GRoot: Proactive Verification of DNS Configurations

Siva Kesava Reddy Kakarla
UCLA
sivakesava@cs.ucla.edu

Ryan Beckett
Microsoft Research
ryan.beckett@microsoft.com

Behnaz Arzani
Microsoft Research
bearzani@microsoft.com

Todd Millstein
UCLA & Intentionet
todd@cs.ucla.edu

George Varghese
UCLA
varghese@cs.ucla.edu

ABSTRACT

The Domain Name System (DNS) plays a vital role in today's Internet but relies on complex distributed management of records. DNS misconfiguration related outages have rendered popular services like GitHub, HBO, LinkedIn, and Azure inaccessible for extended periods. This paper introduces GRoot, the first verifier that performs static analysis of DNS configuration files, enabling *proactive* and *exhaustive* checking for common DNS bugs; by contrast, existing solutions are *reactive* and *incomplete*. GRoot uses a new, fast verification algorithm based on generating and enumerating DNS query *equivalence classes*. GRoot symbolically executes the set of queries in each equivalence class to efficiently find (or prove the absence of) any bugs such as rewrite loops. To prove the correctness of our approach, we develop a formal semantic model of DNS resolution. Applied to the configuration files from a campus network with over a hundred thousand records, GRoot revealed 109 bugs within seconds. When applied to internal zone files consisting of over 3.5 million records from a large infrastructure service provider, GRoot revealed around 160k issues of blackholing, initiating a cleanup. Finally, on a synthetic dataset with over 65 million real records, we find GRoot can scale to networks with tens of millions of records.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; • **Networks** → *Application layer protocols*; *Network management*; • **Theory of computation** → *Logic and verification.*

KEYWORDS

DNS, Verification, Static Analysis, Formal Methods

ACM Reference Format:
Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRoot: Proactive Verification of DNS Configurations. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3387514.3405871

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08.
https://doi.org/10.1145/3387514.3405871

1 INTRODUCTION

The Domain Name System (DNS) is one of the largest distributed systems in use on the Internet today. It implements the fundamental service of name resolution: allowing users to connect to online services through user-friendly domain names in place of IP addresses, and enabling new applications ranging from service discovery [9] to load balancing [6, 23] to spam filtering [12, 13, 30, 31, 45].

To operate at global scale, DNS implements a hierarchical database managed by a distributed collection of organizations; each organization is responsible for maintaining a subset of the DNS infrastructure to provide name resolution for its portion of the DNS namespace (e.g., `sigcomm.org`). Operators within organizations manage the DNS through configuration *zone files*, which specify how DNS should respond to different types of user queries (e.g., whether to return an IP address, rewrite the user query, delegate the query to another domain, etc.). While some automation exists — for example in master-slave replication of servers — many data records are manually configured, especially at the interfaces between ownership boundaries. For example, customers of CDNs such as Akamai must manually configure their DNS records to point to CDN locations [1].

The scale and complexity of DNS makes its management difficult, and consequently, configuration errors that lead to performance or connectivity issues are widespread in practice [3, 16, 20, 24, 39, 41, 49, 51]. To make matters worse, configuration errors in DNS are often highly disruptive due to its global presence and residual caching effects from resolvers. For example, a 2014 misconfiguration at GitHub resulted in a loss of access to open source repositories [16] (possibly impacting SIGCOMM authors that year), and a misconfiguration for the JavaScript Node Package Manager (NPM) caused users to lose access to the service world-wide [15]. In both cases the outages persisted for hours as a result of DNS resolvers caching the misconfigured response. Perhaps the most severe of these outages was one caused by a recent DNS misconfiguration at Microsoft [44] that resulted in a global outage impacting all Azure customers for 2 hours. The error was caused by a management process necessitated by a migration, which resulted in an inconsistency among zone file replicas.

To prevent DNS-related outages, operators today rely on a mix of techniques such as monitoring [28, 50], testing [10, 22], linting [35] and manual review. While these approaches are often effective at identifying issues, most of them can only catch errors after they have already been introduced into a live system. For instance, solutions based on monitoring have this limitation and are further

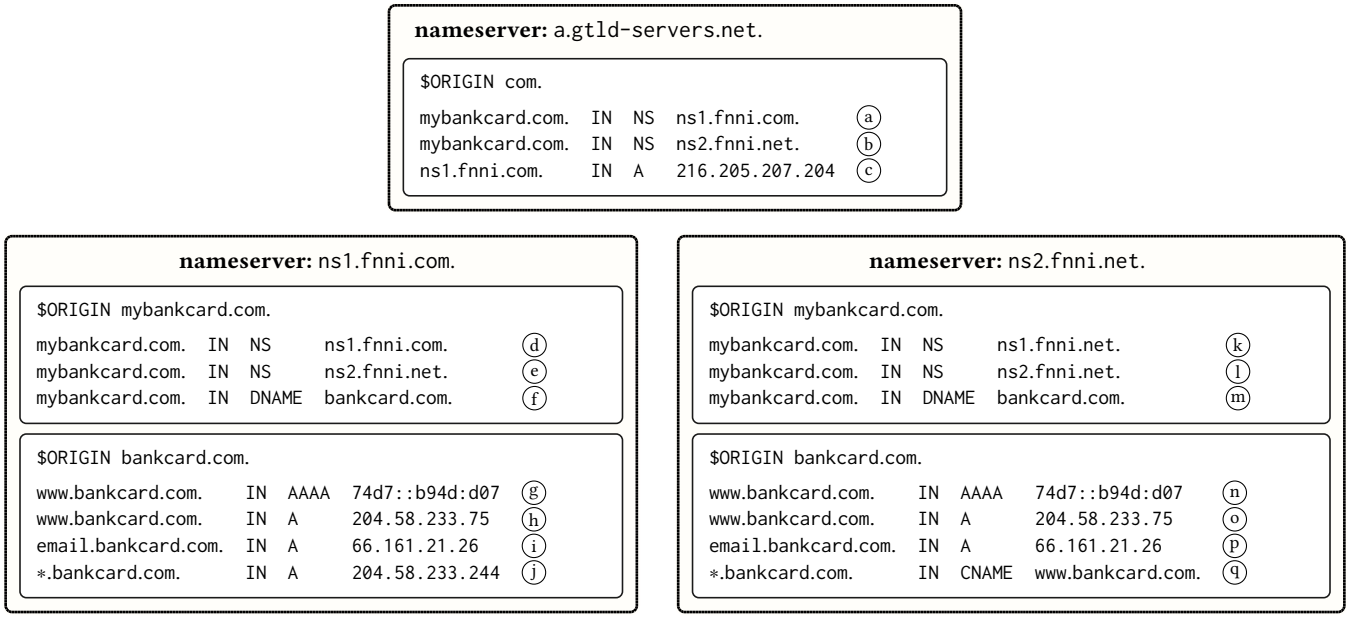

Figure 1: Example zone files for three nameservers: a.gtld-servers.net**,** ns1.fnni.com**, and** ns2.fnni.net**. The query** ⟨support.mybankcard.com, A⟩ **has two possible executions: one for records** $\langle a, f, j\rangle$ **and another for** $\langle b, m, q, o\rangle$**.**

complicated by deployment factors such as caching, which can delay the identification of a problem, and geo-replication, which can alter the nameserver used to resolve a query based on the client's geographic location. Further, none of these approaches can provide strong guarantees — the system may still have bugs even after successfully passing all of these checks.

To address the problem of DNS misconfiguration, we present GRoot, which, to the best of our knowledge, is the first verification tool for DNS configurations. Given the DNS zone files of an organization and a property Φ of interest, GRoot will either verify that Φ holds *for all possible DNS queries* or provide a counterexample.

While the number of possible DNS queries is huge, we observe that the number of distinct *behaviors* is much smaller and is a function of the DNS configuration files. Based on this insight, GRoot first performs an analysis of the DNS configuration to partition all possible queries into *equivalence classes* (ECs) each of which captures a distinct behavior. The key property of this partition is that two queries in the same EC resolve to the same set of possible answers (in general a query can have multiple possible answers due to nondeterminism inherent in the DNS resolution process) in the given DNS configuration. GRoot then performs a *symbolic execution* of each EC to produce its set of answers and check the given property Φ. Although existing standards [37, 38, 42] specify the behaviour of the DNS, these standards are informal and described in English. Therefore, as part of this work we present a mathematical formalization that allows for *automatically verifying* DNS configurations and detecting any misconfiguration. Our formal model of the existing DNS resolution standards is crucial for the efficient symbolic execution in GRoot.

We applied GRoot to the configuration files we obtained from a large campus network which has over a hundred thousand records, GRoot revealed 109 new bugs and completed in under 10 seconds. GRoot identified bugs in the network ranging from delegation inconsistencies to lame delegations to rewrite loops and others. When applied to internal zone files consisting of over 3.5 million records from a large infrastructure service provider, GRoot revealed around 160k issues of blackholing, which initiated a cleanup of the zone files. Finally, on a synthetic dataset that we created from over 65 million real DNS records [11] we found that GRoot can scale to networks with tens of millions of records spread across tens of thousands of zones.

To summarize, we make the following contributions:

- **A formal model of the DNS.** To the best of our knowledge, we present the first formal model of DNS that captures the semantics of both the authoritative and recursive systems.
- **A fast verification algorithm.** Using our formal model of the DNS, we describe, and prove the correctness of, a fast algorithm to generate equivalence classes of DNS queries. These equivalence classes enable GRoot to efficiently, and exhaustively, check the correctness of DNS zone files.
- **Evaluation on production configuration files.** We evaluate GRoot using data from **(1)** configurations obtained from a large campus network, **(2)** configurations obtained from a large infrastructure service provider, and **(3)** a synthetic dataset built from over 65 million Internet records, showing that GRoot is effective at finding bugs and verifying large configurations.

Ethics. Our formal model and tool GRoot could be used to prevent potential attacks against DNS infrastructure (e.g., input queries that result in the most work possible being performed) as one can check if there is any input query that can lead to an attack. On the flip side, if an attacker has access to the tool and the organization's zone files, they could also do the same. However, gaining access to an organization's internal zone files is inherently difficult.

Bug	Description
Delegation Inconsistency	The parent and child zone files do not have the same set of NS and A (glue) records for delegation
Lame Delegation	A name server that is authoritative for a zone does not provide authoritative answers
Missing Glue Records	The zone file is missing required "glue" A or AAAA records for nameservers in NS records
Non-Existent Domain for Service	DNS returns the NXDOMAIN answer for a known service (e.g., sigcomm.org)
Cyclic Zone Dependency	Resolving a query for zone Z_1 depends on Z_2, which depends on Z_1
Rewrite Loop	There exists a query that is rewritten in a loop $q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow \ldots \rightarrow q_1$
Query Exceeds Maximum Length	There exists a query q_1 is that is eventually rewritten to q_n which exceeds the max label or domain length
Answer Inconsistency	Different executions in DNS result in different answers
Zero Time To Live	There exists a query which will return a resource record with the TTL set to 0, which prevents caching
Rewrite Blackholing	There exists a query q_1 that is *eventually* rewritten to q_n which does not exist and DNS returns NXDOMAIN

Table 1: Sample subset of possible bugs. Several are taken from previous work [40] while we proposed the rest.

2 BACKGROUND AND MOTIVATION

Conceptually, DNS provides a mapping between "human-readable" domain names used to identify services and "machine-readable" IP addresses needed to locate those services. In this context, a domain name is a URL such as mybankcard.com. Domain names naturally form a partial order. For example, we say that mybankcard.com is the *parent* of support.mybankcard.com, and accordingly support.mybankcard.com is a *child* of mybankcard.com. We also consider support.mybankcard.com, mybankcard.com, and com to be **prefixes** of support.mybankcard.com (explained in §3). A DNS *query* contains a domain name, and a query is *resolved* by the DNS in order to produce say, a corresponding IP address. For example a DNS query for mybankcard.com may resolve to the IP 204.58.233.75.

Management of the DNS database is partitioned across multiple *nameservers*, which are maintained by different organizations and which are themselves referenced via domain names (e.g., ns1.fnni.com). For example, owners of domains such as google.com, microsoft.com, and netflix.com each manage their own DNS nameservers. A nameserver includes a collection of *zones*, each of which contains DNS *records* that provide information for resolving a particular domain name and possibly some of its children. In its simplest form, this information is simply the IP address corresponding to the domain name. However, as our example below shows, a wide variety of other types of information exist as well, including: start of authority data (SOA), text data (TXT), mail exchange server data (MX), record aliases (CNAME), domain redirections (DNAME), nameserver referrals (NS), reverse IP-to-domain mappings (PTR), and many others [37, 42].

When a user enters a URL (e.g., support.bankcard.com) into a browser, a DNS query is sent to the ISP's DNS *resolver*, which traverses the DNS database to resolve the query. It does so using a recursive process of querying nameservers, starting from a known *root* nameserver and continuing to other nameservers until the query is sent to a nameserver that has an *authoritative* answer for the query. We next illustrate this process with a small example and then use it to describe the challenges of proper DNS configuration.

2.1 An example of DNS resolution

To see how a DNS query gets resolved, consider the configuration zone files shown in Figure 1, which are based on real records we observed in practice (simplified and anonymized for presentation). There are five zone files spread across three different nameservers (a.gtld-servers.net, ns1.fnni.com, and ns2.fnni.net). Each nameserver serves one or more zones (e.g., mybankcard.com and bankcard.com), and is configured to hold a set of resource records in each zone. We depict each record with an accompanying label (e.g., ⓐ) and refer to those labels when discussing a record. Each record has a domain name "key" and a value, along with other information such as the record type. For instance, record ⓐ is for the domain name mybankcard.com and has type NS, which means the "value" refers to another nameserver (ns1.fnni.com).

Suppose a user issues a DNS query for the IP address of the domain name support.mybankcard.com. The query is represented as the tuple ⟨support.mybankcard.com, A⟩, where A represents the IPv4 record type. Assuming the answer is not already cached, the resolver will issue the query to a known default nameserver, for instance a.gtld-servers.net in this example. The nameserver is now responsible for answering this query, either by answering directly, or by referring the resolver to other nameservers.

To do so, the nameserver will lookup the closest matching records for the query (roughly speaking the records with the longest matching prefix). For support.mybankcard.com, this will be the NS records with domain name mybankcard.com { ⓐ, ⓑ }. The nameserver will respond with both records, which indicate the resolver should continue by asking another nameserver (ns1.fnni.com or ns2.fnni.net). In this particular case, the nameserver will also include ⓒ (glue record) in its response, the IPv4 address to reach ns1.fnni.com, according to the wider definition of *Bailiwick rule* [21] (a.gtld-servers.net includes the IPv4 records for the referred nameserver even if under a sibling domain (fnni.com)).

After receiving a response from a.gtld-servers.net, the resolver will then nondeterministically chose one of the two new nameservers to ask next. In practice, this decision is often influenced by heuristics such as the estimated RTT to the nameserver. Suppose the resolver chooses to query ns1.fnni.com next. The same query support.mybankcard.com is sent to the nameserver, which hosts two zones (mybankcard.com and bankcard.com). The nameserver will choose the closest matching zone (mybankcard.com) and then proceed as before. This time, the most relevant record is the DNAME record ⓕ. A DNAME record performs a query rewrite, in this case to redirect the user to bankcard.com. Specifically, ⓕ will rewrite the query prefix mybankcard.com to bankcard.com, yielding the new query support.bankcard.com.

The nameserver will now re-evaluate this new query since it has a configuration locally for the zone bankcard.com. This zone has IP records for the domains www.bankcard.com and

$$
\begin{array}{lrcll}
(1) & d_1 \simeq_j d_2 & \stackrel{\text{def}}{=} & (0 \le j \le \min(|d_1|, |d_2|)) \wedge (\forall i.\, 0 < i \le j \implies d_1[i] = d_2[i]) & \textit{domain prefix match} \\
(2) & \max_{\simeq}(d_1, d_2) & \stackrel{\text{def}}{=} & \max\left\{j \mid d_1 \simeq_j d_2\right\} & \textit{maximal prefix match} \\
(3) & d_1 \le d_2 & \stackrel{\text{def}}{=} & d_1 \simeq_{|d_1|} d_2 & \textit{domain partial order} \\
(4) & d_1 \in_* d_2 & \stackrel{\text{def}}{=} & (|d_2| \le |d_1|) \wedge (d_1 \simeq_{(|d_2|-1)} d_2) \wedge d_1[|d_2|] \ne * = d_2[|d_2|] & \textit{domain wildcard match} \\
(5) & \textsc{Wildcard}(r) & \stackrel{\text{def}}{=} & \mathrm{dn}(r)[|\mathrm{dn}(r)|] = * & \textit{is a wildcard record} \\
(6) & \textsc{Match}(r, q) & \stackrel{\text{def}}{=} & \mathrm{dn}(r) \le \mathrm{dn}(q) \vee \mathrm{dn}(q) \in_* \mathrm{dn}(r) & \textit{record matches query} \\
(7) & \textsc{Rank}(r, q, z) & \stackrel{\text{def}}{=} & \langle \mathcal{I}\ (\textsc{Match}(r, q)), \mathcal{I}\ (\mathrm{ty}(r) = \mathsf{NS} \wedge \mathrm{dn}(r) \ne \mathrm{dn}(z)), \max_{\simeq}(\mathrm{dn}(r), \mathrm{dn}(q)), \mathcal{I}\ (\textsc{Wildcard}(r)) \rangle & \textit{resource record rank} \\
(8) & r_1 <_{q,z} r_2 & \stackrel{\text{def}}{=} & \textsc{Rank}(r_1, q, z) \lhd \textsc{Rank}(r_2, q, z) & \textit{resource record order}
\end{array}
$$

Figure 2: Common DNS definitions, and notations.

`email.bankcard.com`, but not for `support.bankcard.com`. As such, the query will match the *wildcard* record (j). Wildcard records (with $*$) match domain names with a shared prefix that are not matched by other records [37]. Thus, the nameserver will return an answer with IP address `204.58.233.244`.

2.2 DNS configuration challenges

Authoring and maintaining correct DNS configurations is challenging for several reasons. First, the protocol is inherently nondeterministic. In the above example, if the resolver had chosen to send the query to the nameserver `ns2.fnni.net` instead of `ns1.fnni.com`, then after several steps DNS would match the query with the wildcard record (q). The `CNAME` record type (canonical name) performs a rewrite without preserving the query suffix, so the query becomes `www.bankcard.com` and finally matches record (o), which provides the IP address `204.58.233.75`, differing from the result above.

Second, as the example shows, the DNS protocol is intricate and subtle, involving multiple types of records and complex dependencies among these records due to behaviors such as query rewriting. Both `CNAME` and `DNAME` rewrites provide a level of indirection to allow efficient handling of change. For example, `DNAME` records can help when multiple subtrees of the DNS need to be the same. `CNAME` records are useful when users have to be redirected to the same information from different domains as in `example.com` and `www.example.com`. Though `DNAME` records are a bit rare, `CNAME` records are pervasive, and `CNAME` chains are used extensively by CDNs to accelerate the efficiency of content delivery [18, 43, 46].

Third, DNS is managed as a collection of distributed zone files, under the control of different organizations. Finally, all of these issues arise in the context of understanding a *single* DNS query, but operators must ensure that *all possible* queries behave as intended.

For all of these reasons, it is no surprise that configuration changes and operator mistakes are at the heart of many large-scale DNS outages in the past [16, 24, 39, 44, 49, 51]. Indeed, there are many ways in which DNS behavior can go wrong, in addition to nondeterministically returning different answers as shown above. For example, a configuration mistake might result in DNS returning `NX` (non-existent domain) for a popular service, which can result in a loss of connectivity, as was the case in the recent Azure outage [44]. As another example, a query might get stuck in a rewrite loop. Table 1 summarizes several common kinds of DNS misconfigurations. In §7 we demonstrate our tool GRoot's effectiveness in finding such errors in real-world DNS configurations.

3 FORMAL MODEL OF DNS

In order to exhaustively verify the behavior of the DNS, we must first formalize its behavior. In this section we provide a formal, mathematical semantics for DNS, including both nameserver lookup and recursive resolution. A key technical challenge in formalizing this model was to accurately capture the behavior of DNS in the presence of many complex features such as nondeterminism, wildcard records, referrals, different types of rewrites, and many other features, all of which interact in subtle ways. To our knowledge this is the first formal model of DNS, and as such we hope in future researchers can build on this model in order to precisely reason about the behavior of DNS.

3.1 Definitions and Notations

A domain name is a string identifier controlled by some administrative group that DNS associates with other underlying information, such as an IP address or mail record. We model a domain name as a sequence of zero or more strings, called labels. The domain name `foo.com` contains the labels `foo`, `com`, and an implicit empty label (ϵ) for the root domain. For clarity, we often write a domain name as a concatenated sequence of labels delimited by $\circ$ and terminated by the special symbol ϵ, which represents an empty string (e.g., `foo.com` is written as `foo` $\circ$ `com` $\circ$ ϵ). The sequence with only the empty domain name (ϵ) is called the root domain. A full domain name may not exceed either 253 characters in its textual representation, 127 labels in its length, or 63 characters for any individual label. Given a domain $d = l_k \circ \ldots \circ l_0$ where $l_0 = \epsilon$, we write $|d|$ to denote the index of the last label k, and we use the indexing notation $d[i]$ to select label l_i. We denote the set of valid domain names by the set: DOMAIN.

Figure 2 shows a number of definitions that we use to define the behavior of DNS. Specifically, we use the notation $d_1 \simeq_j d_2$ (1) to mean that domains d_1 and d_2 share a common prefix of j labels (not counting ϵ). For example, `foo` $\circ$ `com` $\circ$ $\epsilon \simeq_1$ `com` $\circ$ ϵ. To select the maximal j such that $d_1 \simeq_j d_2$, we write $\max_{\simeq}(d_1, d_2)$ (2). We use the definition of $\simeq_j$ to introduce a partial ordering among domain names (3) that orders them by longest match. In particular, $d_1 \le d_2$ iff d_1 is a prefix of d_2 ($d_1 \simeq_{|d_1|} d_2$). We also use the notation $d_1 \in_* d_2$ to mean that d_1 matches the wildcard domain d_2 (4).

Zones and resource records. A DNS zone $z \in$ ZONE is a set of resource records (ZONE = $\mathcal{P}$(RECORD)). We use the symbol $\mathcal{P}$(RECORD) here to represent the powerset of resource records. A zone is well-formed if it contains exactly one `SOA` (Start of Authority) record listing the domain name of the zone along with other administrative

Resource Record Set Lookup $\boxed{\mathcal{P}(\text{RECORD}) \times \text{QUERY} \times \text{ZONE} \to \text{ANSWER}}$

$$\text{RRLookup}(R,q,z) = \begin{cases} \text{ExactMatch}(R,q,z,T) & \text{dn}(R) = \text{dn}(q) \\ \text{WildcardMatch}(R,q,T) & \text{dn}(q) \in_* \text{dn}(R) \\ \text{Rewrite}(R,q) & \text{dn}(R) < \text{dn}(q),\ \texttt{DNAME} \in T \\ \text{Delegation}(R,z) & \text{dn}(R) < \text{dn}(q),\ \texttt{DNAME} \notin T,\ \texttt{NS} \in T,\ \texttt{SOA} \notin T \\ \langle \text{NX},\ \emptyset \rangle & \text{otherwise} \end{cases}$$
where $T = \{\text{ty}(r) \mid r \in R\}$

$$\text{ExactMatch}(R,q,z,T) = \begin{cases} \langle \text{Ans},\ \mathcal{T}(R,\text{ty}(q)) \rangle & \text{Authoritative}(T), \text{ty}(q) \in T \\ \langle \text{AnsQ},\ \langle R,\ \langle \text{ans}(r),\ \text{ty}(q) \rangle \rangle \rangle & \text{Authoritative}(T), \text{ty}(q) \notin T,\ \texttt{CNAME} \in T,\ R = \{r\} \\ \text{Delegation}(R,z) & \neg\text{Authoritative}(T),\ \texttt{NS} \in T \\ \langle \text{Ans},\ \emptyset \rangle & \text{otherwise} \end{cases}$$

$$\text{WildcardMatch}(R,q,T) = \begin{cases} \langle \text{Ans},\ \text{Syn}\,(\mathcal{T}(R,\text{ty}(q)),\ \text{dn}(q)) \rangle & \text{ty}(q) \in T \\ \langle \text{AnsQ},\ \langle \text{Syn}(R,\text{dn}(q)),\ \langle \text{ans}(r),\ \text{ty}(q) \rangle \rangle \rangle & \text{ty}(q) \notin T,\ \texttt{CNAME} \in T,\ R = \{r\} \\ \langle \text{Ans},\ \emptyset \rangle & \text{otherwise} \end{cases}$$

$$\mathcal{T}(R,\ t) = \{r \in R \mid \text{ty}(r) = t\}$$
$$\text{Rewrite}(R,q) = \langle \text{AnsQ},\ \text{DProc}\,(\mathcal{T}(R,\texttt{DNAME}), q) \rangle$$
$$\text{Authoritative}(T) = \texttt{NS} \notin T \vee \texttt{SOA} \in T$$
$$\text{Delegation}(R,z) = \langle \text{Ref}, \text{Glue}(\mathcal{T}(R,\texttt{NS}), z) \rangle$$
$$\text{DProc}(\{r\}, q) = \langle \{r\} \cup \{\langle \text{dn}(q), \texttt{CNAME}, \texttt{IN}, \text{ttl}(r), d, 1 \rangle\}, \langle d, \text{ty}(q) \rangle \rangle \quad \textbf{where } d = \text{dn}(q)\,[\text{dn}(r) \mapsto \text{ans}(r)]$$
$$\text{Glue}(R,z) = R \cup \{r \in z \mid \exists\, r' \in R.\, \text{ans}(r') = \text{dn}(r) \ \wedge\ \text{ty}(r) \in \{\texttt{A}, \texttt{AAAA}\}\}$$
$$\text{Syn}(R,d) = R \cup \{\langle d, t, \texttt{IN}, \tau, a, 1 \rangle \mid \exists\, d'.\ \langle d',\ t, \texttt{IN}, \tau, a, 0 \rangle \in R\}$$

Nameserver lookup for a query $\boxed{\mathcal{P}(\text{ZONE}) \times \text{QUERY} \to \text{ANSWER}}$

$$\text{ServerLookup}(Z,q) = \begin{cases} \text{ZoneLookup}(z,q) & \mathcal{N}(Z,q) = \{z\} \\ \langle \text{Refused}, \emptyset \rangle & \mathcal{N}(Z,q) = \emptyset \end{cases}$$

$$\mathcal{N}(Z,q) = \max_{\text{dn}} \{z \in Z \mid \text{dn}(z) \le \text{dn}(q)\}$$
$$\text{ZoneLookup}(z,q) = \text{RRLookup}(\{r \in \max_{<_{q,z}} z\}, q, z)$$

Figure 3: Authoritative DNS lookup semantics.

information. Appendix A §10 lists other conditions a well-formed zone has to satisfy. We write $\text{dn}(z)$ to mean the domain name for a zone z, which is stored in this SOA record.

We model a resource record $r \in \text{RECORD} = \langle d, t, c, \tau, a, b \rangle$ as a tuple with six components: **(1)** a domain name $d \in \text{DOMAIN}$, **(2)** a record type $t \in \text{TYPE} = \{\texttt{A}, \texttt{AAAA}, \texttt{MX}, \texttt{NS}, \texttt{DNAME}, \texttt{CNAME}, \texttt{SOA}, \ldots\} \cup \{\texttt{N}\}$ representing either the kind of data the record holds (e.g., AAAA for an IPv6 address) or the type N to represent empty data (we explicitly model empty non-terminals [32, 37] as resource records containing this type N), **(3)** the record class $c = \texttt{IN}$ for the internet, **(4)** the time-to-live value for the record $\tau \in \mathbb{N}$ that defines the number of seconds for which the record can be cached, **(5)** the answer $a \in \Sigma^*$ which gives the DNS result as a string, and finally **(6)** a boolean value b that marks whether a record was synthesized from another.

We write $\text{dn}(r)$ for the domain name of record r, $\text{ty}(r)$ for the type, $\text{class}(r)$ for the class, $\text{ttl}(r)$ for the TTL, $\text{ans}(r)$ for the record answer, and $\text{synth}(r)$ for whether the record was synthesized.

DNS queries. A DNS query $q = \langle d,\ t \rangle$ is a tuple containing a domain name $d \in \text{DOMAIN}$ and a query type $t \in \text{TYPE}$. A user that needs the IPv4 address might send a query $\langle \texttt{www.mybankcard.com}, \texttt{A} \rangle$ to ask for it. As with resource records, we write $\text{dn}(q)$ to mean the domain name of query q, and $\text{ty}(q)$ to mean the query type.

The remaining definitions (5) – (8) in Figure 2 are used to define the order in which DNS prioritizes resource records for a given query. Wildcard (5) determines if a record has a wildcard domain, and Match (6) determines if a record is relevant for a given query (i.e., a potential match). The Rank (7) function for a record, query, and zone returns a tuple of integer values; the indicator function ($\mathcal{I}$) returns 1 if the predicate is true and 0 otherwise. The Rank function then induces a strict partial order ($<_{q,z}$) on records (8) by comparing the resulting tuples lexicographically from left to right ($\lhd$). The ranking is over four values: **(1)** whether the record is a match for the query (note that there will always be at least one match, e.g., the SOA record for queries that are not refused by the server §3.3), **(2)** if there is a zone cut (i.e., an NS record for a subdomain), **(3)** the length of the match between record and query, **(4)** and finally whether the record is a wildcard as a tiebreaker.

DNS answers. We model a DNS answer $a = \langle x, y \rangle$ as a pair of a tag $x \in \{\text{Ans}, \text{AnsQ}, \text{Ref}, \text{NX}, \text{Refused}, \text{ServFail}\}$ indicating the type of answer (e.g., an answer Ans, a delegation Ref, a rewrite AnsQ, etc.), and data y, which is a set of resource records R holding pertinent information when $x \neq \text{AnsQ}$ and is a pair $\langle R,\ q' \rangle$ of a set of records R and a new query q' resulting from a rewrite operation when $x = \text{AnsQ}$. The answer contains a *set* of resource records because multiple records might be relevant for a query (e.g., there might be multiple NS records for a domain).

3.2 DNS Semantics

Given these definitions, we now formally define how DNS resolves user queries. We model the DNS system as a 4-tuple, $C = \langle S, \Theta, \Gamma, \Omega \rangle$, called a configuration C, where:

- S is a set of nameservers (e.g., `ns1.fnni.com`). We leave nameservers as opaque objects and associate them with other information through functions.
- $\Theta \subseteq S$ is a set of "root" nameservers for S.

DNS resolution $\boxed{\text{QUERY} \times \text{CONFIG} \times \mathbb{N} \to \mathcal{P}(\text{ANSWER})}$

$$\textsc{Resolve}(q, \langle S, \Theta, \Gamma, \Omega\rangle, k) = \bigcup_{s \in \Theta} \textsc{Resolve}(s, q, \langle S, \Theta, \Gamma, \Omega\rangle, k)$$

$$\textsc{Resolve}(s, q, \langle S, \Theta, \Gamma, \Omega\rangle, k) = \begin{cases} \{\langle \textsc{ServFail}, \emptyset\rangle\} & s = \bot \ \vee \ k = 0 \\ \textsc{Resolve}(q', \langle S, \Theta, \Gamma, \Omega\rangle, k-1) & s \neq \bot, k > 0, a = \langle \textsc{AnsQ}, \langle R, q'\rangle\rangle, \ \mathcal{N}(\Gamma(s), q') = \emptyset \\ \textsc{Resolve}(s, q', \langle S, \Theta, \Gamma, \Omega\rangle, k-1) & s \neq \bot, k > 0, a = \langle \textsc{AnsQ}, \langle R, q'\rangle\rangle, \ \mathcal{N}(\Gamma(s), q') \neq \emptyset \\ \bigcup_{r \in \mathcal{T}(R,\text{NS})} \textsc{Resolve}(\Omega(\text{ans}(r)), q, \langle S, \Theta, \Gamma, \Omega\rangle, k-1) & s \neq \bot, k > 0, a = \langle \textsc{Ref}, R\rangle \\ \{a\} & \text{otherwise} \end{cases}$$

where $a = \textsc{ServerLookup}(\Gamma(s), q)$

Figure 4: DNS Recursive resolution semantics.

- $\Gamma : S \to \mathcal{P}(\text{ZONE})$ is a function from a nameserver to the zones for which that nameserver is authoritative.
- $\Omega : D \to S \cup \{\bot\}$ is a function from a domain name to the nameserver identified by that name or $\bot$ if no corresponding nameserver exists.

We define the semantics of DNS in two parts: first we define how a single authoritative nameserver processes a query locally, and then using this formulation, we define DNS resolution.

3.3 Authoritative Nameserver semantics

Given a set of zone files Z and a query q, the definition of ServerLookup at the bottom of Figure 3 defines the lookup performed at a nameserver for the query. The result of this lookup is a DNS answer. The first step is to find the zone z that has the longest matching prefix ($\text{dn}(z)$) with the domain in the query ($\text{dn}(q)$) — the function $\mathcal{N}$. The notation max_{dn} selects those zones with maximal domain names according to the domain name partial order, among those that are prefixes of the query domain name. For example, if a nameserver has zones for `com` and `gmail.com` and the user's query is for `help.gmail.com`, then the nameserver will choose the `gmail.com` zone to answer the query. If there is such a matching zone z, then ServerLookup calls ZoneLookup to get an answer by evaluating the query against the zone file. Otherwise, the nameserver refuses (Refused) to perform the lookup operation as it could not find a relevant zone.

ZoneLookup selects the appropriate resource records r for the zone z by choosing the maximal elements with respect to the query ($<_{q,z}$) as defined in equation (8) in Figure 2. The set of records passed to RRLookup will necessarily have the same domain name, *i.e.*, $\text{dn}(r_1) = \text{dn}(r_2)$ for any r_1, r_2 in the set. However their types may differ. Thus, for such a set R, we simply write $\text{dn}(R)$ to refer to the domain name for elements in this set.

The RRLookup function takes a set of resource records R and a query q along with the zone z and produces an answer. The goal of RRLookup is to return either **(1)** an answer (Ans), if the resource records R are sufficient to answer the query q, **(2)** a referral (Ref), if records R cannot answer the query q but indicate who might have the answer, **(3)** an intermediate answer r' and a query q' (AnsQ), if the resource records R establish that the query q would be modified to query q' due to resource record r', or **(4)** an error message (NX), indicating that the domain does not exist.

RRLookup implements the DNS resolution process given in RFC 6672 §3.2 as server algorithm (SA) [42]. Note that we exclude SA steps 1 and 5 since the formal model does not capture dynamic elements like caches. Step 2 of SA is captured by $\mathcal{N}$ stated earlier. When the records' domain name exactly matches the query, the ExactMatch function is applied (SA, steps 3A, 3B). Otherwise, if it is a wildcard domain that matches the query domain, the WildcardMatch case will apply (SA, step 3C). If the records contain a matching `DNAME` record, which is only possible when the other two cases do not apply, then the query will be modified according to the Rewrite function (SA, step 3C). If no such record exists, DNS will delegate the query to another nameserver if it has an `NS` record (Delegation) (SA, step 3B). Finally, if all else fails, the nameserver will return `NXDOMAIN` (non-existent domain, SA, step 3C).

The Exact Match and Wildcard Match cases are both broken down further into several cases. For the Exact Match case, if there is a authoritative record with the same type as the query, then the nameserver will simply return this record (SA, step 3A). Of all the records passed to RRLookup, a zone z is authoritative of all the records except for `NS` records (zone cut) not accompanied by an `SOA` record. Otherwise, if there is a `CNAME` record (SA, step 3A), then the nameserver will perform a rewrite (AnsQ), returning the relevant records R, as well as a new query domain given by $\text{ans}(q)$ with the same type ($\text{ty}(q)$). If there is no `CNAME` record, but there is a non-authoritative `NS` record, then the nameserver will perform a Delegation (SA, step 3B). Finally, if all else fails, it will simply return an answer with no information ($\emptyset$).

The Wildcard Match case is similar to the Exact Match case, except it will perform synthesis (Syn) to generate a new set of records specializing the wildcards. For instance, a lookup for a query with domain `email.com` on a set with a single wildcard record `*.com` generates a (cachable) synthesized record for `email.com`.

The Rewrite case for `DNAME` records returns AnsQ with records and a new, rewritten query. The new query is given by DProc, which generates and adds a new synthesized `CNAME` record for the answer and substitutes the matching prefix of the query with the rewrite described in the record answer ($\text{dn}(q)[\text{dn}(r) \mapsto \text{ans}(r)]$). The DNS adds these `CNAME` records to the answer to facilitate caching — future queries are rewritten based on the cached `CNAME` record (SA, step 3C). The Delegation case returns the `NS` records along with the necessary `A` and `AAAA` glue records (SA, step 6).

3.4 Recursive Resolution Semantics

Now that we have formally defined how a nameserver answers a query q, we can use this definition to formalize the process of recursive resolution (Figure 4). We define two functions named Resolve that return a *set* of possible answers. The functions return *sets* of answers in order to capture the nondeterminism inherent in

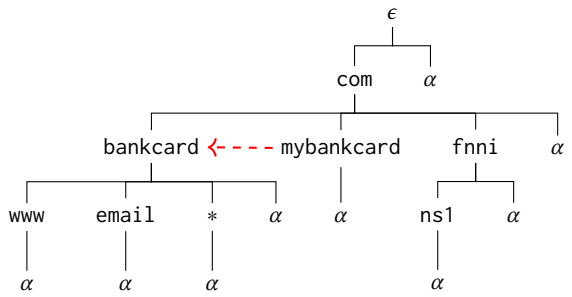

Figure 5: Label Graph used for equivalence class generation for the zone files from Figure 1. Note, only the domain name (d) field of the records are used but not the answer (a) field. The dotted red edge represents the DNAME redirection of (f).

DNS. The first function takes a query q, a configuration $\langle S, \Theta, \Gamma, \Omega \rangle$, and a fuel parameter k, which is used to imitate the mechanism used by DNS to ensure that resolution terminates. The function works by resolving the query q at each root nameserver $s \in \Theta$, taking the union of their results.

The second RESOLVE function performs resolution at a specific nameserver s. There are several cases based on the result of SERVERLOOKUP. In the first case, if the resolver has already exceeded the execution bound k or the input is a $\perp$ due to a nameserver lookup failure in Ω from a recursive call of RESOLVE, it returns SERVFAIL with no records. Otherwise, if s returns a rewrite ANSQ, and does not have a local zone that can process the rewrite ($\mathcal{N}(\Gamma(s), q') = \emptyset$), then DNS resolves the new query q' starting over at the root. If there is a local zone at s, then it processes q' at s. If s returns a referral REF, then the function unions the results from nondeterministically resolving the query at each nameserver identified in the returned NS records ($\Omega(\text{ans}(r))$). Finally, if SERVERLOOKUP returns any other kind of answer, RESOLVE simply returns that answer ($\{a\}$).

4 A FAST VERIFICATION ALGORITHM

We leverage the model in §3 to present a fast verification algorithm based on the enumeration of query *equivalence classes* (ECs).

4.1 Equivalence Class Generation

The idea with our approach is that, instead of enumerating all possible queries, we can construct a collection of *equivalence classes* of queries (sets of queries that will be resolved the same way by DNS). Intuitively, two DNS queries are in the same EC if the queries are resolved locally in the same way (and rewritten similarly) at every nameserver. We define this notion of equivalence more formally and prove that it is correct in §5. The set of ECs our algorithm computes need not be, and indeed is not, always minimal.

Other verification tools such as Veriflow [29] and Atomic Predicates [47, 48] use a similar approach in the context of packet forwarding. However, Veriflow's technique does not support query rewrites, which we require in the context of DNS. Atomic Predicates does support query rewrites but is overly general for our purposes and hence more expensive than necessary. For example, even in the absence of rewrites, using Atomic Predicates to compute ECs for DNS would require a quadratic number of predicate intersections. In contrast, we leverage the hierarchical, tree-like structure of domain names to reduce this cost. Specifically, we show in §5 that in the absence of DNAME rewrites, our approach computes the set of ECs in linear time.

Label graph construction. As a first step to generate query ECs, our algorithm builds a *label graph*, which is the union of the domain names of all the records that appear in any zone file at any nameserver. Consider again the running example from Figure 1: the corresponding label graph is shown in Figure 5. The label graph is rooted at ϵ and every domain name that appears as the key of some resource record in some zone file is represented in the graph as a path (sequence of labels) starting from the root. For instance, nameserver `ns1.fnni.com` has a DNAME record for `mybankcard.com`, so `mybankcard` shows up as a node beneath the node for `com`.

For DNAME records, we also add the rewrite target for the record to the label graph, along with a dashed line between the source and target: because the answer for the `mybankcard.com` DNAME record is `bankcard.com`, a line appears from `mybankcard` to `bankcard`.

Finally, for every node (label) in the graph, we add an α child, which represents an arbitrary sequence of labels $\alpha = l_k \circ \ldots \circ l_0$ such that l_0 is unique from its siblings.

Path enumeration. Every path through the label graph from the root corresponds to several equivalence classes, one for each query type. The algorithm begins by enumerating all paths starting from the root. Whenever it encounters α, it constrains it to exclude its siblings. For the example in Figure 1, we start to compute the following ECs, one for each type $t \in$ TYPE:

(1) $\langle \epsilon,\ t \rangle$
(2) $\langle \texttt{com} \circ \epsilon,\ t \rangle$
(3) $\langle \texttt{bankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(4) $\langle \texttt{www} \circ \texttt{bankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(5) $\langle \alpha \circ \texttt{www} \circ \texttt{bankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(6) $\langle \texttt{email} \circ \texttt{bankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(7) $\langle \alpha \circ \texttt{email} \circ \texttt{bankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$

At this point, the algorithm encounters the wildcard ($*$) label under `bankcard`. For the purposes of building the label graph, we simply treat wildcards as character labels ('$*$'), as such characters are valid and will experience an exact match with a wildcard record. We instead use the α labels to represent ECs for domains not explicitly mentioned in the zone files. At this point, the algorithm produces the ECs:

(8) $\langle * \circ \texttt{bankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(9) $\langle \alpha \circ * \circ \texttt{bankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(10) $\langle \alpha \circ \texttt{bankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle \quad \alpha[0] \notin \{\texttt{www}, \texttt{email}, *\}$

DNAME rewrites. The next paths traversed are those for `mybankcard.com`. Since `mybankcard` has a DNAME record, we continue enumerating paths through the dashed edge. However, since we want to capture the input query before the transformation, we do not concatenate the target of the rewrite to the path. This results in a set of ECs that are analogous to those for `bankcard.com`.

(11) $\langle \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(12) $\langle \texttt{www} \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(13) $\langle \alpha \circ \texttt{www} \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(14) $\langle \texttt{email} \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(15) $\langle \alpha \circ \texttt{email} \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(16) $\langle * \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$
(17) $\langle \alpha \circ * \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$

(18) $\langle \alpha \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle \quad \alpha[0] \notin \{\texttt{www}, \texttt{email}, *\}$

The algorithm would similarly continue to explore paths for fnni.com and terminates with the EC, $\langle \alpha \circ \epsilon,\ t \rangle, \alpha[0] \notin \{\texttt{com}\}$.

DNAME loops. The label graph can have loops due to DNAME edges. The first type of loop has both solid and dotted edges; for example, this type of loop would exist if there were another DNAME edge from email to mybankcard. In such cases, the algorithm traverses the loop and continues to generate ECs until the domain name of the path exceeds the maximum length allowed by DNS. With our example loop, suppose the algorithm takes the DNAME edge from mybankcard node and reaches email. After generating the EC given by (14), it would take the dotted edge back to mybankcard and then the dotted edge back to bankcard. It then traverses the paths underneath bankcard but with the original query prefix before rewriting, so it will generate $\langle \texttt{www} \circ \texttt{email} \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$, $\langle \alpha \circ \texttt{www} \circ \texttt{email} \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon,\ t \rangle$, and so on.

The second type of loop is entirely made up of dotted edges, for example if bankcard had a dotted edge back to mybankcard. This situation can arise if there is another zone file for $\texttt{bankcard} \circ \texttt{com} \circ \epsilon$ at a different nameserver with this DNAME record. This situation constitutes an infinite loop since the length of the path never increases a query enters the loop. To check for an infinite loop, each node in the label graph stores the path length when the algorithm enters the node and checks if the stored length is equal to the new path length before updating. As soon as the algorithm detects an infinite loop, it backtracks and continues.

4.2 Symbolic Execution of ECs

To determine the behavior of the equivalence classes, we symbolically execute each EC using our semantics from Figures 2 to 4. To symbolically execute an EC that starts with α, we observe that by construction α cannot match any of the records present at a given zone file except a wildcard. Therefore, we can leave α opaque during symbolic execution and simply use this knowledge to precisely determine the answers for such an EC. Our symbolic execution algorithm builds an *interpretation* graph for each EC, representing all nondeterministic execution traces that are possible in DNS for that EC. Each node in an interpretation graph represents a call to the second Resolve function and the node stores the nameserver s identified by d, the query q, and the answer a returned by the ServerLookup function. An edge is drawn from one node to the other if the Resolve at the parent node returns a Ref to the nameserver of the child node.

Symbolically executing an EC separately for each query type leads to an inefficient implementation; DNS supports dozens of record types, and there is substantial overlap in how they are treated during execution. Therefore, GRoot executes the ECs for all record types at once using a compact bitset representation for types, splitting nodes when different types experience different behaviors according to Figures 3 and 4. The result is a single graph representing multiple interpretation graphs.

Figure 6 shows the result of symbolic execution for the running example for three equivalence classes, which are compactly represented as: $\langle \alpha \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon, \{\texttt{A}, \texttt{MX}, \texttt{TXT}\} \rangle$. We show just these three records types for simplicity. The execution starts at a.gtld-servers.net and then proceeds to either ns1.fnni.com or ns2.fnni.com from NS referrals. In either case, the execution has a DNAME rewrite before eventually splitting the record types into two cases: one for {A} and another one for {MX, TXT} to capture the diverging behaviors. GRoot encodes the relevant set of types at each node using a fixed-size bitset, with one bit per type.

Delegation Inconsistency
A parent node with the Ref tag and a child node with the Ans tag do not have the same set of NS and A records for delegation.
Lame Delegation
Interpretation graph has a node with the Refused tag.
Missing Glue Records
Node answer contains NS records but not the A (glue) records.
Non-Existent Domain for Service
Sink node return an answer with the NX tag.
Cyclic Zone Dependency
Interpretation graph contains a cycle
Rewriting Loop
Interpretation graph contains a cycle with at least one rewrite.
Query Exceeds Maximum Length
Query at some node exceeds the maximum label or total length.
Answer Inconsistency
Different sink nodes return different answers.
Zero Time To Live
Sink node returns an answer with TTL value set to 0.
Rewrite Blackholing
A path has a rewrite and ends at a node with NX tag.

Table 2: Bug finding Implementation for Table 1.

4.3 Checking Properties

The representation for ECs and their interpretation graphs facilitates efficient checking for a wide variety of properties. We write property checkers as custom graph algorithms (§ 6) that process each of the interpretation graphs. A property that is true of all interpretation graphs holds for all possible executions of Resolve, for all possible queries. Table 2 summarizes the implementation of checkers for the bugs listed in Table 1. Because the interpretation graph contains full information about the execution traces, it can also be used to enforce non-functional properties, for example related to performance, such as a bound on the number of rewrites in any execution of Resolve.

5 PROOF OF CORRECTNESS

We prove our approach with GRoot is correct in two steps. First, we show that our equivalence class generation algorithm computes classes of queries that adhere to a restrictive notion of equivalence called *strong equivalence*. Next, we prove that strong equivalence implies equivalence for DNS resolution.

A challenge for defining equivalence of DNS resolution is that queries that match all the same zone records can still end up with different answers due to record *synthesis* (Syn from Figure 2), which generates specialized records for use in the cache. For example, two queries with domain names a.mybankcard.com and b.mybankcard.com may both match the wildcard record *.mybankcard.com, which will generate new records, one for a and one for b with the exact query names. Since we do not model the effect of caching in this paper, we want to prove equivalence of

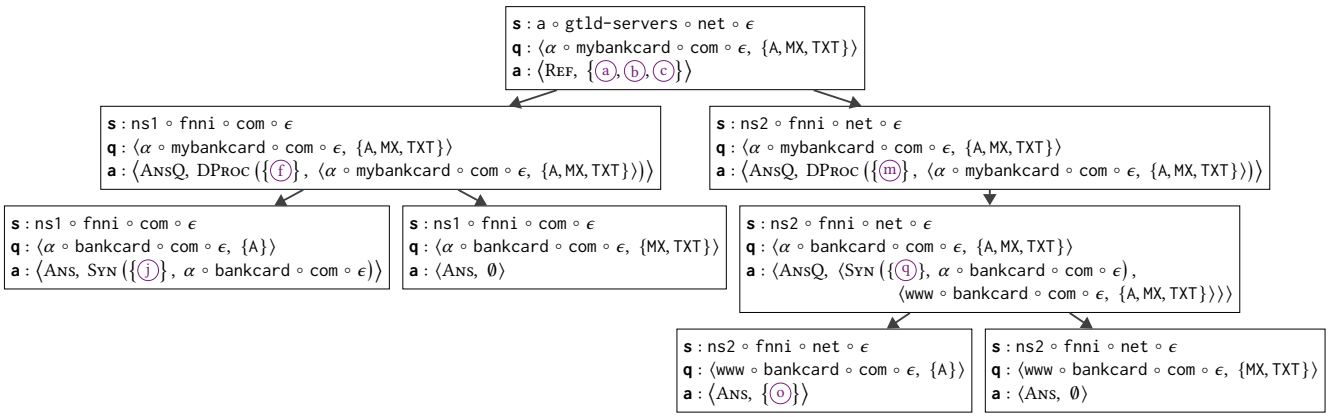

Figure 6: The interpretation graph based on the zone files shown in Figure 1 for types {A, MX, TXT} for the equivalence classes given by the schematic query: ⟨α ∘ mybankcard ∘ com ∘ ϵ, {A, MX, TXT}⟩ with α[0] ∉ {www, email, ∗}.

DNS resolution up to such differences. To do so, we define a notion of equivalence between answers that ignores synthesized records. In particular, we define a relation $a_1 \approx a_2$ to mean two answers are equivalent up to synthesized records. For brevity, we defer defining $\approx$ to the appendix.

We now describe our *strong equivalence* relation. Strong equivalence views queries as equivalent if they are treated equivalently at each individual nameserver s, even if that nameserver can never be contacted with that particular query.

Definition 5.1 (Strong equivalence)**.** For a given configuration $C = \langle S, \Theta, \Gamma, \Omega \rangle$, the binary relation $\sim_C$ on queries, which we call the *strong equivalence* relation, is the greatest relation such that $q_1 \sim_C q_2$ implies that for all servers $s \in S$, where $a_i = \textsc{ServerLookup}(\Gamma(s), q_i)$, we have **(1)** $\mathcal{N}(\Gamma(s), q_1) = \mathcal{N}(\Gamma(s), q_2)$, **(2)** $a_1 \approx a_2$, and **(3)** for any rewrites $q_1' \in \text{query}(a_1)$ and $q_2' \in \text{query}(a_2)$, $q_1' \sim_C q_2'$.

The next step in proving our approach is correct, is to show that the algorithm presented in §4.1 computes equivalence classes of queries satisfying the $\sim_C$ relation.

Theorem 5.1 (EC generation sound)**.** For a given configuration $C = \langle S, \Theta, \Gamma, \Omega \rangle$, if two queries q_1 and q_2 are in the same EC computed by the algorithm, then $q_1 \sim_C q_2$.

Proof. Direct by case analysis of ServerLookup. Full proofs are included as extra material in the appendix. □

Finally we prove that strong equivalence implies equivalence of DNS resolution.

Theorem 5.2 (Soundness)**.** For all C, q_1, q_2, and k, if $q_1 \sim_C q_2$, then $\textsc{Resolve}(q_1, C, k) \approx \textsc{Resolve}(q_2, C, k)$.

Proof. We start by proving a slightly stronger invariant: $\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega \rangle, i) \approx \textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega \rangle, i)$ for all s, i, and show that it implies the result. The proof given in appendix proceeds by induction on the length of resolution step i. □

In the appendix we also prove two other theorems about our technique. First we prove a *completeness* result.

Theorem 5.3 (Completeness)**.** For a configuration C, each query q belongs to at least one computed equivalence class.

Together, soundness and completeness imply that our technique indeed performs verification: all possible queries are represented by the ECs, and all queries within an EC have the same behavior.

Second, we prove that, in the absence of DNAME rewrites, our algorithm will compute a linear number of ECs in linear time with respect to the number of zone records. Given that CNAME rewrites are comparatively much more common that DNAMEs in practice (§7), this result implies that in many cases GRoot can verify DNS configurations very efficiently.

Theorem 5.4 (Linear time)**.** In the absence of DNAME records, for a collection of zone files with n resource records, our algorithm computes $O(n)$ equivalence classes in $O(n)$ time.

Given that the total number of possible DNS queries is $\sum_{i=0}^{253} 38^i$ (for 38 valid characters), this theorem shows that GRoot can provide a massive reduction in complexity.

6 IMPLEMENTATION

GRoot is implemented in over 4300 lines of C++ code and uses the Boost Graph Library [5] as well as custom zone file parsers. GRoot takes as input a directory containing a collection of zone files as well as an optional file specifying what properties to check. In the absence of this properties file, GRoot checks for a set of bugs that are considered always harmful (e.g., rewrite blackholing and loops).

Users implement new static analyses in GRoot as simple C++ functions that process an interpretation graph. To make this easier, we provide three separate checker APIs. The first lets the user process each node in the interpretation graph in isolation, which can be used for simple checks such as: "query X should never return NXDOMAIN". The second lets the user process each path through the graph in isolation, and the third provides the entire graph.

Since each interpretation graph is checked separately by a property checker, the graphs can be checked in parallel. Our implementation takes advantage of this and also pipelines EC generation with symbolic execution: as soon as an EC is generated, GRoot uses an idle worker thread to build the interpretation graph for that EC and checks the properties on the resulting graph.

Since strings are used pervasively in GRoot to represent labels in the zone graphs, label graph, and interpretation graph, and since adding new records to each of these graphs involves multiple string

comparisons, we opted to use a custom string interning strategy that replaces string labels with unique ids, for faster operations.

GRoot is available as open source software[1].

7 EVALUATION

To evaluate GRoot we aim to show (1) it can find bugs in real DNS configurations, and (2) it can scale to large sets of zone files. We describe our methodology and results next.

7.1 Networks studied

We evaluate GRoot on zone files from three networks:

A university network. We ran GRoot on the DNS configurations obtained from a large campus network. The configurations for the network are managed in a decentralized fashion: the campus IT service manages the DNS tree starting from the subdomain `campus.edu`[2] The `campus.edu` zone has four authoritative nameservers (`ns{1,2,3,4}.dns.campus.edu`), which are slaves of a hidden master server. The Infoblox platform [25] is used to maintain the master server and keep the slaves up-to-date. The `campus.edu` zone file has delegations for 1850 subdomains and each department in the university is responsible for managing a subset of those subdomains. Of these subdomains, 895 have secondaried their zones back to the four campus nameservers, which also provide authoritative answers for queries related to those subdomains. The remaining 955 subdomains require delegation via `NS` records.

We use the `campus.edu` zone file and the zone files of the 895 subdomains that are secondaried by the campus nameservers for our experiments, since we are able to obtain these zone files through zone transfers. In total the `campus.edu` zone file has 8555 records and there are a total of 111,539 records across the remaining 895 subdomains. Figure 7(a) shows the cumulative distribution of subdomains to the number of resource records that they contain.

An infrastructure service provider. We ran GRoot on 1241 internal zone files of a large infrastructure service provider. All the zones were independent i.e., there isn't a zone that is a subdomain of another zone in the dataset. All of the zone files are assumed to be taken from a single name server. The data set consists of around 3.6 million resource records with the largest zone file accounting for 1.6 million records.

DNS census data. This data set is publicly available [11]. It consists of around 2.6 billion resource records (157 GB) that were collected through live DNS queries in 2012-2013. These records are stored as CSV files — one file for each DNS record type (`A`, `AAAA`, `CNAME`, `DNAME`, `MX`, `NS`, `SOA`, `TXT`). These records are stored lexicographically: by hostname and time. For each hostname and each type, we picked the resource records corresponding to the latest timestamp. This leaves 1.05 billion resource records. We partition this set into zone files by using the `SOA` records and the DNS namespace hierarchy.

While creating the zones we also added NS records along with the necessary glue records to both parent and child so that there will not be any delegation inconsistencies or lame delegation. The dataset consists of 285 top-level domains (TLDs). For our experiments, we considered all the second-level domains (for example, `co.uk.`)

[1] https://github.com/dns-groot/groot

[2] The university name is anonymized as "campus."

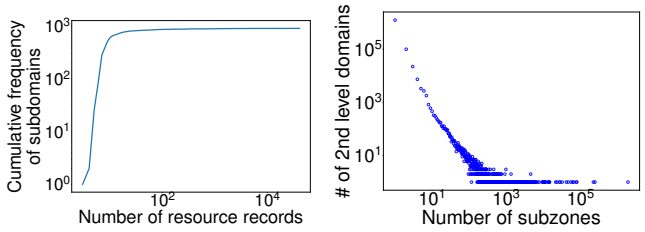

Figure 7: Dataset statistics. (a) Cumulative number of subdomains with a number of resource records in the campus network. (b) Number of 2nd-level domains with a given number of subzones for DNS census.

that have at least one subdomain zone file under them. There are 1,368,523 such domains totalling over 65 million resource records. The synthesized dataset and the software artifact are available on Zenodo [26]. Figure 7(b) shows the distribution of second-level domains to the number of subzones they contain.

Features used. Table 3 shows a summary of the features used in the two datasets. For the campus network, there were 63 wildcard records and over 4000 `CNAME` records. However, the configurations did not make use of `DNAME` records. In contrast, the part of the DNS census dataset that we used included over 200 `DNAME` records and over 2 million `CNAME` records. However, it did not have any wildcards. This is likely due to the dataset being collected from live DNS queries, which are almost never directly for wildcard resources. The service provider dataset is dominated by the `CNAME` records as the provider employs `CNAME` chains frequently to map queries.

7.2 Functionality Experiments

7.2.1 University network. We use the data from the university network to evaluate whether GRoot can find bugs on a real network. We performed two different classes of checks using GRoot (summarized in Table 4) based on properties described in Table 1. For the properties in Table 1 but not in Table 4, GRoot was not applicable for this network (e.g., answer inconsistency due to master-slave replication). Properties shown below the dashed line showcase GRoot's ability to help operators explore and understand the behavior of their DNS configurations. Violations of these properties are not necessarily bugs but are interesting behaviors that an operator may be interested to examine. For example, we used GRoot to identify lookups that involve rewrites outside of the campus domain — most are (likely) intentional. Because GRoot is complete, it reported *all possible ways* in which such rewrites can occur.

Property Violations. Violations of the first seven properties in Table 4 represent true misconfigurations and are common operational and configuration errors described in RFC 1912 [3]. We contacted operators, and those that responded confirmed our findings (because DNS management is decentralized there are many administrators responsible for these domains and we did not hear back from all of them). We discuss some example violations here:

GRoot flagged 49 domains of the form α`.campus.edu` that have a *delegation inconsistency*. These 49 domains are managed by 25 different administrators. We emailed all of them (obtaining email addresses from the SOA records); seven emails bounced, and nine

Dataset	SOA	A	NS	CNAME	DNAME	MX	TXT	Wildcard	Other
Campus	895	97,951	9,209	4,259	0	1,978	363	63	883
Service Provider	1,239	110,052	8,740	3,442,892	0	1,878	1,339	2,059	3,586
Census	6,668,062	18,598,682	29,855,307	2,168,115	218	6,965,866	1,301,472	0	118,629

Table 3: Summary of features used in the two datasets studied.

Property	Number of issues
Delegation Consistency	49★
No lame delegation	9★
No rewrite loops	2★
No missing glue records	1★
No rewrite blackholing	48★
No query exceeds maximum length	0★
No zero TTL	0★
No rewrite to outside domain	378†
No resolution at an external NS	324†
Number of rewrites ≤ 2	24†

Table 4: Properties checked on the campus network and the number of cases GRoot reported. Cases in red (★) are bugs while orange(†) are warnings.

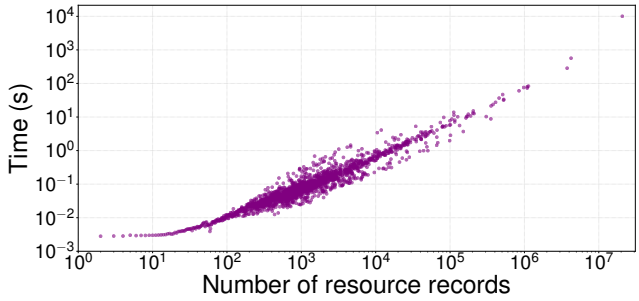

Figure 8: Total time to build label graph and check for properties for the 1,368,523 second-level TLDs. The median time is taken when multiple domains have the same number of resource records.

people responded, in all cases acknowledging the inconsistency as a misconfiguration. Some of the NS records in the `campus.edu` were incorrectly pointing to a web server instead of the zone's authoritative name server. One operator commented: "we haven't noticed this discrepancy because we almost never use DNS names for DNS servers, we use IPs." Another operator explained: "the short answer is negligence."

Some of these violations affect performance. *Lame delegation* affects the mean response time of DNS lookups: a lookup on some name servers will fail, meaning the resolver would then need to contact a different name server. The same is true of *rewrite loops* where we found `CNAME` records that were rewritten to the same record. In both cases of rewrite loops, the relevant admins confirmed the misconfigurations and removed the corresponding entries. Other forms of loops can also add to resolution latency. This was the case for the *missing glue record* bug where a resource record

```
dept.campus.edu   IN   NS   dc1.dept.campus.edu
```

existed but had no `A` record for `dc1.dept.campus.edu`. Resolving `dc1.dept.campus.edu` would lead the resolver to lookup the IP address, only to end up back at this record. GRoot flagged 48 domain names that were rewritten to a domain name not existing in the zone files, causing DNS to return NXDOMAIN. When asked, the operators replied, "they are CNAME entries that were missed during a prior retirement. These are entries that were orphaned accidentally when the source server was removed a few years back. Our tools do not auto clean up the CNAME aliases and this sometimes occurs. We do not actively black-hole server DNS entries."

GRoot found out that there is no input query that can lead to the violation of the last two properties in this network.

Configuration Understanding. The properties at the bottom of Table 4 demonstrate GRoot's utility for understanding and exploring configurations. For example, GRoot found 378 cases where the query is rewritten to a domain that is not a subdomain of `campus.edu`. GRoot guarantees that these 378 are the only cases under `campus.edu` that can be rewritten to outside domains. Hence an admin can manually or automatically inspect the results to spot errors or ensure policy is respected, with the assurance that *all possible scenarios* are covered.

In fact, the other two properties that we checked with GRoot identified actual misconfigurations. One check identified lookups that use a name server not under `campus.edu`. Most of these nameservers belong to AWS or Cloudfare and are likely intentional. But one name server was α.campus**.ed** which looked suspicious; when asked the admin said: "Thanks for the information about the delegation. I've corrected the typo."

The other check identified 24 queries that are rewritten more than twice during lookup. This is unusual as the RFC [37] suggests `CNAME` should point at the primary name and not an alias. Long `CNAME` chains increase the query response time and can lead to loops. Further, certain resolvers do not follow a chain if the length increases beyond a threshold and instead return ServFail [7].

7.2.2 Service Provider. We also performed checks based on the properties described in Table 1 on the zone files from the service provider. Since there are no parent-child zones in the data set, all the violations GRoot flagged were related to *rewrite blackholing*. GRoot flagged around 160k interpretation graphs out of 9.2 million as experiencing rewrite blackholing. Upon further investigation with the service provider, they informed us that nearly all the cases are due to incomplete decommissioning of host names that are no longer in use.

7.3 Performance Experiments

All experiments were run on an 8-core Intel i7 processor with 32GB of RAM running Windows 10 using 8 threads. On the campus network data the total time to parse all of the zone files and build the label and zone graphs was 1.5 seconds. GRoot generated 212,113 graphs and checked properties for the graphs in 7 seconds. The label graph used to generate equivalence classes had 105,030 nodes with 105,029 edges and the interpretation graphs generated had on

average 6 nodes with 5 edges while the maximum graph size was 17 nodes. The median graph size was also around 6 nodes.

We next explore the ability of GRoot to scale to larger zone files by checking the same properties on the DNS census data.

Figure 8 shows the time taken in seconds by GRoot for building the label graph and checking the properties for the 1,368,523 domains. The median time is taken when multiple domains have the same number of resource records. The total time increases roughly linearly with the number of resource records. The two other secondary factors which affect the running time are the number of subdomains and the average size of the interpretation graphs built. The more the number of subdomains and the larger the graphs built, GRoot takes longer to finish. The figure also shows that GRoot can scale to tens of millions of records.

8 DISCUSSION AND LIMITATIONS

To our knowledge GRoot is the first tool that allows operators to verify the correctness of their DNS configuration (zone) files, or those of their hosted customers.

Incremental deployment. GRoot can be incrementally deployed for several reasons. First, operators can independently verify their local zone files. Second, companies such as Akamai, Microsoft, Google, and Amazon not only manage their own DNS but also that of their customers [1, 2, 19, 36]. Hence, these companies have a greater opportunity and incentive to verify customer configurations on their behalf, making it much easier for those customers to leverage GRoot as well.

Static not dynamic bugs. As a "compile time" checker, GRoot does not model dynamic phenomena that affect DNS results such as caching, server failures, and network unreachability. GRoot must be complemented by live testing tools to account for bugs caused by such phenomena.

Local not global correctness. Because GRoot can only analyze the zone files that it is given, it can only verify the correctness of the DNS configuration of the organization that owns those files. The end-to-end correctness of the DNS configuration (globally) hinges on other organizations doing the same.

Snapshot not incremental. GRoot verifies a snapshot of the current zone files which may be inefficient when changes to zones files are small and frequent. We leave optimizing GRoot for small incremental changes for future work.

Properties on single queries not multiple. Our current implementation only supports properties for individual DNS queries. However, our verification approach can be easily modified to support properties over a set of queries, at the cost of increased memory and execution times.

9 RELATED WORK

GRoot is related to several prior lines of work:

DNS testing. Many operators today use blackbox techniques for checking DNS correctness (e.g., live testing and monitoring). For example, operators can monitor for ongoing problems through offerings from commercial vendors, such as ThousandEyes [28], CheckHost [22] or research tools [40].

These approaches are incomplete because they lack direct knowledge of the configurations and cannot comprehensively explore the space of possible DNS queries. Therefore, they cannot provide correctness guarantees. These approaches are further complicated by artifacts of deployed DNS systems such as caching, load balancing, and geo-replication. In contrast, GRoot is based on static verification of zone files and so gives strong guarantees about correctness.

A relevant approach in this space is *linting* of DNS configuration files. Tools like dnslint [35] report possible violations of best practices in configuration files based on a simple syntactic analysis of the files. Such tools can be effective at discovering certain kinds of common misconfigurations but cannot perform deeper semantic analysis (e.g., whether a query might resolve to `NXDOMAIN`).

DNS modeling. To the best of our knowledge, this paper presents the first formal semantics for DNS. Perhaps the closest work is IRONSIDES [8], a DNS server implementation that is provably robust to data flow exceptions such as unexpected exceptions. However, IRONSIDES is a particular *implementation* of DNS and as such neither provides a formal model for DNS nor can be used to verify DNS configurations.

Network verification. There is a large body of work on verifying the network routing layer, and researchers have proposed numerous techniques to perform such verification generally, efficiently, and incrementally [4, 14, 17, 27, 29, 33, 34]. While the semantics of routing and forwarding are well understood (e.g., longest prefix matching), the semantics of DNS is relatively poorly understood by comparison. As such, we believe our formal model of DNS is a contribution that can serve as the basis of future work in this area. More generally, while there are some superficial similarities between routing and DNS, the specific details are vastly different. For example, DNS introduces new challenges due to nondeterminism, query rewriting, delegation, and distributed management. For certain cases, GRoot can generate equivalence classes asymptotically faster than approaches used for routing verification due to the hierarchical structure of domain names.

10 CONCLUSIONS

In this paper, we presented GRoot, the first verification tool for DNS configurations. GRoot operates by generating an exhaustive set of equivalence classes of DNS queries and then symbolically computing the DNS resolution process for each class. Properties in GRoot are added as simple C++ functions that analyze the structure of the resulting symbolic execution graphs. To show that our approach is sound, we present a novel formal model of DNS resolution and prove that queries in the same equivalence class will be resolved the same way by DNS. Finally, we demonstrate that GRoot can efficiently analyze real DNS configurations in practice, leading to the discovery of numerous misconfigurations.

ACKNOWLEDGMENTS

Thanks to the SIGCOMM reviewers and our shepherd for their helpful comments. Thanks to the anonymous DNS operators for their feedback on GRoot's results. Thanks to Kyle Schomp for suggesting the rewrite blackholing property. Thanks to Karthick Jayaraman and Karthikeyan Ravichandran for feedback on use cases. Thanks to Saswat Padhi for feedback on the formal model and GRoot's implementation. This work was partially supported by an MSR internship and NSF grants CNS-1704336 and CNS-1901510.

REFERENCES

[1] Akamai. 2020. Fast DNS. https://www.akamai.com/us/en/products/security/fast-dns.jsp

[2] Amazon. 2020. Amazon Route 53. https://aws.amazon.com/route53/

[3] David Barr. 1996. Common DNS Operational and Configuration Errors. RFC 1912. https://doi.org/10.17487/RFC1912

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. ACM, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

[5] David Abrahams Beman Dawes. 2005. Boost C++ Libraries. https://www.boost.org/

[6] Thomas P. Brisco. 1995. DNS Support for Load Balancing. RFC 1794. https://doi.org/10.17487/RFC1794

[7] Jonas Bushart and Christian Rossow. 2018. DNS Unchained: Amplified Application-Layer DoS Attacks Against DNS Authoritatives. In *Research in Attacks, Intrusions, and Defenses*, Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.). Springer International Publishing, Cham, 139–160.

[8] Martin Carlisle and Barry Fagin. 2012. IRONSIDES: DNS with no single-packet denial of service or remote code execution vulnerabilities. (2012), 839–844.

[9] Stuart Cheshire and Marc Krochmal. 2013. DNS-Based Service Discovery. RFC 6763. https://doi.org/10.17487/RFC6763

[10] Internet Systems Consortium. 2020. BIND 9. https://www.isc.org/bind/

[11] DNS Census 2013. Retrieved September 2019 from https://dnscensus2013.neocities.org/

[12] DNS Response Policy Zones 2019. Retrieved June 2020 from https://dnsrpz.info/

[13] DNSBL information - spam database and blacklist check. 2020. https://www.dnsbl.info/

[14] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 217–232. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/fayaz

[15] Incident Report for npm. 2018. DNS misconfiguration cached in ISP DNS caches. Retrieved June 2020 from https://status.npmjs.org/incidents/v22ffls5cd6h

[16] James Fryman. 2014. DNS Outage Post Mortem. Retrieved June 2020 from https://github.blog/2014-01-18-dns-outage-post-mortem/

[17] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. ACM, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

[18] Thomas Gleason. 2018. DNS Based Lever - An Untapped DevOps tool. https://community.akamai.com/customers/s/article/DNS-Based-Lever-An-Untapped-DevOps-tool?language=en_US.

[19] Google. 2020. Cloud DNS. https://cloud.google.com/dns/

[20] Scott Hilton. 2016. Dyn Analysis Summary Of Friday October 21 Attack. https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.

[21] Paul E. Hoffman, Andrew Sullivan, and Kazunori Fujiwara. 2019. DNS Terminology. RFC 8499. https://doi.org/10.17487/RFC8499

[22] Check Host. 2020. Check Host. http://check-host.net/check-dns

[23] Internet Initiative Japan Inc. 2019. *IP Location Load Balancing Resource Record*. Internet-Draft draft-sonoda-dnsop-lb-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-sonoda-dnsop-lb-01 Work in Progress.

[24] InfinityFree. 2019. DNS Outage at iFastNet: Softaculous down. Retrieved June 2020 from https://forum.infinityfree.net/t/dns-outage-at-ifastnet-softaculous-down/19374

[25] Infoblox 2020. https://www.infoblox.com/products/ddi/

[26] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. *Software Artifact for the SIGCOMM'20 Paper Titled "GRoot: Proactive Verification of DNS Configurations"*. https://doi.org/10.5281/zenodo.3905968 This work was supported in part by the National Science Foundation (NSF) under grants CNS-1704336 and CNS-1901510. The lead author was also supported by an internship from Microsoft Research.

[27] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2228298.2228311

[28] Nick Kephart. 2020. Best Practices for Monitoring DNS. https://www.thousandeyes.com/resources/dns-webinar

[29] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. Veriflow: Verifying Network-wide Invariants in Real Time. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Sept. 2012), 467–472. https://doi.org/10.1145/2377677.2377766

[30] Scott Kitterman. 2014. Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1. RFC 7208. https://doi.org/10.17487/RFC7208

[31] Murray Kucherawy, Dave Crocker, and Tony Hansen. 2011. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376. https://doi.org/10.17487/RFC6376

[32] Edward P. Lewis. 2006. The Role of Wildcards in the Domain Name System. RFC 4592. https://doi.org/10.17487/RFC4592

[33] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 499–512.

[34] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (Toronto, Ontario, Canada) *(SIGCOMM '11)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/2018436.2018470

[35] Microsoft. 2018. Description of the DNSLint utility. https://support.microsoft.com/en-us/help/321045/description-of-the-dnslint-utility

[36] Microsoft. 2020. Azure DNS. https://azure.microsoft.com/en-us/services/dns/

[37] P. Mockapetris. 1987. Domain names - concepts and facilities. RFC 1034. https://doi.org/10.17487/RFC1034

[38] P. Mockapetris. 1987. Domain names - implementation and specification. RFC 1035. https://doi.org/10.17487/RFC1035

[39] SecurityWeek News. 2010. Reports of Massive DNS Outages in Germany. Retrieved June 2020 from https://www.securityweek.com/content/reports-massive-dns-outages-germany

[40] Vasileios Pappas, Patrik Fältström, Daniel Massey, and Lixia Zhang. 2004. Distributed DNS troubleshooting. In *Proceedings of the ACM SIGCOMM workshop on Network troubleshooting: research, theory and operations practice meet malfunctioning reality*. 265–270.

[41] Vasileios Pappas, Zhiguo Xu, Songwu Lu, Daniel Massey, Andreas Terzis, and Lixia Zhang. 2004. Impact of Configuration Errors on DNS Robustness. *SIGCOMM Comput. Commun. Rev.* 34, 4 (Aug. 2004), 319 – 330. https://doi.org/10.1145/1030194.1015503

[42] Scott Rose and Wouter Wijngaards. 2012. DNAME Redirection in the DNS. RFC 6672. https://doi.org/10.17487/RFC6672

[43] Ao-Jan Su, David R Choffnes, Aleksandar Kuzmanovic, and Fabián E Bustamante. 2006. Drafting behind Akamai (travelocity-based detouring). *ACM SIGCOMM Computer Communication Review* 36, 4 (2006), 435–446.

[44] Liam Tung. 2019. Azure global outage: Our DNS update mangled domain records, says Microsoft. Retrieved June 2020 from https://www.zdnet.com/article/azure-global-outage-our-dns-update-mangled-domain-records-says-microsoft/

[45] Paul A. Vixie and Vernon Schryver. 2018. *DNS Response Policy Zones (RPZ)*. Internet-Draft draft-vixie-dnsop-dns-rpz-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-vixie-dnsop-dns-rpz-00 Work in Progress.

[46] Russ Wright and Martin Hamilton. 1997. Use of DNS Aliases for Network Services. RFC 2219. https://doi.org/10.17487/RFC2219

[47] Hongkun Yang and Simon S. Lam. 2016. Real-time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. Netw.* 24, 2 (April 2016), 887–900. https://doi.org/10.1109/TNET.2015.2398197

[48] Hongkun Yang and Simon S Lam. 2017. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking* 25, 5 (2017), 2900–2915.

[49] Dan York. 2015. HBO NOW DNSSEC Misconfiguration Makes Site Unavailable From Comcast Networks (Fixed Now). Retrieved June 2020 from https://www.internetsociety.org/blog/2015/03/hbo-now-dnssec-misconfiguration-makes-site-unavailable-from-comcast-networks-fixed-now/

[50] Bojan Zdrnja, Nevil Brownlee, and Duane Wessels. 2007. Passive monitoring of DNS anomalies. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 129–139.

[51] Help Net Security Zeljka Zorz, Managing Editor. 2013. LinkedIn outage was due to DNS records misconfiguration. Retrieved June 2020 from https://www.helpnetsecurity.com/2013/06/21/linkedin-outage-was-due-to-dns-records-misconfiguration/

Appendices are supporting material that has not been peer-reviewed.

APPENDIX A

A collection of resource records R (assumed to be IN class) is considered as a well-formed zone z if it satisfies all of the following conditions:

(1) There should be exactly one SOA record.
$|\{r \in R \mid \text{ty}(r) = \text{SOA}\}| = 1$

(2) No record can be a synthesized one.
$\langle d, t, \text{IN}, \tau, a, b\rangle \in R \implies b = 0$

(3) The domain name of the SOA record should be a prefix of the domain name of all the records in R.
$\langle d, \text{SOA}, \text{IN}, \tau, a, b\rangle \in R \wedge \langle d', t, \text{IN}, \tau', a', b'\rangle \in R \implies d \leq d'$

(4) The answer of a CNAME, DNAME and an NS record should be a valid domain name.
$\langle d, t, \text{IN}, \tau, a, b\rangle \in R \wedge t \in \{\text{CNAME}, \text{DNAME}, \text{NS}\} \implies a \in \text{DOMAIN}$

(5) There can be only one CNAME record for a domain name.
$\langle d, \text{CNAME}, \text{IN}, \tau, a, b\rangle \in R \wedge \langle d, \text{CNAME}, \text{IN}, \tau', a', b'\rangle \in R \implies \tau = \tau' \wedge a = a' \wedge b = b'$

(6) If there is a CNAME record for a domain name, then there cannot be any other record type for that domain name.
$\langle d, \text{CNAME}, \text{IN}, \tau, a, b\rangle \in R \wedge \langle d, t, \text{IN}, \tau', a', b'\rangle \in R \implies t = \text{CNAME}$

(7) There can be only one DNAME record for a domain name.
$\langle d, \text{DNAME}, \text{IN}, \tau, a, b\rangle \in R \wedge \langle d, \text{DNAME}, \text{IN}, \tau', a', b'\rangle \in R \implies \tau = \tau' \wedge a = a' \wedge b = b'$

(8) A domain name cannot have both DNAME and NS records unless there is an SOA record.
$\langle d, \text{DNAME}, \text{IN}, \tau, a, b\rangle \in R \wedge \langle d, \text{NS}, \text{IN}, \tau', a', b'\rangle \in R \implies \langle d, \text{SOA}, \text{IN}, \tau'', a'', b''\rangle \in R$

(9) If there is a DNAME record for a domain name d, then there cannot be any records for domain names for which d is a proper prefix.
$\langle d, \text{DNAME}, \text{IN}, \tau, a, b\rangle \in R \wedge \langle d', t, \text{IN}, \tau', a', b'\rangle \in R \wedge d \neq d' \implies d \not\leq d'$

(10) If there is an NS record for a domain name d but not an SOA record, then there cannot be any NS records for domain names for which d is a proper prefix.
$\langle d, \text{NS}, \text{IN}, \tau, a, b\rangle \in R \wedge \neg\exists\, \tau_s, a_s, b_s.\, \langle d, \text{SOA}, \text{IN}, \tau_s, a_s, b_s\rangle \in R \wedge \langle d', t, \text{IN}, \tau', a', b'\rangle \in R \wedge d < d' \implies t \neq \text{NS}$

(11) Wildcard domain names can not have a DNAME or an NS record.
$\langle d, t, \text{IN}, \tau, a, b\rangle \in R \wedge d[|d|] = * \implies t \neq \text{DNAME} \wedge t \neq \text{NS}$

(12) The set of resource records R are prefix-closed for the domain name of the zone *i.e.,* if there is a resource record whose domain name d is different from the domain name of the SOA record, then there has to be a resource record whose domain name d' is a proper prefix of d and is of length one less. (A real zone file can be made to satisfy this requirement by adding resource records for the empty non-terminals with the type N we introduced earlier.)
$\langle d, t, \text{IN}, \tau, a, b\rangle \in R \wedge \langle d_s, \text{SOA}, \text{IN}, \tau_s, a_s, b_s\rangle \in R \wedge d \neq d_s \implies \exists\, \langle d', t, \text{IN}, \tau, a, b\rangle \in R \wedge d' < d \wedge |d'| = |d| - 1$

APPENDIX B

First we introduce several helpful definitions. The first lets us more easily extract information from DNS answers:

Definition .1 (DNS answer extraction)**.** Given answer a, we write records(a) to refer the records in a, tag(a) for the record tag, and query(a) for the rewritten query (undefined if there is none). For example, if $a = \langle \text{AnsQ},\ \langle R,\ q\rangle\rangle$, then records($a$) = R, tag(a) = AnsQ, and query(a) = q. We lift each of these definitions to sets of answers, e.g., query(A) = $\{q \mid a \in A,\ q = \text{query}(a) \text{ is defined}\}$

Definition .2 (Real record extraction)**.** Given a set of resource records R, we extract those that are not synthesized with real(R) = $\{r \in R \mid \text{synth}(r) = 0\}$. This definition of real is lifted to DNS answers as: real(a) = $\langle \text{tag}(a), \text{real}(\text{records}(a))\rangle$ and to sets of answers pointwise: real(A) = $\{\text{real}(a), a \in A\}$.

Definition .3 (Equivalence modulo synthesis)**.** Given answer sets A_1 and A_2, we say the sets are equivalence modulo synthesis, written $A_1 \approx A_2$, if real(A_1) = real(A_2).

Theorem 5.1 (EC generation sound)**.** For a given configuration $C = \langle S, \Theta, \Gamma, \Omega\rangle$, if two queries q_1 and q_2 are in the same EC computed by the algorithm, then $q_1 \sim_C q_2$.

Proof. We assume that q_1 and q_2 are computed to be in the same EC, and we introduce variables a_i for a given server s:

$$\begin{aligned} a_1 &= \text{ServerLookup}(\Gamma(s), q_1) \\ a_2 &= \text{ServerLookup}(\Gamma(s), q_2) \end{aligned}$$

Given these assumptions, we must prove the following three conditions:

(1) $\mathcal{N}(\Gamma(s), q_1) = \mathcal{N}(\Gamma(s), q_2)$
(2) $a_1 \approx a_2$
(3) $q_1' \in \text{query}(a_1), q_2' \in \text{query}(a_2) \implies q_1' \sim_C q_2'$

Assume an arbitrary label graph generated by the EC generation algorithm. Each EC generated by the algorithm corresponds to a path through the label graph. Assume an arbitrary EC corresponding to path ρ through the label graph, where $q_1, q_2 \in \text{EC}(\rho)$. We note that q_1 and q_2 can only differ if the final label in the path is α.

Condition ($\mathcal{N}(\Gamma(s), q_1) = \mathcal{N}(\Gamma(s), q_2)$):
From the definition of $\mathcal{N}$, we must show:

$$\max_{\text{dn}}\{z \in \Gamma(s) \mid \text{dn}(z) \leq \text{dn}(q_1)\} = \max_{\text{dn}}\{z \in \Gamma(s) \mid \text{dn}(z) \leq \text{dn}(q_2)\}$$

The sets $\{z \in \Gamma(s) \mid \text{dn}(z) \leq \text{dn}(q_1)\}$ and $\{z \in \Gamma(s) \mid \text{dn}(z) \leq \text{dn}(q_2)\}$ select all zones where q_1 and q_2 are prefixes of the domain name. By virtue of q_1 and q_2 sharing the same path ρ, we now prove that these two sets are equivalent:

Case 1 (ρ does not end with α). In this case, q_1 and q_2 are the same query, and the equality is trivial.

Case 2 (ρ ends with α). In this case there are two possibilities. The first is that $q_i = \underbrace{\ldots \circ l_{k+1}}_{\alpha} \circ l_k \circ l_{k-1} \circ \ldots \circ \epsilon$ and $\text{dn}(z) = l_j \circ \ldots \circ \epsilon$ for $j \leq k$. In this case, we know that $\text{dn}(z) \leq q_1 \Leftrightarrow \text{dn}(z) \leq q_2$ since q_1 and q_2 have the same shared prefix. The other case is where $j > k$. In this case, we know that $\text{dn}(z)$ is given by the SOA record in the zone file, which means that $\text{dn}(z)$ will appear in the label tree. However, if this were the case, then we know that α is restricted such that $\alpha[0] = l_{k+1}$ is not equal to label $k+1$ in $\text{dn}(z)$. As such, $\text{dn}(z) \not\leq q_1$ and $\text{dn}(z) \not\leq q_2$.

Condition ($a_1 \approx a_2$):
By the definition of ServerLookup, and the fact that $\mathcal{N}(\Gamma(s), q_1) = \mathcal{N}(\Gamma(s), q_2)$ from before, there are now two cases. If $\mathcal{N}(\Gamma(s), q_1) = \emptyset$, then $a_1 = a_2 = \langle \text{Refused}, \emptyset \rangle$. Otherwise, we have $\mathcal{N}(\Gamma(s), q_1) = \{z\}$ for some z, and therefore:

$$\begin{aligned} a_1 &= \text{ZoneLookup}(z, q_1) \\ a_2 &= \text{ZoneLookup}(z, q_2) \end{aligned}$$

Expanding the definition of ZoneLookup, we get:

$$\begin{aligned} a_1 &= \text{RRLookup}(\{r \in \max_{<_{q_1,z}} z\}, q_1, z) \\ a_2 &= \text{RRLookup}(\{r \in \max_{<_{q_2,z}} z\}, q_2, z) \end{aligned}$$

The inner set $\{r \in \max_{<_{q_1,z}} z\}$ selects the resource records that are a closest match to the query q_1 and similarly for q_2. These two sets *must* be equal for the same reasons as in the proof of the first condition. In other words, if two records can distinguish between q_1 and q_2 in α, then α would have excluded the domains of those records. Specifically, it must be that $\text{Rank}(r, q_1, z) = \text{Rank}(r, q_1, z)$. This can be shown by showing that each component of the Rank functions are equivalent.

The first components $\text{Match}(r, q_1) = \text{Match}(r, q_2)$ must be true since $\text{dn}(r) \leq \text{dn}(q_1) \iff \text{dn}(r) \leq \text{dn}(q_2)$ since $\text{dn}(r)$ cannot equal $\text{dn}(q_1)$ or $\text{dn}(q_2)$ (or else they would be in different ECs). Hence $\text{dn}(r)$ can only be a prefix of both $\text{dn}(q_1)$ and $\text{dn}(q_2)$. Similarly, if $\text{dn}(q_i) \in_* \text{dn}(r)$, then $|\text{dn}(r)| \leq |\text{dn}(q_i)|$. Again assume $q_i = \underbrace{\ldots \circ l_{k+1}}_{\alpha} \circ l_k \circ l_{k-1} \circ \ldots \circ \epsilon$ and $\text{dn}(r) = l_j \circ \ldots \circ \epsilon$. If $j \leq k$, then $\text{dn}(q_1) \in_* \text{dn}(r) \Leftrightarrow \text{dn}(q_2) \in_* \text{dn}(r)$. If $j > k+1$, then $\text{dn}(r)$ would be in the label graph and α would exclude l_{k+1} ($\alpha[0] \neq l_{k+1}$). If $j = k+1$, then it must be that $l_j = *$, in which case both q_i match the wildcard for $\text{dn}(r)$.

The second and fourth components of Rank do not depend on the query value and are thus the same. The third components must also be equal since $\text{dn}(q_1)$ and $\text{dn}(q_2)$ share the same prefix (except their last label) and $\text{dn}(r)$ cannot share a label in this last position with either query since this would have caused q_1 and q_2 to be separated into different ECs.

Note that if a record r is an exact match ($\text{dn}(r) = \text{dn}(q_i)$), then it must be that $q_1 = q_2$, since otherwise the labels of r would be in the label graph, and thus q_1 would not be placed in the same EC as q_2.

Continuing, we then have a set R such that:

$$\begin{aligned} a_1 &= \text{RRLookup}(R, q_1, z) \\ a_2 &= \text{RRLookup}(R, q_2, z) \end{aligned}$$

We continue by case analysis on the execution of RRLookup for q_1.

Case ($\text{dn}(R) = \text{dn}(q_1)$). This is an exact match. As just stated, it must then be that $q_1 = q_2$. and so the equality trivially holds.

Case ($\text{dn}(q_1) \in_* \text{dn}(R)$). In this case, the matching record(s) are wildcard records. From before, we know that $\text{dn}(q_2) \in_* \text{dn}(R)$. We therefore get the following:

$$\begin{aligned} a_1 &= \text{WildcardMatch}(R, q_1, \{\text{ty}(r) \mid r \in R\}) \\ a_2 &= \text{WildcardMatch}(R, q_2, \{\text{ty}(r) \mid r \in R\}) \end{aligned}$$

There are now three cases for how WildcardMatch can evaluate. We know that q_1 and q_2 have the same type by how the algorithm generates ECs. If the types are equal:

$$\begin{aligned} a_1 &= \langle \textsc{Ans},\ \textsc{Syn}(\mathcal{T}(R, \text{ty}(q_1)), \text{dn}(q_1))\rangle \\ a_2 &= \langle \textsc{Ans},\ \textsc{Syn}(\mathcal{T}(R, \text{ty}(q_2)), \text{dn}(q_2))\rangle \end{aligned}$$

Expanding the definition of Syn:

$$\begin{aligned} a_1 &= \langle \textsc{Ans},\ \mathcal{T}(R, \text{ty}(q_1)) \cup \{\langle d, t, \texttt{IN}, \tau, a, 1\rangle \mid \exists\ d',\ \langle d',\ t, \texttt{IN}, \tau, a, 0\rangle \in \mathcal{T}(R, \text{ty}(q_1))\}\rangle \\ a_2 &= \langle \textsc{Ans},\ \mathcal{T}(R, \text{ty}(q_2)) \cup \{\langle d, t, \texttt{IN}, \tau, a, 1\rangle \mid \exists\ d',\ \langle d',\ t, \texttt{IN}, \tau, a, 0\rangle \in \mathcal{T}(R, \text{ty}(q_2))\}\rangle \end{aligned}$$

Since we must show that $a_1 \approx a_2$, we compute:

$$\begin{aligned} & \text{real}(a_1) \\ = \ & \langle \textsc{Ans},\ \text{real}(\mathcal{T}(R, \text{ty}(q_1)) \cup \{\langle q_1, t, \texttt{IN}, \tau, a, 1\rangle \mid \exists\ d',\ \langle d',\ t, \texttt{IN}, \tau, a, 0\rangle \in \mathcal{T}(R, \text{ty}(q_1)))\rangle \\ = \ & \langle \textsc{Ans},\ \text{real}(\mathcal{T}(R, \text{ty}(q_1)))\rangle \\ = \ & \langle \textsc{Ans},\ \text{real}(\mathcal{T}(R, \text{ty}(q_2)))\rangle \\ = \ & \text{real}(a_2) \end{aligned}$$

In the second case, we have $\text{ty}(q_1) \notin T, \texttt{CNAME} \in T, R = \{r\}$. Again we assume the types are equal, so we have $\text{ty}(q_1) \notin T$ and $\text{ty}(q_1) = \text{ty}(q_2)$ and it follows that $\text{ty}(q_2) \notin T$. Therefore, q_2 will evaluate to the same case, giving us:

$$\begin{aligned} a_1 &= \langle \textsc{AnsQ},\ \textsc{Syn}(R, \text{dn}(q_1)), \langle \text{ans}(q_1),\ \text{ty}(q_1)\rangle\rangle \\ a_2 &= \langle \textsc{AnsQ},\ \textsc{Syn}(R, \text{dn}(q_2)), \langle \text{ans}(q_2),\ \text{ty}(q_2)\rangle\rangle \end{aligned}$$

As before, we compute real:

$$\begin{aligned} \text{real}(a_1) &= \langle \textsc{AnsQ},\ \text{real}(\textsc{Syn}(R, \text{dn}(q_1)))\rangle \\ \text{real}(a_2) &= \langle \textsc{AnsQ},\ \text{real}(\textsc{Syn}(R, \text{dn}(q_2)))\rangle \end{aligned}$$

And then

$$\begin{aligned} \text{real}(a_1) &= \langle \textsc{AnsQ},\ \text{real}(R)\rangle \\ \text{real}(a_2) &= \langle \textsc{AnsQ},\ \text{real}(R)\rangle \end{aligned}$$

Which gives the desired result.

In the final case, for WildcardMatch we trivially have $a_1 = \langle \textsc{Ans},\ \emptyset\rangle = a_2$.

Case ($\text{dn}(R) < \text{dn}(q_1),\ \texttt{DNAME} \in T$). In this case there is a single DNAME record ($R = \{r\}$). Given that q_1 and q_2 share the same prefix, it must be the case that $\text{dn}(R) < \text{dn}(q_2)$ Therefore we get the same case for q_2. We compute:

$$\begin{aligned} a_1 &= \textsc{Rewrite}(\{r\}, q_1) \\ a_2 &= \textsc{Rewrite}(\{r\}, q_2) \end{aligned}$$

Expanding the definition of Rewrite:

$$\begin{aligned} a_1 &= \langle \textsc{AnsQ},\ \textsc{DProc}(\mathcal{T}(\{r\}, \texttt{DNAME}), q_1)\rangle \\ a_2 &= \langle \textsc{AnsQ},\ \textsc{DProc}(\mathcal{T}(\{r\}, \texttt{DNAME}), q_2)\rangle \end{aligned}$$

Unfolding the definition of DProc, we get:

$$\begin{aligned} a_1 &= \langle \textsc{AnsQ},\ \langle \{r\} \cup \{\langle \text{dn}(q_1), \texttt{CNAME}, \texttt{IN}, \text{ttl}(r), \text{dn}(q_1)[\text{dn}(r) \mapsto \text{ans}(r)], 1\rangle\},\ \langle \text{dn}(q_1)[\text{dn}(r) \mapsto \text{ans}(r)],\ \text{ty}(q_1)\rangle\rangle\rangle \\ a_2 &= \langle \textsc{AnsQ},\ \langle \{r\} \cup \{\langle \text{dn}(q_2), \texttt{CNAME}, \texttt{IN}, \text{ttl}(r), \text{dn}(q_2)[\text{dn}(r) \mapsto \text{ans}(r)], 1\rangle\},\ \langle \text{dn}(q_2)[\text{dn}(r) \mapsto \text{ans}(r)],\ \text{ty}(q_2)\rangle\rangle\rangle \end{aligned}$$

Applying the definition of real, we drop the synthesized records:

$$\text{real}(a_1) = \langle \textsc{AnsQ},\ \text{real}(\{r\})\rangle = \text{real}(a_2)$$

Case ($\text{dn}(R) < \text{dn}(q_1),\ \texttt{DNAME} \notin T, \texttt{NS} \in T, \texttt{SOA} \notin T$). As in the previous case, we know that $\text{dn}(R) < \text{dn}(q_2)$. It follows that q_2 will also match this case. We compute:

$$a_1 = \textsc{Delegation}(R, z) = a_2$$

Case (otherwise). This case is trivial, since q_2 must also fall into this case since it matched the same conditions for all other cases. As such, then we get $a_1 = \langle \textsc{Ans},\ \emptyset\rangle = a_2$.

Condition ($q_1' \in \text{query}(a_1), q_2' \in \text{query}(a_2) \implies q_1' \sim_C q_2'$): The final condition we must prove is for rewrites. There are two possible ways a rewrite can happen: a DNAME or CNAME record. The proof follows the exact structure as in the previous condition, except we show only these two cases since any other records with result in $\text{query}(a_i) = \emptyset$.

Case ($\text{dn}(R) = \text{dn}(q_1), \textsc{Authoritative}(T), \text{ty}(q_1) \notin T, \texttt{CNAME} \in T, R = \{r\}$). This is the ExactMatch case for a CNAME record. As before, we observe that $q_1 = q_2$, so the property is trivially satisfied.

Case ($\text{dn}(q_1) \in_* \text{dn}(R), \text{ty}(q_1) \notin T, \texttt{CNAME} \in T, R = \{r\}$). This is the WildcardMatch case for a CNAME record. As before, we observe that $\text{dn}(q_2) \in_* \text{dn}(R)$, so q_2 will execute in the same case. We have $\text{query}(a_1) = \{\langle \text{ans}(r),\ \text{ty}(q_1)\rangle\} = \{\langle \text{ans}(r),\ \text{ty}(q_2)\rangle\} = \text{query}(a_2)$, so the property holds since CNAME simply rewrites to a fixed new query.

Case ($\text{dn}(R) < \text{dn}(q_1), \texttt{DNAME} \in T$). This is the Rewrite case for a DNAME record. As before, we observe that $\text{dn}(R) < \text{dn}(q_2)$, so q_2 will execute in the same case. Unfolding the definition of DProc, we have:

$$\begin{array}{lcl} \text{query}(a_1) & = & \{\langle \text{dn}(q_1)[\text{dn}(r) \mapsto \text{ans}(r)],\ \text{ty}(q_1)\rangle\} \\ \text{query}(a_2) & = & \{\langle \text{dn}(q_2)[\text{dn}(r) \mapsto \text{ans}(r)],\ \text{ty}(q_2)\rangle\} \end{array}$$

For this DNAME case, we know that $\text{dn}(q_i)$ (represented by path ρ) are prefixes of $\text{dn}(r)$. Suppose that $q_1 = \underbrace{\ldots \circ l_k}_{\alpha} \circ l_{k-1} \circ \ldots \circ \epsilon$ and $q_2 = \underbrace{\ldots \circ l'_k}_{\alpha} \circ l_{k-1} \circ \ldots \circ \epsilon$, and that $\text{dn}(r) = l''_j \circ l''_{j-1} \circ \ldots \circ \epsilon$ where $j < k$ and $l_i = l''_i$. Further, suppose that $\text{ans}(r)$ is given by the target domain name ρ' The rewritten queries will be $q'_1 = \ldots \circ l_k \circ l_{k-1} \circ \ldots \circ \rho'$ and $q'_2 = \ldots \circ l'_k \circ l_{k-1} \circ \ldots \circ \rho'$. Since we always add the target of a DNAME record to the label graph, path ρ' will be a path that exists in the label graph. Moreover, there will be a dashed edge from the node representing path $\text{dn}(r)$ to a node corresponding to ρ'. We will show that q'_1 and q'_2 now belong to the same label graph path. Since q_1 and q_2 could only have been in the same EC if ρ ended in α in the label graph, and since by construction this α excluded all possible subdomains for extensions of ρ' after the rewrite, we know that the path matching q'_1 and q'_2 must end in α. Since they match the same path, we conclude that $q'_1 \sim_C q'_2$. □

Theorem 5.2 (Soundness)**.** For all C, q_1, q_2, and k, if $q_1 \sim_C q_2$, then $\textsc{Resolve}(q_1, C, k) \approx \textsc{Resolve}(q_2, C, k)$.

Proof. From Resolve, we must show:

$$\bigcup_{s \in \Theta} \textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, k) \approx \bigcup_{s \in \Theta} \textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, k)$$

In particular, we prove a stronger inductive invariant:

$$\forall C, q_1, q_2, s, i.\ q_1 \sim_C q_2 \implies \textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i) \approx \textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i)$$

which then implies this equality. The proof proceeds by induction on the resolution step i.

Base case ($i = 0$) trivial since we have

$$\begin{array}{l} \text{real}(\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, 0)) = \text{real}(\{\textsc{ServFail}, \emptyset\}) = \{\textsc{ServFail}, \emptyset\} \\ \text{real}(\textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, 0)) = \text{real}(\{\textsc{ServFail}, \emptyset\}) = \{\textsc{ServFail}, \emptyset\} \end{array}$$

Inductive case (i) We must prove that

$$\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i) \approx \textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i)$$

First, we observe that if $s = \bot$, then both the left and right hand sides evaluate to $\{\textsc{ServFail}, \emptyset\}$ as in the base case.
There are now three cases for how $\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i)$ may evaluate. We consider each in turn:

Case 1 ($\textsc{ServerLookup}(\Gamma(s), q_1) = \langle \textsc{AnsQ}, \langle R,\ q'_1\rangle\rangle$).
From the assumption of $q_1 \sim_C q_2$ we know that

$$\textsc{ServerLookup}(\Gamma(s), q_1) \approx \textsc{ServerLookup}(\Gamma(s), q_2)$$

Substituting on the left, we get:

$$\textsc{ServerLookup}(\Gamma(s), q_2) \approx \langle \textsc{AnsQ}, \langle R,\ q'_1\rangle\rangle$$

Expanding the definition of $\approx$, we get

$$\text{real}(\textsc{ServerLookup}(\Gamma(s), q_2)) = \text{real}(\langle \textsc{AnsQ}, \langle R,\ q'_1\rangle\rangle)$$

Simplifying on the right:

$$\text{real}(\textsc{ServerLookup}(\Gamma(s), q_2)) = \langle \textsc{AnsQ}, \text{real}(R)\rangle$$

This equality can only hold if: $\textsc{ServerLookup}(\Gamma(s), q_2) = \langle \textsc{AnsQ}, \langle R',\ q'_2\rangle\rangle$ and also $\text{real}(R') = \text{real}(R)$. We note that from the assumption of $q_1 \sim_C q_2$, we know that $q'_1 \sim_C q'_2$. This also implies that $\mathcal{N}(\Gamma(s), q'_1) = \mathcal{N}(\Gamma(s), q'_1)$

There are now two cases. In the first case we have $\mathcal{N}(\Gamma(s), q'_1) = \emptyset$, which implies $\mathcal{N}(\Gamma(s), q'_2) = \emptyset$ from the assumption $_C$, and in the second case we have $\mathcal{N}(\Gamma(s), q'_1) \neq \emptyset$ which implies $\mathcal{N}(\Gamma(s), q'_2) \neq \emptyset$. Both cases are proved the same way, so we show one ($\mathcal{N}(\Gamma(s), q'_1) = \emptyset$).

Since both cases will resolve using the AnsQ case, we can compute

$$\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i) = \textsc{Resolve}(q_1', \langle S, \Theta, \Gamma, \Omega\rangle, i-1)$$
$$\textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i) = \textsc{Resolve}(q_2', \langle S, \Theta, \Gamma, \Omega\rangle, i-1)$$

therefore, we have:

$$\text{real}(\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i)) = \text{real}(\textsc{Resolve}(q_1', \langle S, \Theta, \Gamma, \Omega\rangle, i-1))$$
$$\text{real}(\textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i)) = \text{real}(\textsc{Resolve}(q_2', \langle S, \Theta, \Gamma, \Omega\rangle, i-1))$$

From the inductive hypothesis, and the fact that $q_1' \sim_C q_2'$, then we can conclude:

$$\textsc{Resolve}(q_1', \langle S, \Theta, \Gamma, \Omega\rangle, i-1) \approx \textsc{Resolve}(q_1', \langle S, \Theta, \Gamma, \Omega\rangle, i-1)$$

and we can finally prove the desired result:

$$\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i) \approx \textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i)$$

Case 2 ($\textsc{ServerLookup}(\Gamma(s), q_1) = \langle \textsc{Ref}, R\rangle$)
From the assumption of $q_1 \sim_C q_2$ we know that

$$\textsc{ServerLookup}(\Gamma(s), q_1) \approx \textsc{ServerLookup}(\Gamma(s), q_2)$$

Substituting on the left, we get:

$$\textsc{ServerLookup}(\Gamma(s), q_2) \approx \langle \textsc{Ref}, R\rangle$$

Expanding the definition of $\approx$, we get

$$\text{real}(\textsc{ServerLookup}(\Gamma(s), q_2)) = \text{real}(\langle \textsc{Ref}, R\rangle)$$

Simplifying on the right:

$$\text{real}(\textsc{ServerLookup}(\Gamma(s), q_2)) = \langle \textsc{Ref}, \text{real}(R)\rangle$$

This equality can only hold if: $\textsc{ServerLookup}(\Gamma(s), q_2) = \langle \textsc{Ref}, R'\rangle$ and also $\text{real}(R') = \text{real}(R)$.

Since both cases will resolve using the Ref case, we can compute therefore, we have:

$$\text{real}(\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i)) = \text{real}(\textstyle\bigcup_{r \in \mathcal{T}(\text{real}(R), \text{NS})} \textsc{Resolve}(\Omega(\text{ans}(r)), q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i-1))$$
$$\text{real}(\textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i)) = \text{real}(\textstyle\bigcup_{r \in \mathcal{T}(\text{real}(R'), \text{NS})} \textsc{Resolve}(\Omega(\text{ans}(r)), q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i-1))$$

from the definition of real, we can distribute over set union:

$$\text{real}(\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i)) = \textstyle\bigcup_{r \in \mathcal{T}(\text{real}(R), \text{NS})} \text{real}(\textsc{Resolve}(\Omega(\text{ans}(r)), q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i-1))$$
$$\text{real}(\textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i)) = \textstyle\bigcup_{r \in \mathcal{T}(\text{real}(R'), \text{NS})} \text{real}(\textsc{Resolve}(\Omega(\text{ans}(r)), q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i-1)))$$

From the inductive hypothesis, and the fact that $q_1 \sim_C q_2$, then we can conclude that for each $r \in \text{real}(R) = \text{real}(R')$:

$$\textsc{Resolve}(\Omega(\text{ans}(r)), q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i-1) \approx \textsc{Resolve}(\Omega(\text{ans}(r)), q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i-1)$$

Since the components are pointwise equal, the set unions are also equal, so we obtain the desired result:

$$\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i) \approx \textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i)$$

Case 3 (otherwise)
The final case is immediate from the assumption of $q_1 \sim_C q_2$. In particular, this means:

$$\{\textsc{ServerLookup}(\Gamma(s), q_1)\} \approx \{\textsc{ServerLookup}(\Gamma(s), q_2)\}$$

and since real is applied pointwise over sets:

$$\textsc{ServerLookup}(\Gamma(s), q_1) \approx \textsc{ServerLookup}(\Gamma(s), q_2)$$

and by the definition of Resolve:

$$\textsc{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega\rangle, i) \approx \textsc{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega\rangle, i)$$

□

Theorem 5.3 (Completeness)**.** For a configuration C, each query q belongs to at least one computed equivalence class.

Proof. The proof is straightforward: Assume we are given an arbitrary query q. We must prove that q belongs to some equivalence class. In particular, we simply pick the path through the label graph that shares the longest matching prefix with $\text{dn}(q)$. If the longest matching path is an exact match, then we are done since we generate an EC for each type for that exact domain name $\text{dn}(q)$. If however, there is not an exact match, the we select that last label in common with $\text{dn}(q)$, which will have an α child. This child, by construction, will match any domain name not already matched by a sibling or a child of some rewrite along the same path. □

Theorem 5.4 (Linear time)**.** In the absence of DNAME records, for a collection of zone files with n resource records, our algorithm computes $O(n)$ equivalence classes in $O(n)$ time.

Proof. Without DNAME records, the label graph is a tree, and hence the number of paths in the tree is equal to the number of nodes in the tree. The number of nodes in the tree is at most 127 * n, since each record can have at most 127 labels in it. Since we generate, at most, $|T|$ (constant number) equivalence classes for each path, there are at most O(n) ECs. To build the label graph, we add each of the n records to the tree. Since each domain name in a record can have at most 127 labels, adding the domain name to the tree involves walking through at most 127 levels of the tree to find where to add the new labels for the domain name. At each level, we find if there is a matching label by using a hash table with amortized constant time lookup. So each insertion takes constant bounded time, and there are n insertions. □