THANJAVUR – 613 401

<u>BONAFIDE CERTIFICATE</u>

This is to certify that the report titled **“Implementing Recursive DNS Resolver in Python and Unbound”** submitted as a requirement for the course, **CSE302: COMPUTER NETWORKS** for B.Tech. is a bonafide record of the work done by Ms **NUNNA SRINIDHI (124003418, B.Tech CSE**) during the academic year 2022-23, in the School of Computing

Project Based Work Viva Voce held on 13/12/2022

Examiner 1 **Examiner 2**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

DNS	Domain Name System
DNSSEC	Domain Name System Security Extensions
DoT	DNS over TLS
MiTM	Man in The Middle Attacks
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
TTL	Time-to-Live
IETF	Internet Engineering Task Force
RFC	Request for Comments
CNAME	Canonical name record
RRSIG	Resource Record Set
DS	Delegation Signer
NSEC	Next Secure Record
NSEC3	Next Secure Record Version 3
ZSK	Zone Signing Key
KSK	Key Signing Key
SOA	Start of Authority
EDNS	Extension Mechanisms for DNS

ABSTRACT

Domain Name System (DNS) is an integral part of the internet, which serves as the phonebook of the Internet. DNS translates domain names to IP addresses so machines can load internet resources without the user having to remember the IP address of their favourite sites. But DNS consists mostly of unencrypted responses and queries over UDP.

DNS is almost invented 30 years ago. So, privacy and security aspects are not considered initially. Due to the unencrypted nature of the DNS responses and queries, it can be used to censor the traffic, spy on the user's internet activity and spoof the IPs in various phishing attacks and DNS cache poisoning attacks. As the internet progressed to the current state, the security and privacy aspects are being implemented.

Here, I have implemented DNS recursive resolver in Python and Unbound with assortment of security and privacy enhancements against DNS cache poisoning attacks, censorship using Unbound 1.9.4, and a python code which mimics the recursive resolver using dnspython library.

KEYWORDS: DNS, DNSSEC, MiTM, Privacy,UDP

INTRODUCTION

The Internet is essentially a collection of networks that are connected through wired (underwater cables) and wireless (satellite) links. Every connection has a distinct IP address that other computers can use to locate the device. There are two types of IP addresses: IPv4 (192.168.1.1) and IPv6 (2409:4072:6114:f4c0:fd36:8259:57b6:435e). The more recent IPv6 addresses are far more difficult to remember than their IPv4 counterparts, and we also find it impossible to remember the IP addresses of every single website because the resource owner has the right to alter their IP addresses at any time. As a result, the necessity for DNS—a system that converts domain names into IP addresses that are friendly to computers—arose.

The Domain Name System (DNS) is an application layer protocol that acts as the phonebook of the internet. DNS uses port 53 and UDP to transmit data. The naming scheme used to categorize computers, services, and other resources accessible over the internet or other internet protocol networks is hierarchical and decentralized. These are most frequently used to translate between human-friendly domain names and the corresponding numerical IP addresses that computers require in order to find services and gadgets by using the underlying network protocols. Since 1985, DNS has played a crucial role in the operation of the Internet.

Due to the fact that UDP is faster than TCP and requires no handshake between the server and client, which lessens the burden on the DNS servers, as well as the fact that DNS queries and responses are frequently quite brief and fit well within UDP segments, DNS transfers data through UDP.

A single UDP request from the client and a single UDP response from the server make up a DNS query. Larger UDP packets are utilised when the answer is longer than 512 bytes and EDNS is supported by both the client and the server. If not, a new TCP inquiry request is sent. Zone transfers are another duty that TCP is used for. Some resolver implementations send all requests via TCP.

Domains:

A tree of domain names makes up the domain name space. The information related to the domain name is stored at each node in the tree. Starting with the DNS root zone, the tree is divided into zones. The domain name space was split into two major domain groups when the Domain Name System was developed in the 1980s. The two-character territory codes of the ISO-3166 country abbreviations served as the foundation for the country code top-level domains (ccTLD). In addition, a set of seven generic top-level domains (gTLDs) representing various name categories and multi-organizations was implemented. The domains were gov, edu, com, org, and net. The highest level of Internet domain names are these two categories of top-level domains (TLDs). The DNS root zone, which is hierarchical, is made up of top-level domains. A top-level domain label appears at the end of every domain name.

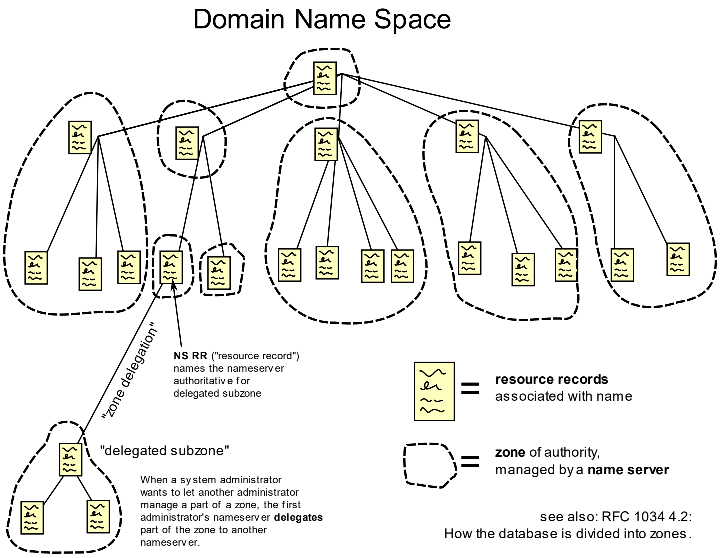

Figure 1

NAMESERVERS

A distributed database system using the client-server architecture looks after the Domain Name System. The name servers are the nodes in this database. The name servers of all domains that are subordinate to a given domain are published by at least one authoritative DNS server for each domain. The root name servers, the servers to contact when seeking up (resolving) a TLD, provide service to the top of the hierarchy.

ADDRESS RESOLUTION MECHANISM

- **DNS recursor:** A server created to answer DNS requests from client computers. The DNS Recursive resolver is another name for this. In order to respond to the client's DNS query, this recursor must make further requests.

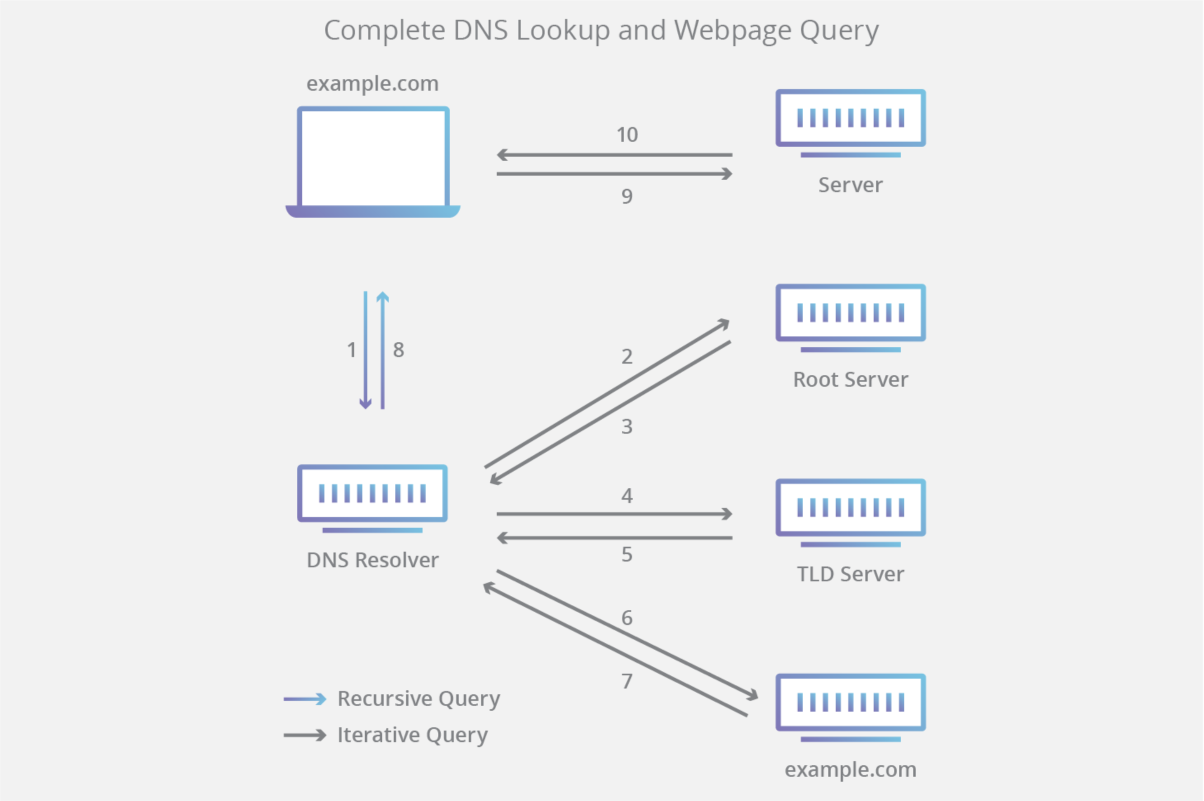

Figure 1a

A recursive resolver is the first stop in a DNS query. The recursive resolver acts as a middleman between a client and a DNS nameserver. A recursive resolver will respond to a DNS request from a web client by using cached information, or it will make three requests: one to a root nameserver, one to a TLD nameserver, and one to an authoritative nameserver. The recursive resolver then sends a response to the client after getting a reply from the authoritative nameserver that includes the requested IP address.

The recursive resolver will save the data it receives from authoritative name servers during this procedure. The resolver can skip the nameserver communication process and just provide the client with the requested record from its cache when a client requests the IP address of a domain name that was just requested by another client.

- **Root Nameserver:**

  A recursive resolver's search for DNS records starts with one of the 13 DNS root nameservers. When a recursive resolver asks a root server for help, the root nameserver responds by pointing the recursive resolver to a TLD nameserver based on the domain's extension (.com, .net, .org, etc.). The Internet Corporation for

Assigned Names and Numbers, a non-profit, is in charge of maintaining the root nameservers (ICANN).

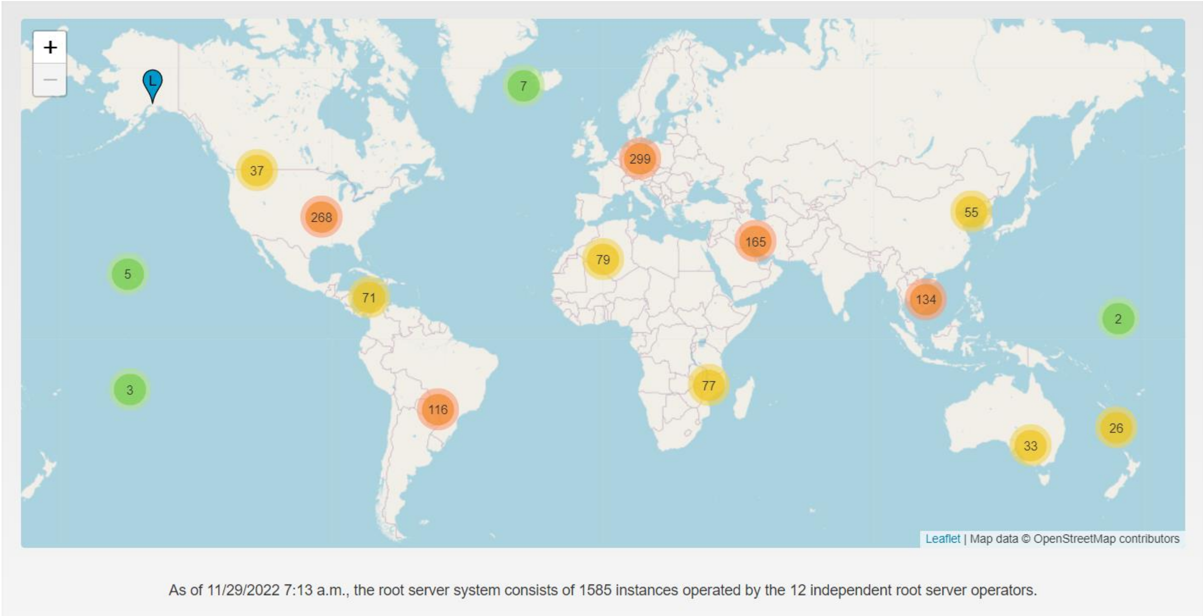

Figure 1b

There are more than just 13 machines in the root nameserver system, despite the existence of just 13 root nameservers. There are 13 different types of root nameservers, but each one is duplicated globally and uses Anycast routing to deliver quick responses. As of 29/11/2022 07:13am., the root server system consists of 1585 instances operated by the 12 idependent root server operators.[1]

List of Root Servers

HOSTNAME	IP ADDRESSES	OPERATOR
a.root-servers.net	198.41.0.4, 2001:503:ba3e::2:30	Verisign, Inc.
b.root-servers.net	199.9.14.201, 2001:500:200::b	University of Southern California, Information Sciences Institute
c.root-servers.net	192.33.4.12, 2001:500:2::c	Cogent Communications
d.root-servers.net	199.7.91.13, 2001:500:2d::d	University of Maryland
e.root-servers.net	192.203.230.10, 2001:500:a8::e	NASA (Ames Research Center)
f.root-servers.net	192.5.5.241, 2001:500:2f::f	Internet Systems Consortium, Inc.
g.root-servers.net	192.112.36.4, 2001:500:12::d0d	US Department of Defense (NIC)
h.root-servers.net	198.97.190.53, 2001:500:1::53	US Army (Research Lab)
i.root-servers.net	192.36.148.17, 2001:7fe::53	Netnod
j.root-servers.net	192.58.128.30, 2001:503:c27::2:30	Verisign, Inc.
k.root-servers.net	193.0.14.129, 2001:7fd::1	RIPE NCC
l.root-servers.net	199.7.83.42, 2001:500:9f::42	ICANN
m.root-servers.net	202.12.27.33, 2001:dc3::35	WIDE Project

Figure 2

- **TLD Nameserver:**

A TLD nameserver keeps track of data for all domain names that have the same domain extension, such as.com,.net, or whatever follows the last dot in a URL. For instance, a.com TLD nameserver stores data for each website whose domain name ends in.com. A recursive resolver would send a query to a.com TLD nameserver in the case of a user searching for google.com after receiving a response from a root nameserver, and that nameserver would respond by pointing to the authoritative nameserver for that domain.

The Internet Assigned Numbers Authority (IANA), an arm of ICANN, is in charge of managing TLD nameservers. The TLD servers are divided into two categories by the IANA:

Top-level domains (TLDs) with a generic suffix are those that are not country-specific; among of the most well-known generic TLDs are.com,.org, .net, .edu, and .gov. Top-level domains with a country code include any domains that are particular to a state or nation. A few examples are.uk,.us,.ru, and.in.

A third category for infrastructure domains exists, but it is almost ever used. This category was developed for the .arpa domain, which served as a temporary domain during the development of the current DNS and has mainly historical significance today.

- **Authoritative Nameserver:**

A TLD nameserver's response to a recursive resolver will point it in the direction of an authoritative nameserver. The final stop on the resolver's path to an IP address is typically the authoritative nameserver. It can provide a recursive resolver with the IP address of that server found in the DNS A record or, if the domain has a CNAME record (alias), it will provide the recursive resolver with an alias domain, at which point the recursive resolver will have to conduct a completely new DNS lookup to procure a record from an authoritative nameserver. The authoritative nameserver contains information specific to the domain name it serves (for example, cloudflare (often an A record containing an IP address).

Circular dependencies and glue records

In delegations, name servers are designated by name rather than by IP address. This implies that in order to learn the IP address of the server being referenced, a resolving name server must send out another DNS request. There is a circular dependency if the name specified in the delegation is a subdomain of the domain for which the delegation is being provided.

In this situation, the name server delivering the delegation is also required to offer one or more IP addresses for the named authoritative name server. This knowledge is referred to as glue. The delegating name server offers the delegation in the authority portion of the DNS response and delivers the glue in the form of records in the extra section. A glue record combines an IP address and name server.

A computer first resolves ns1.example.org while attempting to resolve www.example.org, for instance, if ns1.example.org is the authoritative name server for example.org. This creates a circular dependency because example.org contains ns1, which must be resolved first. The delegation for example.org is combined with glue on the name server for the top-level domain org in order to eliminate the dependency. The IP addresses for ns1.example.org are provided by the glue records, which are address records. In order to query one of the domain's authoritative servers and complete the DNS query, the resolver uses one or more of these IP addresses.

Record caching

Caching results locally or in intermediary resolver hosts is a common strategy when implementing name resolution in apps to lessen the strain on the Domain Name System servers. The time to live (TTL), an expiration date after which the results must be destroyed or updated, is always connected with results retrieved from a DNS request. The administrator of the authoritative DNS server controls the TTL. A few seconds to several days or even weeks can be included in the validity period.

This distributed caching architecture causes DNS record changes to not immediately spread throughout the network, but rather necessitate all caches to expire and be refilled after the TTL. Basic guidelines for choosing adequate TTL values are provided by RFC 1912.

As the protocol permits caching for up to 68 years or no caching at all, some resolvers may override TTL values. Name servers authoritative for a zone, which are required to include the Start of Authority (SOA) record when reporting no data of the requested type exists, are responsible for negative caching, or the caching of the fact that a record does not exist. The TTL for the negative response is determined using the value of the minimum field in the SOA record and the TTL of the SOA itself.

Reverse lookup

A reverse DNS lookup is a query to the DNS for domain names when the IP address is known. An IP address can have multiple domain names associated with it. The DNS stores IP addresses in the form of domain names as specially formatted names in pointer (PTR) records within the infrastructure top-level domain arpa. In-addr.arpa is the domain for IPv4. For IPv6, the reverse lookup domain is ip6.arpa. The IP address is represented as a name in reverse-ordered octet representation for IPv4 and reverse-ordered nibble representation for IPv6.

When performing a reverse lookup, the DNS client converts the address into these formats before querying the name for a PTR record, as it does for any DNS query. For example, assuming Wikimedia has the IPv4 address 208.80.152.2, it is represented as a DNS name in reverse order: 2.152.80.208.in-addr.arpa. When a pointer (PTR) request is received, the DNS resolver begins by querying the root servers, which point to the servers of the American Registry for Internet Numbers (ARIN) for the 208.in-addr.arpa zone. ARIN's servers delegate 152.80.208.in-addr.arpa to Wikimedia, which responds with an authoritative response.

STEPS IN DNS LOOKUP

1. Your client queries the local DNS resolver (Stub resolver) about Wikipedia.org.
2. If the answer is already known, your local DNS resolver will check its cache and respond.
3. Otherwise, the request is routed to the specified recursive resolver. The recursive server will ask the DNS root servers, "Who is in charge of.org?"
4. The root server responds with a referral to the.org TLD servers.
5. Your recursive server will send the following query to one of the.org TLD DNS servers: "Who is handling wikipedia.org?"
6. The TLD server responds by directing the user to the authoritative name servers for wikipedia.org.

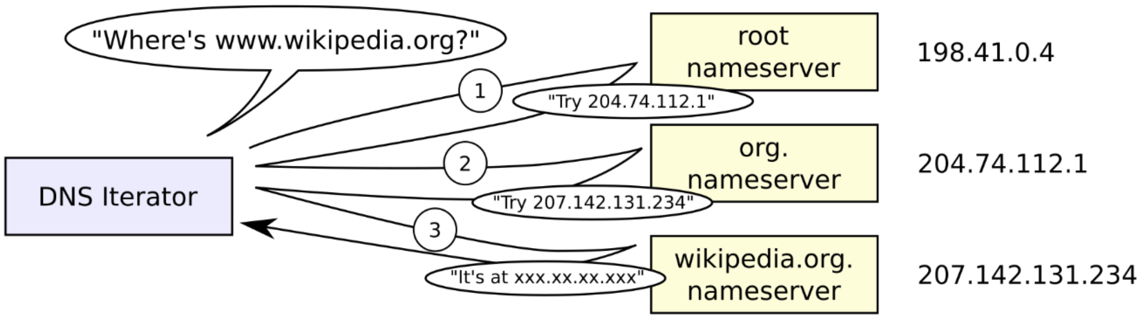

Figure 3

7. Your recursive server will ask the authoritative name servers, "What is the IP address of wikipedia.org?"
8. The IP address of the domain wikipedia.org will be returned by the authoritative server.

9. The recursive server will send the response to your local resolver, which will then respond to your client and inform it of the outcome of its request.
10. Finally, your local resolver will save the answer in its cache so that it can respond faster the next time one of your clients queries the same domain.

TYPES OF DNS RECORDS

TYPE	DESCRIPTION
A	Address record
AAAA	IPv6 address record
CNAME	Canonical name record - Alias of one name to another: the DNS lookup will continue by retrying the lookup with the new name.
SOA	Start of authority record - Stores information about domains and is used to direct how a DNS zone propagates to secondary name servers
NS	Name server record - Delegates a DNS zone to use the given authoritative name servers
MX	Mail exchange record - Maps a domain name to a list of message transfer agents for that domain
PTR	Pointer - A reverse of A and AAAA records, which maps IP addresses to domain names. These records require domain authority and can't exist in the same zone as other DNS record types (put in reverse zones)
TXT	Text Record - Allows administrators to add limited human and machine-readable notes and can be used for things such as email validation, site, and ownership verification, framework policies, etc., and doesn't require specific formatting
DNSKEY	DNS Key record - The key record used in DNSSEC. Uses the same format as the KEY record
DS	Delegation signer - The record used to identify the DNSSEC signing key of a delegated zone
RRSIG	DNSSEC signature - Signature for a DNSSEC-secured record set. Uses the same format as the SIG record

DNS MESSAGE FORMAT

The DNS protocol employs two types of DNS messages, queries and replies, both of which follow the same format. Each message has a header and four sections: a question, an answer, authority, and an extra space. The content of these four sections is controlled by a header field (flags).

Identification, Flags, Number of questions, Number of answers, Number of authority resource records (RRs), and Number of additional RRs are the fields in the header section. Each field is 16 bits long and appears in the specified order. Responses are matched with queries using the identification field. The flag field consists of sub-fields as follows:

Header Flags Format

Field	**Description**	**Length (bits)**
QR	Indicates if the message is a query (0) or a reply (1)	1
OPCODE	The type can be QUERY (standard query, 0), IQUERY (inverse query, 1), or STATUS (server status request, 2)	4
AA	Authoritative Answer, in a response, indicates if the DNS server is authoritative for the queried hostname	1
TC	TrunCation, indicates that this message was truncated due to excessive length	1
RD	Recursion Desired, indicates if the client means a recursive query	1
RA	Recursion Available, in a response, indicates if the replying DNS server supports recursion	1
Z	Zero, reserved for future use	3

Following the flag, the header concludes with four 16-bit integers containing the number of records in each of the following sections, in the same order.

Question section

The format of the question section is simpler than that of the resource record format used in the other sections. Each question record (there is usually just one in the section) contains the following fields:

Resource record (RR) fields

Field	Description	Length (octets)
NAME	Name of the requested resource	Variable
TYPE	Type of RR (A, AAAA, MX, TXT, etc.)	2
CLASS	Class code	2

TYPES OF DNS QUERIES

1. **Recursive query:** A DNS client expects a DNS server (typically a DNS recursive resolver) to respond to a recursive query with either the requested resource record or an error message if the resolver cannot find the record.

2. **Iterative query**: In this case, the DNS client will allow the DNS server to return the best possible response. If the queried DNS server does not match the query name, it will redirect to a DNS server authoritative for a lower level of the domain namespace. After that, the DNS client will query the referral address. This process is repeated with each DNS server in the query chain until an error or timeout occurs.

3. **Non-recursive query:** This typically happens when a DNS resolver client queries a DNS server for a record that it has access to, either because it is authoritative for the record or because the record is in its cache. A DNS server will typically cache DNS records to avoid additional bandwidth consumption and load on upstream servers.

DNS Caching

Browser DNS caching

By default, modern web browsers cache DNS records for a set amount of time. The goal is obvious; the closer DNS caching occurs to the web browser, the fewer processing steps are required to check the cache and make the correct requests to an IP address. When a DNS record request is made, the browser cache is the first location checked for the requested record.

In Chrome, you can see the status of your DNS cache by going to chrome://net-internals/#dns. In Edge, edge://net-internals/#dns

OS Level DNS Caching

The DNS resolver at the operating system level is the second and final local stop before a DNS query leaves your machine. A Stub Resolver or DNS client is the process within your operating system that is designed to handle this query. When a stub resolver receives a request from an application, it first checks its own cache to see if the record is available. If not, it sends a DNS query (with the recursive flag set) outside the local network to a DNS recursive resolver within the Internet service provider.

Protocol Extensions

The original DNS protocol had limited provisions for extension with new features. In 1999, RFC 2671 (superseded by RFC 6891) an extension mechanism, called Extension Mechanisms for DNS (EDNS) that introduced optional protocol elements without increasing overhead when not in use. This was accomplished through the OPT pseudo-resource record that only exists in wire transmissions of the protocol, but not in any zone files. Initial extensions were also suggested (EDNS0), such as increasing the DNS message size in UDP datagrams.

DNS Transport Protocols

DNS-over-UDP/53 ("Do53")

DNS has mostly responded to queries on User Datagram Protocol (UDP) port 53. Such queries consist of a clear-text request from the client sent in a single UDP packet, followed by a clear-text reply from the server sent in a single UDP packet. Larger UDP packets may be used when the length of the response exceeds 512 bytes and both the client and server support Extension Mechanisms for DNS (EDNS). The lack of transport-layer encryption, authentication, reliable delivery, and message length, among other things, limit the use of DNS-over-UDP.

DNS-over-TCP/53 ("Do53/TCP")

For DNS requests, answers, and in particular zone transfers, RFC 1123 from 1989 specifies an optional Transmission Control Protocol (TCP) transport. Longer responses, dependable delivery, and reuse of durable connections between clients and servers are all made possible by TCP through the fragmentation of lengthy responses.

DNS-over-TLS ("DoT")

In 2016, the IETF released a standard for encrypted DNS that uses regular Transport Layer Security (TLS) to secure the entire connection as opposed to just the DNS payload. TCP port 853 is used by DoT servers. Opportunistic encryption and authenticated encryption may be supported, according to RFC7858, but server or client authentication is not required.

DNS Crypt

Clients encrypt query payloads using servers' public keys, which are published in the DNS and may also be protected by DNSSEC signatures, according to the DNS Crypt protocol, which enabled DNS encryption on the downstream side of recursive resolvers. The same port that HTTPS-encrypted web traffic uses is 443, which DNS Crypt uses either via TCP or UDP. Through this, a sizable amount of firewall-traversal capacity was added in addition to privacy regarding the query's content. To support a "anonymized" mode akin to the "Oblivious DNS" proposal, DNS Crypt was further developed. In this mode, an ingress node receives a query that has been encrypted with the public key of a different server and relays it to that server, acting as an egress node and carrying out the recursive resolution. Since neither the ingress nor the egress nodes are aware of the contents of the query, a user's query and its associated user are kept private.

To counter the privacy and security issues following approaches can be used:

- VPN’s, which move DNS resolution to the VPN operator and hide user traffic from local ISP. But we ultimately have to trust the VPN operator which comes with its own concerns.

- TOR, which replaces traditional DNS resolution with anonymous .onion domains, hiding both name resolution and user traffic behind onion routing counter-surveillance. But Tor network is slow due to the multiple hops required and due to decreasing tor nodes and malicious entities running a significant portion of the tor nodes.[2]

DNSSEC

By securing current DNS records with cryptographic signatures, DNSSEC establishes a secure domain name system. The typical record types A, AAAA, MX, CNAME, etc. are stored in DNS name servers alongside these digital signatures. You may determine whether a requested DNS record originates from its authoritative name server and wasn't changed enroute by looking at its associated signature, as opposed to a phone record inserted during a man-in-the-middle attack.

To facilitate signature validation, DNSSEC adds a few new DNS record types:

- **RRSIG** - Contains a cryptographic signature
- **DNSKEY** - Contains a public signing key
- **DS** - Contains the hash of a DNSKEY record
- **NSEC** and **NSEC3** - For explicit denial-of-existence of a DNS record
- **CDNSKEY** and **CDS** - For a child zone requesting updates to DS record(s) in the parent zone.

RRset

The first step in protecting a zone with DNSSEC is to compile all of the records of the same kind into a resource record set (RRset). As opposed to individual DNS records, the entire RRset is digitally signed. Naturally, this also implies that you must request and validate each AAAA record from a zone that has the same label, as opposed to only one.

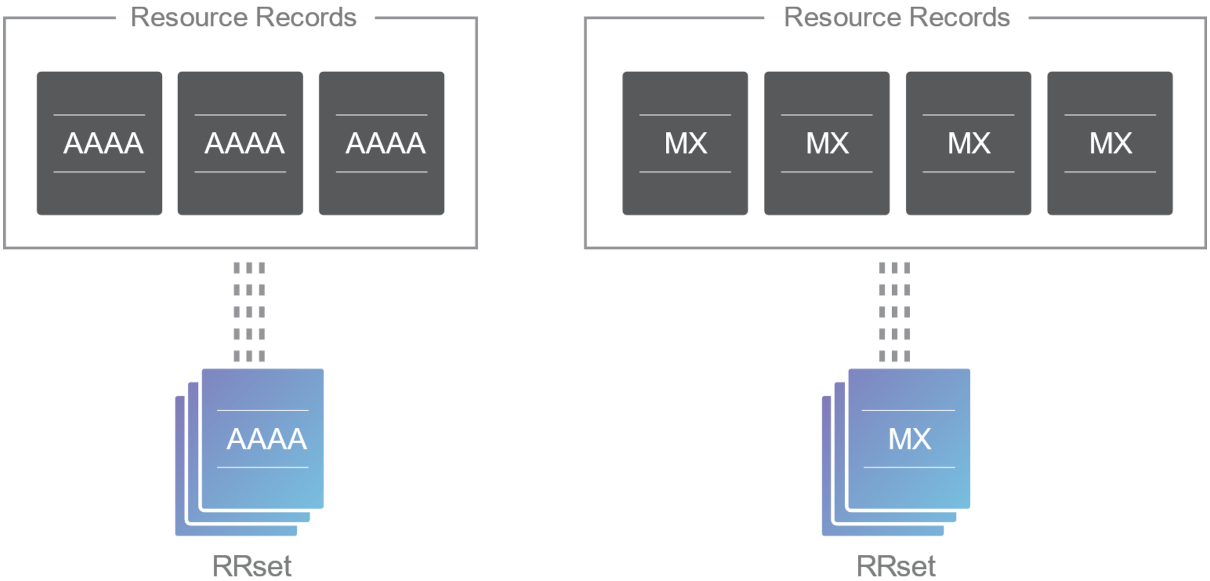

Figure 4

Zone-Signing Keys

In DNSSEC, each zone has a zone-signing key pair (ZSK), of which the private half digitally signs each RRset within the zone and the public half authenticates the signature. Using the private ZSK, a zone operator generates digital signatures for each RRset and saves them as RRSIG entries in their name server to enable DNSSEC.

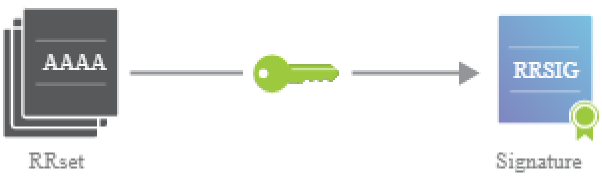

Figure 5

However, unless DNS resolvers have a way to validate the signatures, these RRSIG records are useless. The zone operator must additionally make their public ZSK accessible by including it in a DNSKEY record on their name server.

The name server also provides the corresponding RRSIG in response to a DNSSEC resolver's request for a certain record type, like AAAA. The DNSKEY record with the public ZSK can then be retrieved by the resolver from the name server. The response can be validated by the RRset, RRSIG, and public ZSK together.

If we trust the zone-signing key in the DNSKEY record, we can trust all the records in the zone. But, what if the zone-signing key was compromised? We need a way to validate the public ZSK

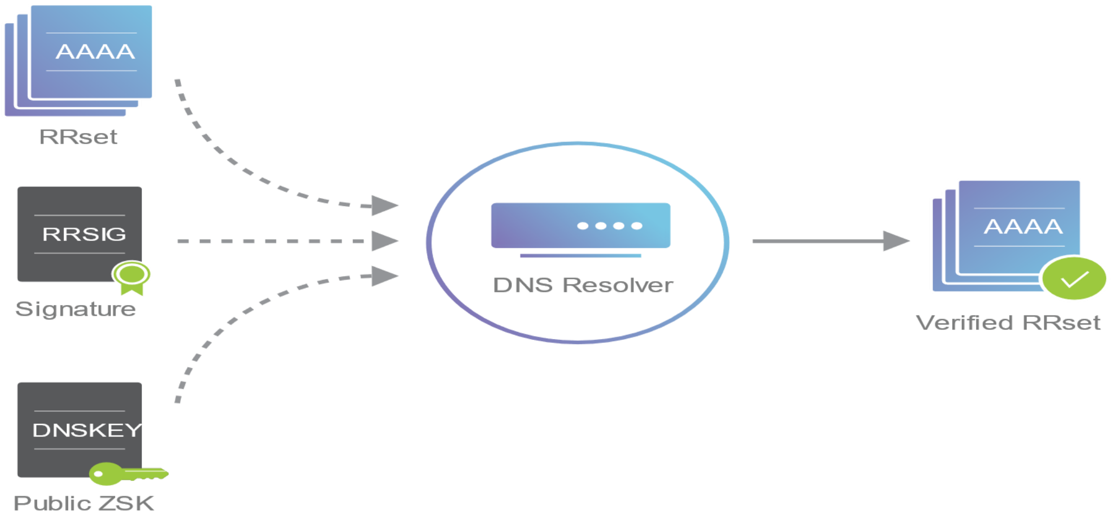

Figure 6

Key-Signing Keys

DNSSEC name servers have a key-signing key in addition to a zone-signing key (KSK). The ZSK protected the rest of our RRsets in the previous step, and the KSK checks the DNSKEY record in the exact same way. In doing so, it generates an RRSIG for the DNSKEY and signs the public ZSK (which is kept in a DNSKEY record).

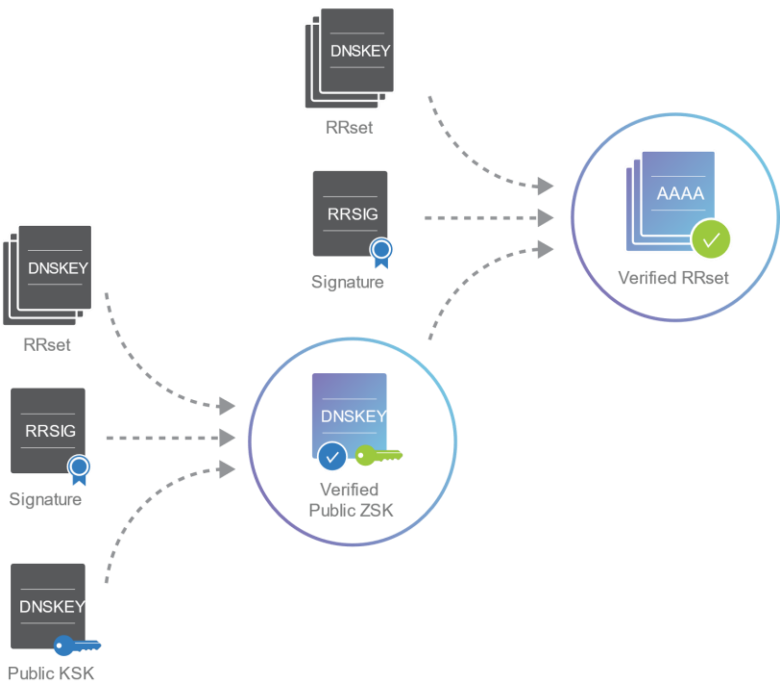

Figure 7

To lessen the strain on the server, certain DNSKEY RRset and related RRSIG entries are cached. Since it's impossible to replace a KSK that's been hacked or is too old, we utilise separate zone-signing keys and key-signing keys. On the other hand, it's lot simpler to change the ZSK. As a result, we may utilise a smaller ZSK without jeopardising the server's security and reduce the amount of data the server must send with each answer.

Delegation Signer Records

A delegation signer (DS) record is a new feature of DNSSEC that enables trust to be transferred from a parent zone to a child zone. The DNSKEY record containing the public KSK is hashed by a zone operator and given to the parent zone for publication as a DS record. Any KSK modification necessitates a corresponding DS record modification in the parent zone. A multi-step procedure called changing the DS record has the potential to break the zone if done incorrectly. The parent must add the new DS record first, then they must wait until the original DS record's TTL has passed before deleting it. This is the reason zone-signing keys are significantly simpler to replace than key-signing keys.

The Chain of Trust

The DS record has a corresponding RRSIG in the parent because it is signed similarly to other RRsets. Up until we reach the parent's public KSK, the entire validation process is repeated. We need to check that parent's DS record to confirm it, and so on as we move up the chain of trust.

A key component of DNSSEC is the capability to establish trust between parent and child zones. We cannot rely on the records we are requesting if any link in the chain is compromised because a man-in-the-middle attacker could change the records and point us in any direction.

Example for Chain of Trust

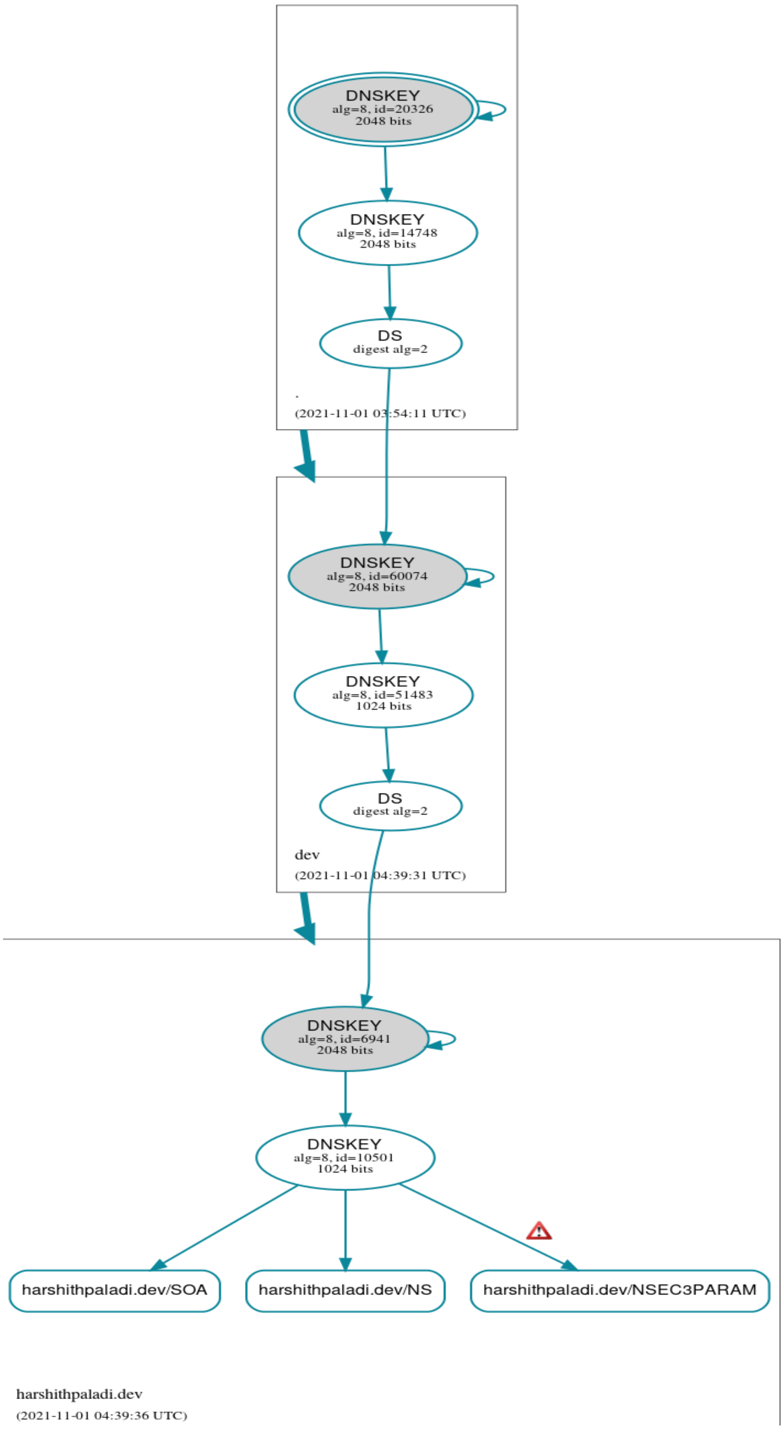

Figure 8

SOURCE CODE AND OUTPUT PICTURES

UNBOUND (Recursive Resolver)

==

Unbound.conf

Unbound configuration file for Debian.

See the unbound.conf(5) man page.

See /usr/share/doc/unbound/examples/unbound.conf for a commented reference config file.

The following line includes additional configuration files from the

/etc/unbound/unbound.conf.d directory.

include: "/etc/unbound/unbound.conf.d/*.conf"

===

qname-minimisation.conf

```
server:
    # Send minimum amount of information to upstream servers to enhance
    # privacy. Only sends minimum required labels of the QNAME and sets
    # QTYPE to NS when possible.
    # See RFC 7816 "DNS Query Name Minimisation to Improve Privacy" for
    # details.
    qname-minimisation: yes
```

===

root-auto-trust-anchor-file.conf

```
server:
    # The following line will configure unbound to perform cryptographic
    # DNSSEC validation using the root trust anchor.
    auto-trust-anchor-file: "/var/lib/unbound/root.key"
```

===

Recursive.conf

```
server:
```

```
# If no logfile is specified, syslog is used
# Logging
logfile: "/var/log/unbound/unbound.log"
verbosity: 5
use-syslog: no
log-time-ascii: yes
log-queries: yes
log-replies: yes
log-tag-queryreply: yes
log-servfail: yes
# Interface Info
interface: 127.0.0.1        # Server listening at 127.0.0.1
port: 53                    # Server listening on Port 53
do-ip4: yes
do-udp: yes                 # Send Responses in UDP
do-tcp: yes                 # Send Responses in TCP
# May be set to yes if you have IPv6 connectivity
do-ip6: no                  # Since no IPv6 connectivity, currently set as no
# Leave this to no unless you have native IPv6. With 6to4 and
# Teredo tunnels your web browser should favor IPv4 for the same reasons
prefer-ip6: no
# Use this only when you downloaded the list of primary root servers
# If you use the default dns-root-data package, unbound will find it automatically
#root-hints: "/var/lib/unbound/root.hints"
```

Trust glue only if it is within the server's authority

harden-glue: yes

Require DNSSEC data for trust-anchored zones, if such data is absent, the zone becomes #BOGUS

harden-dnssec-stripped: yes

Harden against algorithm downgrade when multiple algorithms are advertised in the DS # record.

harden-algo-downgrade: yes

Don't use Capitalization randomization as it known to cause DNSSEC issues sometimes

use-caps-for-id: no

Reduce EDNS reassembly buffer size.

Suggested by the unbound man page to reduce fragmentation reassembly problems

edns-buffer-size: 1472

Rotates RRSet order in response (the pseudo-random number is taken from Ensure privacy of local IP ranges the query ID, for speed and thread safety).

private-address: 192.168.0.0/16

rrset-roundrobin: yes

Time to live minimum for RRsets and messages in the cache. If the minimum

kicks in, the data is cached for longer than the domain owner intended,

and thus less queries are made to look up the data. Zero makes sure the

data in the cache is as the domain owner intended, higher values,

especially more than an hour or so, can lead to trouble as the data in

the cache does not match up with the actual data anymore

cache-min-ttl: 600

#cache-max-ttl: 86400

Have unbound attempt to serve old responses from cache with a TTL of 0 in

the response without waiting for the actual resolution to finish. The

actual resolution answer ends up in the cache later on.

serve-expired: yes

Limit serving of expired responses to configured seconds after expiration. 0 disables the limit.

This option only applies when serve-expired is enabled. A suggested value per RFC 8767 is between 86400 (1 day) and 259200 (3 days). The default is 0.

serve-expired-ttl: 86400 # one day, in second

Fetch the DNSKEYs earlier in the validation process, when a DS record is

encountered. This lowers the latency of requests at the expense of little more CPU usage.

prefetch-key: yes

Hides the id.server and hostname.bind queries, version.server and version.bind queries.

hide-identity: yes

hide-version: yes

Perform prefetching of close to expired message cache entries

This only applies to domains that have been frequently queried

prefetch: yes

One thread should be sufficient, can be increased on beefy machines. In reality for most users running on small networks or on a single machine

it should be unnecessary to seek performance enhancement by increasing num-threads above 1.

num-threads: 1

Ensure kernel buffer is large enough to not lose messages in traffic spikes

so-rcvbuf: 2m

```
    # Ensure privacy of local IP ranges
    private-address: 192.168.0.0/16
    private-address: 169.254.0.0/16
    private-address: 172.16.0.0/12
    private-address: 10.0.0.0/8
    private-address: fd00::/8
    private-address: fe80::/10
    # Remote control of unbound
remote-control:
        control-enable: yes
        # unbound server key file.
        server-key-file: "/etc/unbound/unbound_server.key"
        # unbound server certificate file.
            server-cert-file: "/etc/unbound/unbound_server.pem"
            # unbound-control key file.
            control-key-file: "/etc/unbound/unbound_control.key"
            # unbound-control certificate file.
            control-cert-file: "/etc/unbound/unbound_control.pem"
```

PYTHON CODE

```
=========================================================================
import logging
import argparse
import dns.message
import dns.name
```

```
import dns.query
import dns.rdata
import dns.rdataclass
import dns.rdatatype
from dns.exception import DNSException, Timeout
# Root Servers IP addresses
IP_ROOT_SERVERS = (
    # IP Address        # Name of the Root Servers
    "198.41.0.4",       # a.root-servers.net
    "199.9.14.201",     # b.root-servers.net
    "192.33.4.12",      # c.root-servers.net
    "199.7.91.13",      # d.root-servers.net
    "192.203.230.10",   # e.root-servers.net
    "192.5.5.241",      # f.root-servers.net
    "192.112.36.4",     # g.root-servers.net
    "198.97.190.53",    # h.root-servers.net
    "192.36.148.17",    # i.root-servers.net
    "192.58.128.30",    # j.root-servers.net
    "193.0.14.129",     # k.root-servers.net
    "199.7.83.42",      # l.root-servers.net
    "202.12.27.33",     # m.root-servers.net
)
FORMATS = (
    ("CNAME", "{alias} -> alias -> {name}"),
```

```
    ("A", "{name} -> IPv4 address -> {address}"),
    ("AAAA", "{name} -> IPv6 address -> {address}"),
    ("MX", "{name} -> mail by -> #{preference} {exchange}"),
)
Count = 0
def Results_Collect_DNS(name: str, Dns_cache: dict) -> dict:
    """
    Function parses final answers into the proper data structure that
    print_results requires.
    """
    Responses_Full = {}
    Domain_Name = dns.name.from_text(name)
    # Query A records
    response = Dns_lookup(Domain_Name, dns.rdatatype.A, Dns_cache)
    A = []
    for answers in response.answer:
        A_Rec = answers.name
        for answer in answers:
            if answer.rdtype == 1:  # A record
                A.append({"name": A_Rec, "address": str(answer)})
    # Query AAAA records
    response = Dns_lookup(Domain_Name, dns.rdatatype.AAAA, Dns_cache)
    AAAA = []
    for answers in response.answer:
```

```
    AAAA_Rec = answers.name
    for answer in answers:
        if answer.rdtype == 28:  # AAAA record
            AAAA.append({"name": AAAA_Rec, "address": str(answer)})
# Query CNAME records
response = Dns_lookup(Domain_Name, dns.rdatatype.CNAME, Dns_cache)
CNAME = []
for answers in response.answer:
    for answer in answers:
        CNAME.append({"name": answer, "alias": name})
# Query MX records
response = Dns_lookup(Domain_Name, dns.rdatatype.MX, Dns_cache)
MX = []
for answers in response.answer:
    mx_name = answers.name
    for answer in answers:
        if answer.rdtype == 15:  # MX record
            MX.append(
                {
                    "name": mx_name,
                    "preference": answer.preference,
                    "exchange": str(answer.exchange),
                }
            )
```

```
    Responses_Full["CNAME"] = CNAME
    Responses_Full["A"] = A
    Responses_Full["AAAA"] = AAAA
    Responses_Full["MX"] = MX
    Dns_cache.get("response_cache")[name] = Responses_Full
    return Responses_Full
def Recurse_Look(
    Domain_Name: dns.name.Name, qtype: dns.rdata.Rdata, ip_, resolved, Dns_cache: dict
) -> dns.message.Message:
    """
    This function uses a recursive resolver to find the relevant answer to the
    query.
    """
    global Count
    Count += 1
    outbound_query = dns.message.make_query(Domain_Name, qtype)
    try:
        response = dns.query.udp(outbound_query, ip_, 3)
        if response.answer:
            resolved = True
            return response, resolved
        elif response.additional:
            if response.authority:
```

```
            update_cache(response, Dns_cache)
        response, resolved = lookup_additional(
            response, Domain_Name, qtype, resolved, Dns_cache
        )
    elif response.authority and not resolved:
        response, resolved = lookup_authority(
            response, Domain_Name, qtype, resolved, Dns_cache
        )
    return response, resolved
  except Timeout:
    return dns.message.Message(), False
  except DNSException:
    return dns.message.Message(), False
def Dns_lookup(
  Domain_Name: dns.name.Name, qtype: dns.rdata.Rdata, Dns_cache: dict
) -> dns.message.Message:
  """
  Recursive resolver has been used by a function to get the response for the
  query.
  """
  incre = 0
  Resolved = False
  while incre < len(IP_ROOT_SERVERS):
    get_Ip_cache = ""
```

```
Name_Find = str(Domain_Name)
next_dot = str(Domain_Name).find(".")
while not get_Ip_cache and next_dot > -1:
    get_Ip_cache = Dns_cache.get(Name_Find)
    Name_Find = str(Name_Find)[next_dot + 1 :]
    next_dot = Name_Find.find(".")
if get_Ip_cache:
    ip_ = get_Ip_cache
    logging.debug("======== Found in cache =======\n")
else:
    ip_ = IP_ROOT_SERVERS[incre]

try:
    response, Resolved = Recurse_Look(
        Domain_Name, qtype, ip_, Resolved, Dns_cache
    )
    if response.answer:
        answer_type = response.answer[0].rdtype
        if qtype != dns.rdatatype.CNAME and answer_type == dns.rdatatype.CNAME:
            Domain_Name = dns.name.from_text(str(response.answer[0][0]))
            Resolved = False
            logging.debug(
                "--------- LOOKUP CNAME ----------- %s \n %s",
                Domain_Name,
```

```
                response.answer[0],
            )
            response = Dns_lookup(Domain_Name, qtype, Dns_cache)
        return response
    elif (
        response.authority and response.authority[0].rdtype == dns.rdatatype.SOA
    ):
        break
    else:
        incre += 1
  except Timeout:
    incre += 1
  except DNSException:
    incre += 1
return response


def update_cache(response: dns.message.Message, Dns_cache):
    """
    Function updates the cache latest results
    """
    domain_name = response.authority[0].to_text().split(" ")[0]


    A_Records = []
    rrsets = response.additional
```

```
    for rrset in rrsets:
        for rr_ in rrset:
            if rr_.rdtype == dns.rdatatype.A:
                A_Records.append(str(rr_))
                Dns_cache[domain_name] = str(rr_)
def lookup_additional(
    response,
    Domain_Name: dns.name.Name,
    qtype: dns.rdata.Rdata,
    resolved,
    Dns_cache: dict,
):
    """
    Function recursively finds additional data
    """
    rrsets = response.additional
    for rrset in rrsets:
        for rr_ in rrset:
            if rr_.rdtype == dns.rdatatype.A:
                response, resolved = Recurse_Look(
                    Domain_Name, qtype, str(rr_), resolved, Dns_cache
                )
            if resolved:
                break
```

```
        if resolved:
            break
    return response, resolved
def lookup_authority(
    response,
    Domain_Name: dns.name.Name,
    qtype: dns.rdata.Rdata,
    resolved,
    Dns_cache: dict,
):
    """
    Function recursively finds authority
    """
    rrsets = response.authority
    ns_ip = ""
    for rrset in rrsets:
        for rr_ in rrset:
            if rr_.rdtype == dns.rdatatype.NS:
                ns_ip = Dns_cache.get(str(rr_))
                if not ns_ip:
                    ns_arecords = Dns_lookup(str(rr_), dns.rdatatype.A, Dns_cache)
                    ns_ip = str(ns_arecords.answer[0][0])
                    Dns_cache[str(rr_)] = ns_ip
                response, resolved = Recurse_Look(
```

```
                Domain_Name, qtype, ns_ip, resolved, Dns_cache
            )
        elif rr_.rdtype == dns.rdatatype.SOA:
            resolved = True
            break
    if resolved:
        break
    return response, resolved
def print_results(results: dict) -> None:
    """
    Function takes results from Dns_lookup, prints to the screen.
    """
    for rtype, fmt_str in FORMATS:
        for result in results.get(rtype, []):
            print(fmt_str.format(**result))
def MainFn():
    global Count
    Dns_cache = {}
    Dns_cache["response_cache"] = {}
    Args_Parse = argparse.ArgumentParser()
    Args_Parse.add_argument("NAME", nargs="+", help="Domain name(s) to query")
    Args_Parse.add_argument(
        "-v", help="Increase the verbosity", action="store_true"
    )
```

```
    Proj_Args = Args_Parse.parse_args()

    for Domain in Proj_Args.NAME:

        Count = 0

        cache_result = Dns_cache.get("response_cache").get(Domain)

        if cache_result:

            print_results(cache_result)

        else:

            print_results(Results_Collect_DNS(Domain, Dns_cache))

        #logging.debug("Count %s", Count)

if __name__ == "__main__":

    #logging.basicConfig(level=logging.DEBUG)
```

PYTHON GUI CODE

```
import PySimpleGUI as sg

import subprocess

import sys

def main():

    sg.theme('DarkAmber')   # Add a touch of color

    # All the stuff inside your window.

    layout = [  [sg.Text('Enter Domain Name to resolve : '), sg.InputText()],

                [sg.Button('Ok'), sg.Button('Cancel')],

                [sg.Output(size=(80,20))]]

    # Create the Window

    window = sg.Window('DNS Lookup', layout)
```

```
    # Event Loop to process "events" and get the "values" of the inputs
    while True:
        event, values = window.read()
        if event == sg.WIN_CLOSED or event == 'Cancel': # if user closes window or clicks cancel
            break
        print('Entered Domain Name : ', values[0])
        runCommand(["python","python.py",values[0]])
    window.close()
def runCommand(cmd, timeout=None, window=None):
    p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    output = ''
    for line in p.stdout:
        line = line.decode(errors='replace' if (sys.version_info) < (3, 5) else 'backslashreplace').rstrip()
        output += line
        print(line)
        window.Refresh() if window else None        # yes, a 1-line if, so shoot me
    retval = p.wait(timeout)
    return (retval, output)                         # also return the output just for fun
if __name__ == '__main__':
    main()
```

===

OUTPUT PICTURES

UNBOUND

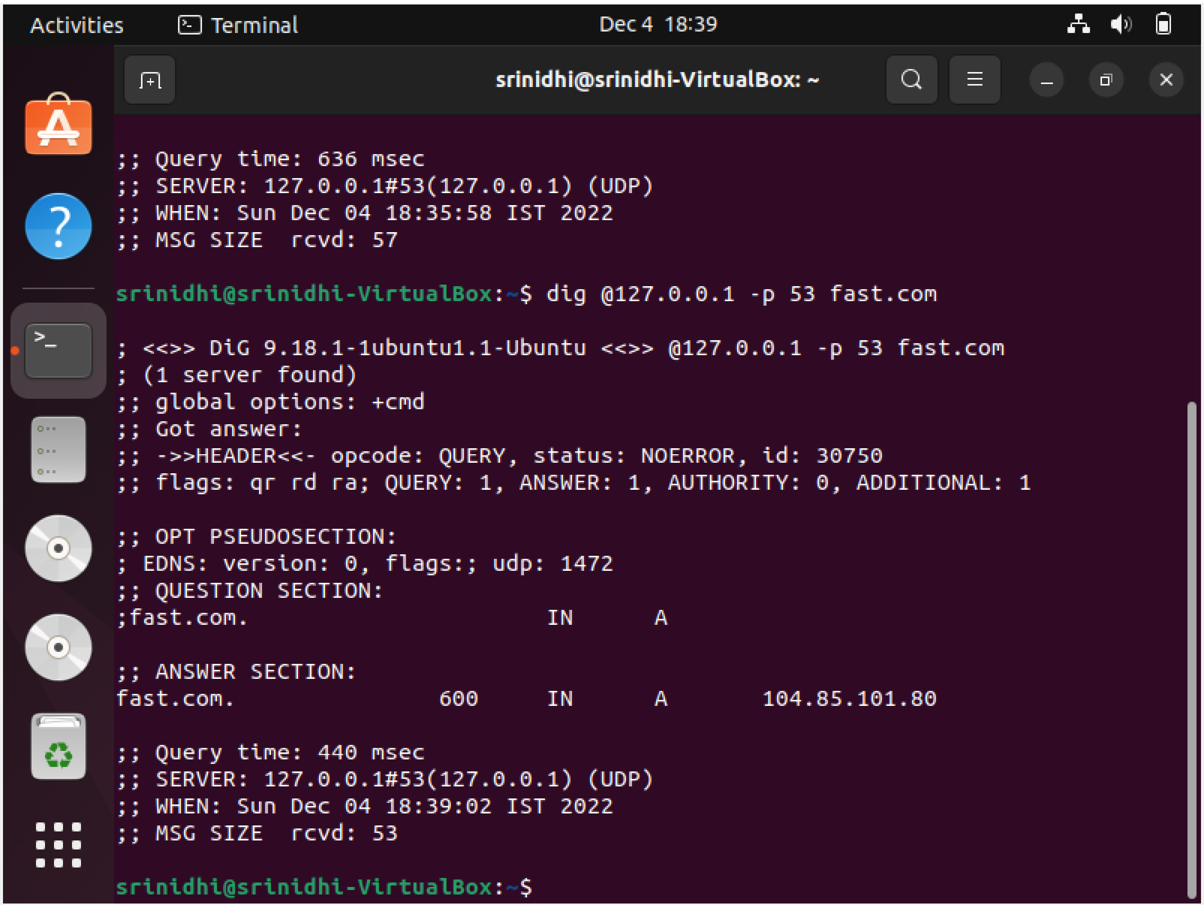

Figure 8

This is output pic from a machine that's not behind a NAT.

Cache Responses

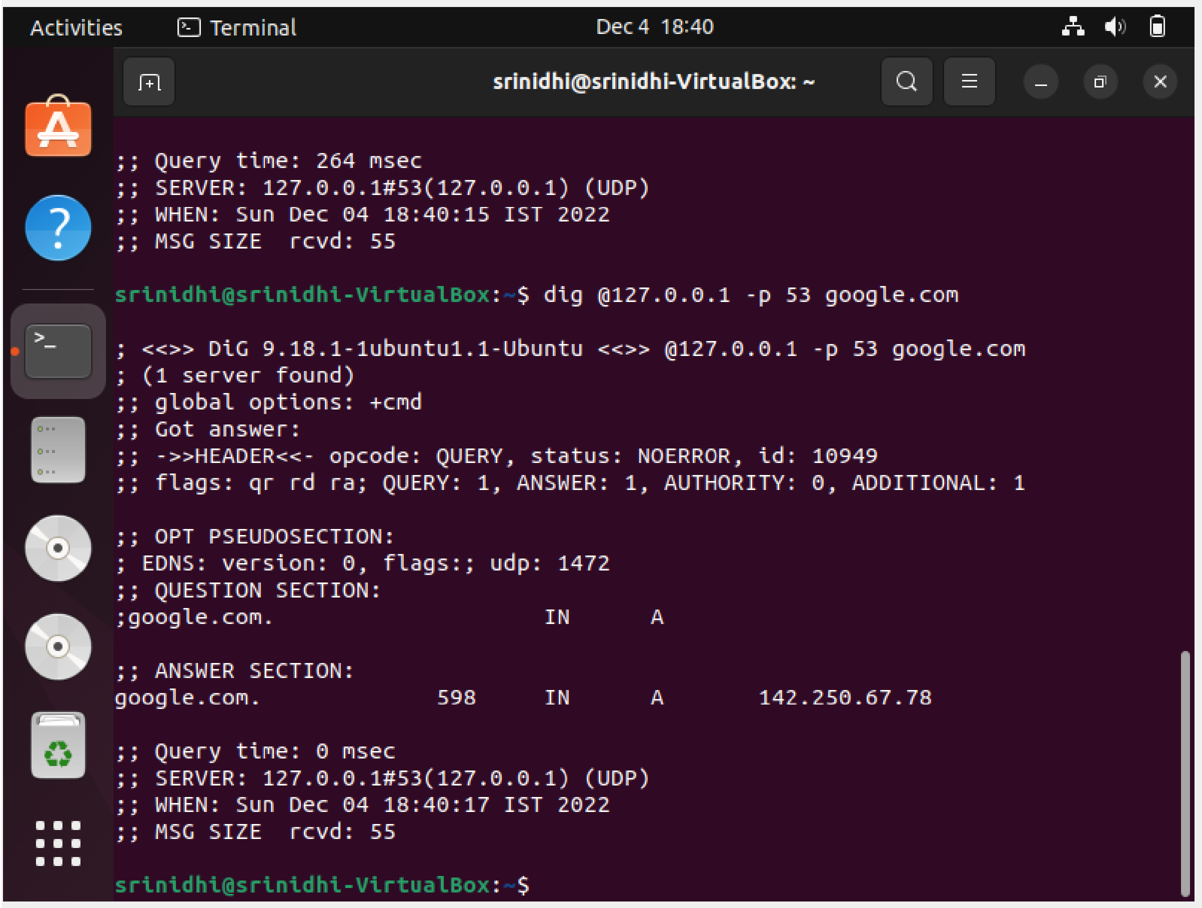

Figure 10

Here, Since the resolver already queried for google.com, it's DNS response is stored in the cache which is transmitted to the client without any queries. The value shown here is 0 ms, which is very fast.

In Unbound, DNSSEC validation is turned on and necessary security configurations are made.

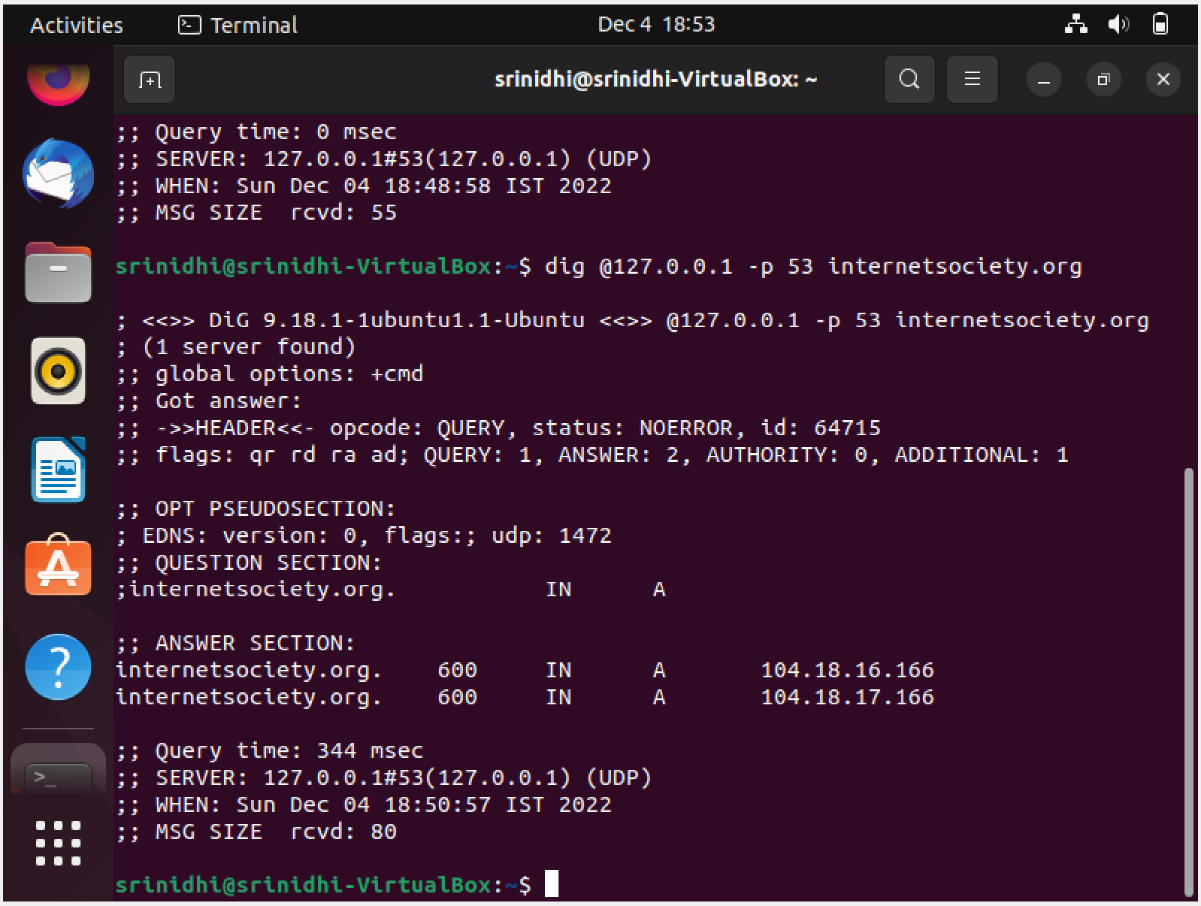

Figure 11

This site, internetsociety.org is configured with DNSSEC, so the DNS lookup is normal. Unbound is configured to strictly verify the signature of the DNS response and validate it. If there is no signature in the response, it treats the zone as insecure and responds to the client with the responses it got from the authoritative DNS server.

But if the signature is invalid, then the recursive resolver declares the response as BOGUS and doesn't reply with DNS responseresolver got the IP address from the authoritative nameserver but it responds with SERVFAIL indicating an issue with the signature.

PYTHON CODE

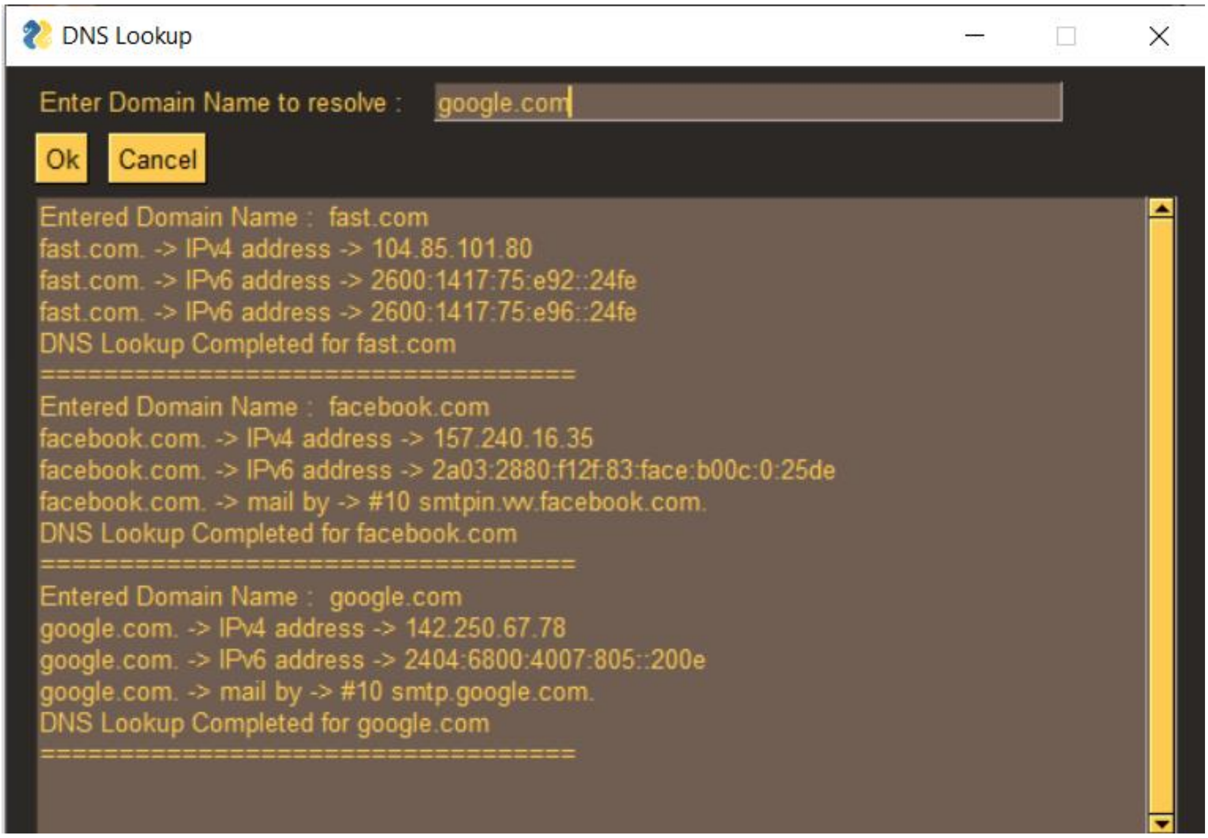

Figure 12

```
PS C:\Users\Harsha Vardhan\OneDrive\Desktop\CN Project\Python-Codes> python python.py google.com
google.com. -> IPv4 address -> 142.250.67.78
google.com. -> IPv6 address -> 2404:6800:4007:805::200e
google.com. -> mail by -> #10 smtp.google.com.
DNS Lookup Completed for google.com
==================================
PS C:\Users\Harsha Vardhan\OneDrive\Desktop\CN Project\Python-Codes> python python.py lalitha.com
lalitha.com. -> IPv4 address -> 15.197.142.173
lalitha.com. -> IPv4 address -> 3.33.152.147
DNS Lookup Completed for lalitha.com
==================================
PS C:\Users\Harsha Vardhan\OneDrive\Desktop\CN Project\Python-Codes> python python.py internetsociety.org
internetsociety.org. -> IPv4 address -> 104.18.16.166
internetsociety.org. -> IPv4 address -> 104.18.17.166
internetsociety.org. -> IPv6 address -> 2606:4700::6812:10a6
internetsociety.org. -> IPv6 address -> 2606:4700::6812:11a6
internetsociety.org. -> mail by -> #0 internetsociety-org.mail.protection.outlook.com.
DNS Lookup Completed for internetsociety.org
==================================
PS C:\Users\Harsha Vardhan\OneDrive\Desktop\CN Project\Python-Codes> python python.py worldwoo.com
worldwoo.com -> alias -> traff-1.hugedomains.com.
DNS Lookup Completed for worldwoo.com
==================================
PS C:\Users\Harsha Vardhan\OneDrive\Desktop\CN Project\Python-Codes> python python.py cloudfare.com
cloudfare.com. -> IPv4 address -> 104.21.77.216
cloudfare.com. -> IPv4 address -> 172.67.211.231
cloudfare.com. -> IPv6 address -> 2606:4700:3034::ac43:d3e7
cloudfare.com. -> IPv6 address -> 2606:4700:3031::6815:4dd8
cloudfare.com. -> mail by -> #20 mailstream-central.mxrecord.mx.
cloudfare.com. -> mail by -> #10 mailstream-west.mxrecord.io.
cloudfare.com. -> mail by -> #10 mailstream-east.mxrecord.io.
cloudfare.com. -> mail by -> #5 mailstream-canary.mxrecord.io.
DNS Lookup Completed for cloudfare.com
==================================
PS C:\Users\Harsha Vardhan\OneDrive\Desktop\CN Project\Python-Codes>
```

Figure 13

UNBOUND Configurations for Speed and Security

- harden-glue: yes, harden-dnssec-stripped: yes, harden-algo-downgrade: yes
  These are configured as yes because we want our resolver to strictly check the signature of the DNS response and prevent algorithm downgrade attacks etc.
- qname-minimisation: yes
  Here, QNAME minimization follows the principle of [RFC6973]: the less data you send out, the fewer privacy problems you have. The idea is to minimize the amount of data sent from the DNS resolver to the authoritative name server. Suppose for querying www.example.com, sending "What are the NS records for .com?" would have been sufficient (since it will be the answer from the root anyway).
- cache-min-ttl: 600, serve-expired: yes, serve-expired-ttl: 86400
  Here, we are overriding the TTL values of the DNS responses and setting it to minimum 600 seconds so that we need not query for the IP address of that domain
  Since IP addresses of the domains do not change very often. The additional queries will consume more resources on both the recursive resolver and the authoritative resolver sides. Also, usually the cache is removed after the TTL reaches 0 but here we are serving the expired values without waiting for the actual resolution to finish. The actual resolution answer ends up in the cache later on. With these settings, after running the server for a while, the response times from the server to the client will decrease drastically, hence providing the fast DNS resolution times with minimal risk.
- prefetch: yes, prefetch-key: yes, hide-identity: yes, hide-version: yes
  Here, we are prefetching the close to expired message cache entries which are frequently queried i.e., during the last 10% of TTL if a query comes for that cache entry, that entry will be fetched before any query comes for that entry, which optimizes the DNS query and response times. Also, hiding the identity of the server helps in protecting the server against attacks against specific vulnerabilities found in particular version.

CONCLUSIONS

- DNS is primarily unencrypted and usually carried over UDP port 53.
- DNS recursive resolvers plays an important role in the internet infrastructure. By Securing and optimizing the resolvers, it ensures in increasing the speed, privacy and security for all of the internet.
- These latest protocols come at the expense of traditional security measures being rendered useless in corporate networks and in censor states. So they can blocked by censors and corporate offices.
- More advancements in this field is necessary since it is one of the backbones of the internet.

FUTURE PLANS

- An advancement in the DNS-over-HTTPS is recently made called Oblivious DoH. ODoH is an emerging protocol being developed at the IETF. ODoH works by adding a layer of public key encryption, as well as a network proxy between clients and DoH servers. The combination of these two added elements guarantees that only the user has access to both the DNS messages and their own IP address at the same time.

REFERENCES

- **Michael Dooley; Timothy Rooney, "Introduction to the Domain Name System (DNS)," in DNS Security Management, IEEE, 2017, pp.17-29, Doi: 10.1002/9781119328292.ch2.**
- **Nguyen Phong Hoang, Arian Akhavan Niaki, Nikita Borisov, Phillipa Gill, and Michalis Polychronakis. 2020. Assessing the Privacy Benefits of Domain Name Encryption. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20). Association for Computing Machinery, New York, NY, USA, 290–304. DOI: https://doi.org/10.1145/3320269.3384728**

- https://www.rfc-editor.org/rfc/rfc7816#section-2 # QNAME MINIMIZATION
- https://www.cloudflare.com/en-in/learning/dns/dns-server-types/
- https://en.wikipedia.org/wiki/List_of_DNS_record_types
- https://en.wikipedia.org/wiki/Domain_Name_System
- https://www.cloudflare.com/en-in/learning/dns/what-is-dns/
- https://www.cloudflare.com/dns/dnssec/how-dnssec-works/
-

[1] https://root-servers.org.

[2] https://therecord.media/a-mysterious-threat-actor-is-running-hundreds-of-malicious-tor-relays/

CREDITS

[1] https://upload.wikimedia.org/wikipedia/commons/b/b1/Domain_name_space.svg

[2] https://www.cloudflare.com/img/learning/dns/what-is-dns/dns-lookup-diagram.png

[3] https://root-servers.org/

[4] https://www.iana.org/domains/root/servers

[5] https://upload.wikimedia.org/wikipedia/commons/thumb/a/a5/Example_of_an_iterative_DNS_resolver.svg/1920px-Example_of_an_iterative_DNS_resolver.svg.png

[6] https://www.cloudflare.com/img/products/ssl/diagram-rrsets.svg

[7] https://www.cloudflare.com/img/products/ssl/diagram-zone-signing-keys-1.svg

[8] https://www.cloudflare.com/img/products/ssl/diagram-zone-signing-keys-2.svg

[9] https://www.cloudflare.com/img/products/ssl/diagram-key-signing-keys-2.svg