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

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 resolver. 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...
identifying bugs in the network ranging from delegation

GRoot revealed 109 new bugs and completed in under 10 seconds.

efficient symbolic execution in the given DNS configuration.

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

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 264.58.233.75.

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., 2) 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 2 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 \(<\text{support.mybankcard.com}, A>\), 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 \{ 2, 3 \}. 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 2 (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 7. A DNAME record performs a query rewrite, in this case to redirect the user to bankcard.com. Specifically, 2 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
email.bankcard.com, but not for support.bankcard.com. As such, the query will match the wildcard record (4). 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 284.58.233.244.

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.fnii.net instead of ns1.fnii.com, then after several steps DNS would match the query with the wildcard record (4). The CNAME record type (canonical name) performs a rewrite without preserving the query suffix, so the query becomes www.support.bankcard.com and finally matches record (4), which provides the IP address 284.58.233.75, 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 · and terminated by the special symbol ε, which represents an empty string (e.g., foo.com is written as foo · com 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 = l0 · · · ln where ln = ε, we write |d| to denote the index of the last label k, and we use the indexing notation d[i] to select label li. We denote the set of valid domain names by the set: DOM.

Figure 2 shows a number of definitions that we use to define the behavior of DNS. Specifically, we use the notation d1 ∼j d2 (1) to mean that domains d1 and d2 share a common prefix of j labels (not counting ε). For example, foo · com · ε ≃ 1 com ε. To select the maximal j such that d1 ∼j d2, we write maxj(d1,d2) (2). We use the definition of ∼j to introduce a partial ordering among domain names (3) that orders them by longest match. In particular, d1 ≤ d2 iff d1 is a prefix of d2 (d1 ∼|d2| d2). We also use the notation d1 ∈ d2 to mean that d1 matches the wildcard domain d2 (4).

Zones and resource records. A DNS zone z ∈ ZONE is a set of resource records (ZONE = ℘(RECORD)). We use the symbol ℘(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
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 \( r \in \text{record} = (d, t, c, \tau, a, b) \) as a tuple with six components: (1) a domain name \( d \in \text{domain} \), (2) a record type \( t \in \text{type} = \{A, AAAA, NS, DNSNAME, CNAME, SOA, ... \} \cup \{N\} \) 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 \( c = IN \) for the internet, (4) the time-to-live value for the record \( r \in \text{N} \) 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 \( \text{syn}(r) \) for whether the record was synthesized.

**DNS queries.** A DNS query \( q = (d, t) \) 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 (www.mybankcard.com, A) 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. \text{Wildcard} (5) determines if a record has a wildcard domain, and \text{Match} (6) determines if a record is relevant for a given query (i.e., a potential match). The \text{Rank} (7) function for a record, query, and zone returns a tuple of integer values; the indicator function \( I \) returns 1 if the predicate is true and 0 otherwise. The \text{Rank} function then induces a strict partial order \( <_{q, z} \) on records (8) by comparing the resulting tuples lexicographically from left to right \((< )\). 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 \( a = (x, y) \) 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 \( \text{Ans} \), a delegation \( \text{Ref} \), a rewrite \( \text{AnsQ} \), etc.), and data \( y \), which is a set of resource records \( R \) holding pertinent information when \( x \neq \text{AnsQ} \) and is a pair \((R, q')\) 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, \( C = (S, \Theta, \Gamma, \Omega) \), called a configuration \( C \), where:

- \( S \) is a set of nameservers (e.g., ns1.fnn1.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 \( S \).
The first step is to find the zone \( \Gamma \) that names the zone for which that nameserver is authoritative.

- \( \Omega : D \rightarrow S \cup \{\bot\} \) 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 \texttt{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 (dn \( z \)) with the domain in the query (dn \( q \)) — the function \( N \). The notation \( \text{max}_{dn} \) 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 \( \text{com} \) and \( \text{gmail.com} \) and the user’s query is for \( \text{help.gmail.com} \), then the nameserver will choose the \( \text{gmail.com} \) zone to answer the query. If there is such a matching zone \( z \), then \texttt{ServerLookup} calls \texttt{ZoneLookup} to get an answer by evaluating the query against the zone file. Otherwise, the nameserver refuses (\texttt{Refused}) to perform the lookup operation as it could not find a relevant zone.

\texttt{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 \texttt{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 \texttt{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 \texttt{RRLookup} is to return either \( 1 \) an answer (\texttt{Ans}), if the resource records \( R \) are sufficient to answer the query \( q \), \( 2 \) a referral (\texttt{Ref}), if records \( R \) cannot answer the query \( q \) but indicate who might have the answer, \( 3 \) an intermediate answer \( r' \) and a query \( q' \) (\texttt{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 (\texttt{NX}), indicating that the domain does not exist.

\texttt{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 \( N \) stated earlier. When the records’ domain name exactly matches the query, the \texttt{ExactMatch} function is applied (SA, steps 3A, 3B). Otherwise, if it is a wildcard domain that matches the query domain, the \texttt{WildcardMatch} case will apply (SA, step 3C). If the records contain a matching \texttt{NAME} record, which is only possible when the other two cases do not apply, then the query will be modified according to the \texttt{Rewrite} function (SA, step 3C). If no such record exists, DNS will delegate the query to another nameserver if it has an \texttt{NS} record (\texttt{Delegation}) (SA, step 3B). Finally, if all else fails, the nameserver will return \texttt{NXDOMAIN} (non-existent domain, SA, step 3C).

The \texttt{ExactMatch} and \texttt{WildcardMatch} cases are both broken down further into several cases. For the \texttt{ExactMatch} 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 \texttt{RRLookup}, a zone \( z \) is authoritative of all the records except for \texttt{NS} records (zone cut) not accompanied by an \texttt{SOA} record. Otherwise, if there is a \texttt{NAME} record (SA, step 3A), then the nameserver will perform a rewrite (\texttt{AnsQ}), returning the relevant records \( R \), as well as a new query domain given by \texttt{ans}(\( q \)) with the same type (\( ty(\text{dn}(R)) \)). If there is no \texttt{NAME} record, but there is a non-authoritative \texttt{NS} record, then the nameserver will perform a \texttt{Delegation} (SA, step 3B). Finally, if all else fails, it will simply return an answer with no information (\texttt{0}).

The \texttt{WildcardMatch} case is similar to the \texttt{ExactMatch} case, except it will perform synthesis (\texttt{SYN}) to generate a new set of records specializing the wildcards. For instance, a lookup for a query with domain \texttt{ema11.com} on a set with a single wildcard record \( * \cdot \text{com} \) generates a (cachable) synthesized record for \texttt{ema11.com}.

The \texttt{Rewrite} case for \texttt{NAME} records returns \texttt{AnsQ} with records and a new, rewritten query. The new query is given by \texttt{DProc}, which generates and adds a new synthesized \texttt{NAME} record for the answer and substitutes the matching prefix of the query with the rewrite described in the record answer (\texttt{dn}(\( q \))(\text{dn}(\( r \)) \rightarrow \text{ans}(\text{\( r \ })))). The DNS adds these \texttt{NAME} records to the answer to facilitate caching — future queries are rewritten based on the cached \texttt{NAME} record (SA, step 3C). The \texttt{Delegation} case returns the \texttt{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 \texttt{Resolve} that return a set of possible answers. The functions return sets of answers in order to capture the nondeterminism inherent in
We leverage the model in §3 to present a fast verification algorithm with no records. Otherwise, if works by resolving the query DNAME (15)

ψ

\textit{GRoot: Proactive Verification of DNS Configurations}

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

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 DNS redirection of (f).

DNS. The first function takes a query q, a configuration (S, Θ, Γ, Ω), 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 ∈ Θ, taking the union of their results.

The second \texttt{Resolve} function performs resolution at a specific nameserver s. There are several cases based on the result of \texttt{ServerLookup}. In the first case, if the resolver has already exceeded the execution bound k or the input is a . due to a nameserver lookup failure in Ω from a recursive call of \texttt{Resolve}, it returns \texttt{ServFail} with no records. Otherwise, if s returns a rewrite \texttt{AnsQ}, and does not have a local zone that can process the rewrite (N(T(s), g) = ∅), 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 \texttt{Ref}, then the function unions the results from nondeterministically resolving the query at each nameserver identified in the returned NS records (∏(ans(r))). Finally, if \texttt{ServerLookup} returns any other kind of answer, \texttt{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 by the same 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 ε, 𝑡 such that lₖ 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 ∈ TYPE:

(1) (ε, t)
(2) (com ∗ ε, t)
(3) (bankcard ∗ com ∗ ε, t)
(4) (www ∗ bankcard ∗ com ∗ ε, t)
(5) (α ∗ www ∗ bankcard ∗ com ∗ ε, t)
(6) (email ∗ bankcard ∗ com ∗ ε, t)
(7) (α ∗ email ∗ bankcard ∗ com ∗ ε, t)

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) (∗ bankcard ∗ com ∗ ε, t)
(9) (∗ bankcard ∗ com ∗ ε, t)
(10) (α bankcard ∗ com ∗ ε, t) α[0] ∉ {www, 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) (mybankcard ∗ com ∗ ε, t)
(12) (www mybankcard ∗ com ∗ ε, t)
(13) (α www mybankcard ∗ com ∗ ε, t)
(14) (email mybankcard ∗ com ∗ ε, t)
(15) (α email mybankcard ∗ com ∗ ε, t)
(16) (∗ mybankcard ∗ com ∗ ε, t)
(17) (α ∗ ∗ mybankcard ∗ com ∗ ε, t)
The algorithm would similarly continue to explore paths for fnni.com and terminates with the EC, \((\alpha \circ \text{mybankcard} \circ \text{com} \circ e, t)\)

\[\alpha[0] \not\in \{\text{www, email, *}\}\]

**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 email1. After generating the EC given by (14), it would take the dotted edge back to mybankcard and then the dotted edge back to bankcard. If then traverses the paths underneath bankcard but with the original query prefix before rewriting, so it will generate \((\text{www} \circ \text{email} \circ \text{mybankcard} \circ \text{com} \circ e, t)\), \((\alpha \circ \text{www} \circ \text{email} \circ \text{mybankcard} \circ \text{com} \circ e, t)\), 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 bankcard \circ \text{com} \circ e \at \text{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 \(\alpha\), we observe that by construction \(\alpha\) cannot match any of the records present at a given zone file except a wildcard. Therefore, we can leave \(\alpha\) 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: \((\alpha \circ \text{mybankcard} \circ \text{com} \circ e, [\text{MX, TXT}])\). 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 \(\{\text{MX, TXT}\}\) 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.

### 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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delegation Inconsistency</td>
<td>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.</td>
</tr>
<tr>
<td>Lame Delegation</td>
<td>Interpretation graph has a node with the Refused tag.</td>
</tr>
<tr>
<td>Missing Glue Records</td>
<td>Node answer contains NS records but not the A (glue) records.</td>
</tr>
<tr>
<td>Non-Existing Domain for Service</td>
<td>Sink node return an answer with the NX tag.</td>
</tr>
<tr>
<td>Cyclic Zone Dependency</td>
<td>Interpretation graph contains a cycle.</td>
</tr>
<tr>
<td>Rewriting Loop</td>
<td>Interpretation graph contains a cycle with at least one rewrite.</td>
</tr>
<tr>
<td>Query Exceeds Maximum Length</td>
<td>Query at some node exceeds the maximum label or total length.</td>
</tr>
<tr>
<td>Answer Inconsistency</td>
<td>Different sink nodes return different answers.</td>
</tr>
<tr>
<td>Zero Time To Live</td>
<td>Sink node returns an answer with TTL value set to 0.</td>
</tr>
<tr>
<td>Rewrite Blackholing</td>
<td>A path has a rewrite and ends at a node with NX tag.</td>
</tr>
</tbody>
</table>

Table 2: Bug finding Implementation for Table 1.
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 $C = (S, \Theta, \Gamma, \Omega)$, 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 $a_i = \text{SERVERLOOKUP}(\Gamma(s), q_i)$, we have (1) $N(\Gamma(s), q_1) = N(\Gamma(s), q_2)$, (2) $a_1 \equiv a_2$, 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 $C = (S, \Theta, \Gamma, \Omega)$, 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 Resolve($q_1$, $C$, $k$) $\approx$ Resolve($q_2$, $C$, $k$).

**Proof.** We start by proving a slightly stronger invariant: Resolve($s$, $q_1$, $S$, $\Theta$, $\Gamma$, $\Omega$, $i$) $\approx$ Resolve($s$, $q_2$, $S$, $\Theta$, $\Gamma$, $\Omega$, $i$) 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 DNAME rewrites are comparatively much more common that CNAMEs 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 secondaryed their zones back to the four campus nameservers, which also provide authoritative answers for queries related to those subdomains. The remaining 953 subdomains require delegation via NS records.

We use the campus.edu zone file and the zone files of the 895 subdomains that are secondaryed 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."
We do not actively black-hole server DNS entries. CNAME entries that 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

dc1.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 NULLDOMAIN. 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 Cloudflare and are likely intentional. But one name server was a.campus.edu 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
The more the number of subdomains and the larger the graphs built, verify the correctness of their DNS configuration (zone) files, or 17 nodes. The median graph size was also around 6 nodes. Many operators today use blackbox techniques for DNS testing. GRoot is the first tool that allows operators to check 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:
DNN 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.

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:
DNN 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).

DNN 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.

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

[42] Zeljka Zorz, Managing Editor. 2013. LinkedIn outage was due to DNS records misconfiguration. Retrieved June 2020 from https://www.zdnet.com/article/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.
   \( (d, t, IN, \tau, a, b) \in R \implies b = 0 \)

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

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

5. There can be only one DNAME record for a domain name.
   \( (d, \text{DNAME}, IN, \tau, a, b) \in R \land \langle d', \text{CNAME}, IN, \tau', a', b' \rangle \in R \implies t = \text{CNAME} \)

6. There can be only one DNAME record for a domain name.
   \( (d, \text{DNAME}, IN, \tau, a, b) \in R \land \langle d, \text{DNAME}, IN, \tau', a', b' \rangle \in R \implies t = \tau' \land a = a' \land b = b' \)

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

8. The set of resource records \( R \) for a domain name \( d \), then there cannot be any other record type for that domain name.
   \( (d, \text{DNAME}, IN, \tau, a, b) \in R \land \langle d, t, IN, \tau', a', b' \rangle \in R \implies t = \text{CNAME} \)

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.
   \( (d, \text{DNAME}, IN, \tau, a, b) \in R \land \langle d', t, IN, \tau', a', b' \rangle \in R \land d \neq d' \implies d \neq d' \)

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.
    \( (d, \text{NS}, IN, \tau, a, b) \in R \land \neg \exists \langle d, a, b \rangle \in R \land \langle d', t, IN, \tau', a', b' \rangle \in R \land d < d' \implies t \neq \text{NS} \)

11. Wildcard domain names can not have a DNAME or an NS record.
    \( (d, t, IN, \tau, a, b) \in R \land d \notin \{d, a, b\} \implies t \neq \text{DNAME} \land t \neq \text{NS} \)

12. The set of resource records \( R \) for a domain name \( d \), 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.)
    \( (d, t, IN, \tau, a, b) \in R \land \langle d', t, IN, \tau, a, b \rangle \in R \land d \neq d' \implies \exists \langle d'', t, IN, \tau, a, b \rangle \in R \land d'' < d \land |d''| = |d| - 1 \)

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

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

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

**Theorem 5.1** (EC generation sound). For a given configuration \( C = \langle S, \emptyset, \Theta, \Gamma, \Omega \rangle \), 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 \( s \):

\[
\begin{align*}
a_1 &= \text{ServerLookup}(\Gamma(s), q_1) \\
a_2 &= \text{ServerLookup}(\Gamma(s), q_2)
\end{align*}
\]

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

1. \( \mathcal{N}(\Gamma(s), q_1) = \mathcal{N}(\Gamma(s), q_2) \)
2. \( a_1 \equiv a_2 \)
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 $\rho$ through the label graph, where $q_1, q_2 \in EC(\rho)$. We note that $q_1$ and $q_2$ can only differ if the final label in the path is $\alpha$.

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

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

The sets $\{z \in \Gamma(s) \mid dn(z) \leq dn(q_1)\}$ and $\{z \in \Gamma(s) \mid dn(z) \leq dn(q_2)\}$ 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 $\rho$, we now prove that these two sets are equivalent:

**Case 1 ($\rho$ does not end with $\alpha$).** In this case, $q_1$ and $q_2$ are the same query, and the equality is trivial.

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

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

$$a_1 = \text{ZoneLookup}(z, q_1)$$
$$a_2 = \text{ZoneLookup}(z, q_2)$$

Expanding the definition of ZoneLookup, we get:

$$a_1 = \text{RRLookup}(\{r \in \max_{<q_1,q_2} z \}, q_1, z)$$
$$a_2 = \text{RRLookup}(\{r \in \max_{<q_1,q_2} z \}, q_2, z)$$

The inner set $\{r \in \max_{<q_1,q_2} z \}$ 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 $\alpha$, then $\alpha$ 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 $dn(r) \leq dn(q_1) \iff dn(r) \leq dn(q_2)$ since $dn(r)$ cannot equal $dn(q_1)$ or $dn(q_2)$ (or else they would be in different ECs). Hence $dn(r)$ can only be a prefix of both $dn(q_1)$ and $dn(q_2)$. Similarly, if $dn(q_1) \in_\alpha dn(r)$, then $|dn(r)| \leq |dn(q_1)|$. Again assume $q_1 = \ldots \cdot l_{k+1} \cdot l_k \cdot l_{k-1} \cdot \ldots \cdot e$ and $dn(r) = l_j \cdot \ldots \cdot e$. If $j \leq k$, then $dn(q_1) \in_\alpha dn(r) \Rightarrow dn(q_2) \in_\alpha dn(r)$. If $j > k+1$, then $dn(r)$ would be in the label graph and $\alpha$ would exclude $l_{k+1}$ ($\alpha[0] = l_{k+1}$). If $j = k+1$, then it must be that $l_j = \ast$, in which case both $q_1$ match the wildcard for $dn(r)$.

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 $dn(q_1)$ and $dn(q_2)$ share the same prefix (except their last label) and $dn(r)$ 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 ($dn(r) = dn(q_1)$), 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:

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

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

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

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

$$a_1 = \text{WildcardMatch}(R, q_1, \{ty(r) \mid r \in R\})$$
$$a_2 = \text{WildcardMatch}(R, q_2, \{ty(r) \mid r \in R\})$$
There are now three cases for how \textsc{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:

\[
\begin{align*}
 a_1 &= (\text{Ans},\ \text{Syn}(\mathcal{T}(R,\text{ty}(q_1)),\text{dn}(q_1))) \\
 a_2 &= (\text{Ans},\ \text{Syn}(\mathcal{T}(R,\text{ty}(q_2)),\text{dn}(q_2)))
\end{align*}
\]

Expanding the definition of SYN:

\[
\begin{align*}
 a_1 &= (\text{Ans},\ \mathcal{T}(R,\text{ty}(q_1)) \cup \{(d,t,\text{IN},r,a,1) | \exists d', \langle d', t, \text{IN}, r, a, 0 \rangle \in \mathcal{T}(R,\text{ty}(q_1))\}) \\
 a_2 &= (\text{Ans},\ \mathcal{T}(R,\text{ty}(q_2)) \cup \{(d,t,\text{IN},r,a,1) | \exists d', \langle d', t, \text{IN}, r, a, 0 \rangle \in \mathcal{T}(R,\text{ty}(q_2))\})
\end{align*}
\]

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

\[
\begin{align*}
\text{real}(a_1) &= (\text{Ans},\ \text{real}(\mathcal{T}(R,\text{ty}(q_1)) \cup \{(q_1,t,\text{IN},r,a,1) | \exists d', \langle d', t, \text{IN}, r, a, 0 \rangle \in \mathcal{T}(R,\text{ty}(q_1))\})) \\
\text{real}(a_2) &= (\text{Ans},\ \text{real}(\mathcal{T}(R,\text{ty}(q_2))))
\end{align*}
\]

In the second case, we have \(\text{ty}(q_1) \not\in T\). Again we assume the types are equal, so we have \(\text{ty}(q_1) \not\in T\) and \(\text{ty}(q_1) = \text{ty}(q_2)\) and it follows that \(\text{ty}(q_2) \not\in T\). Therefore, \(q_2\) will evaluate to the same case, giving us:

\[
\begin{align*}
 a_1 &= (\text{AnsQ},\ \text{Syn}(R,\text{dn}(q_1)),\text{ans}(q_1),\text{ty}(q_1))) \\
 a_2 &= (\text{AnsQ},\ \text{Syn}(R,\text{dn}(q_2)),\text{ans}(q_2),\text{ty}(q_2))
\end{align*}
\]

As before, we compute real:

\[
\begin{align*}
\text{real}(a_1) &= (\text{AnsQ},\ \text{real}(\text{Syn}(R,\text{dn}(q_1)))) \\
\text{real}(a_2) &= (\text{AnsQ},\ \text{real}(\text{Syn}(R,\text{dn}(q_2))))
\end{align*}
\]

And then

\[
\begin{align*}
\text{real}(a_1) &= (\text{AnsQ},\ \text{real}(R)) \\
\text{real}(a_2) &= (\text{AnsQ},\ \text{real}(R))
\end{align*}
\]

Which gives the desired result.

In the final case, for \textsc{WildcardMatch} we trivially have \(a_1 = (\text{Ans},\ 0) = a_2\).

Case (\(\text{dn}(R) < \text{dn}(q_1),\ \text{DNAME} \in T\)). In this case there is a single DNAME record \(\langle R = \{r\} \rangle\). 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:

\[
\begin{align*}
 a_1 &= \text{Rewrite}(\{r\}, q_1) \\
 a_2 &= \text{Rewrite}(\{r\}, q_2)
\end{align*}
\]

Expanding the definition of \text{Rewrite}:

\[
\begin{align*}
 a_1 &= (\text{AnsQ},\ \text{DProc}(\mathcal{T}(\{r\},\text{DNAME}),\text{q_1})) \\
 a_2 &= (\text{AnsQ},\ \text{DProc}(\mathcal{T}(\{r\},\text{DNAME}),\text{q_2}))
\end{align*}
\]

Unfolding the definition of \text{DProc}, we get:

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

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

\[
\begin{align*}
\text{real}(a_1) &= (\text{AnsQ},\ \text{real}(\{r\})) = \text{real}(a_2)
\end{align*}
\]

Case (\(\text{dn}(R) < \text{dn}(q_1),\ \text{DNAME} \not\in T,\ \text{NS} \in T,\ \text{SOA} \not\in T\)). 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:

\[
 a_1 = \text{Delegation}(R, z) = a_2
\]

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 \(a_1 = (\text{Ans},\ 0) = a_2\).

Condition \(q'_1 \in \text{query}(a_1), q'_2 \in \text{query}(a_2) \Rightarrow q'_1 \sim_C q'_2\): 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) = 0.
Case (\( \text{dn}(R) = \text{dn}(q_1) \), \text{AUTHORITATIVE}(T), \text{ty}(q_1) \notin T, \text{CNAME} \in T, R = \{ r \})$. This is the \text{EXACTMATCH} case for a CNAME record. As before, we observe that \( q_1 = q_2 \), so the property is trivially satisfied.

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

Case (\( \text{dn}(R) < \text{dn}(q_1), \text{DNAME} \in T \)). This is the \text{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 \text{DProc}, we have:

\[
\begin{align*}
\text{query}(a_1) &= \{ (\text{dn}(r) \mapsto \text{ans}(r)), \text{ty}(q_1)) \} \\
\text{query}(a_2) &= \{ (\text{dn}(q_2) \mapsto \text{ans}(r)), \text{ty}(q_2)) \}
\end{align*}
\]

For this DNAME case, we know that \( \text{dn}(q_1) \) (represented by path \( \rho \)) are prefixes of \( \text{dn}(r) \). Suppose that \( q_1 = \ldots \circ l_k \circ l_{k-1} \circ \ldots \circ \rho' \) 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 \( \rho' \) 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 \( \rho' \). We will show that \( q'_1 \circ C q'_2 \) now belong to the same label graph path. Since \( q_1 \) and \( q_2 \) could only have been in the same EC if \( \rho \) ended in \( a \) in the label graph, and since by construction this \( a \) excluded all possible subdomains for extensions of \( \rho' \) after the rewrite, we know that the path matching \( q'_1 \) and \( q'_2 \) must end in \( a \). Since they match the same path, we conclude that \( q'_1 \sim C q'_2 \).

\[\square\]

\textbf{Theorem 5.2} (Soundness). For all \( C, q_1, q_2, \) and \( k \), if \( q_1 \sim C q_2 \), then \( \text{Resolve}(q_1, C, k) = \text{Resolve}(q_2, C, k) \).

\textbf{Proof.} From \text{Resolve}, we must show:

\[
\bigcup_{s \in \Theta} \text{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega \rangle, k) = \bigcup_{s \in \Theta} \text{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega \rangle, k)
\]

In particular, we prove a stronger inductive invariant:

\[\forall C, q_1, q_2, s, i. \; q_1 \sim C q_2 \implies \text{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega \rangle, i) = \text{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega \rangle, i)\]

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

\textbf{Base case} \((i = 0)\) trivial since we have

\[
\begin{align*}
\text{real}(\text{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega \rangle, 0)) &= \text{real}(\langle \text{ServFail}, \Theta \rangle) = \{ \text{ServFail} \} \\
\text{real}(\text{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega \rangle, 0)) &= \text{real}(\langle \text{ServFail}, \Theta \rangle) = \{ \text{ServFail} \}
\end{align*}
\]

\textbf{Inductive case} \((i)\) We must prove that

\[
\text{Resolve}(s, q_1, \langle S, \Theta, \Gamma, \Omega \rangle, i) = \text{Resolve}(s, q_2, \langle S, \Theta, \Gamma, \Omega \rangle, i)
\]

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

\textbf{Case 1} (\text{ServerLookup}(\Gamma(s), q_1) = (\text{AnsQ}(R, q'_1))\).

From the assumption of \( q_1 \sim C q_2 \) we know that

\[
\text{ServerLookup}(\Gamma(s), q_1) = \text{ServerLookup}(\Gamma(s), q_2)
\]

Substituting on the left, we get:

\[
\text{ServerLookup}(\Gamma(s), q_2) = (\text{AnsQ}(R, q'_1))
\]

Expanding the definition of \( \approx \), we get

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

Simplifying on the right:

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

This equality can only hold if \( \text{ServerLookup}(\Gamma(s), q_2) = (\text{AnsQ}(R, q'_2)) \) 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 \( \mathcal{N}(\Gamma(s), q'_1) = \mathcal{N}(\Gamma(s), q'_2) \).

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
\[
\text{Resolve}(s, q_1, (S, \Theta, \Gamma, \Omega), i) = \text{Resolve}(q'_1, (S, \Theta, \Gamma, \Omega), i-1)
\]
\[
\text{Resolve}(s, q_2, (S, \Theta, \Gamma, \Omega), i) = \text{Resolve}(q'_2, (S, \Theta, \Gamma, \Omega), i-1)
\]
therefore, we have:
\[
\text{real}(\text{Resolve}(s, q_1, (S, \Theta, \Gamma, \Omega), i)) = \text{real}(\text{Resolve}(q'_1, (S, \Theta, \Gamma, \Omega), i-1))
\]
\[
\text{real}(\text{Resolve}(s, q_2, (S, \Theta, \Gamma, \Omega), i)) = \text{real}(\text{Resolve}(q'_2, (S, \Theta, \Gamma, \Omega), i-1))
\]
From the inductive hypothesis, and the fact that \( q'_1 \sim_C q'_2 \), then we can conclude:
\[
\text{Resolve}(q'_1, (S, \Theta, \Gamma, \Omega), i-1) = \text{Resolve}(q'_1, (S, \Theta, \Gamma, \Omega), i-1)
\]
and we can finally prove the desired result:
\[
\text{Resolve}(s, q_1, (S, \Theta, \Gamma, \Omega), i) = \text{Resolve}(s, q_2, (S, \Theta, \Gamma, \Omega), i)
\]

**Case 2** (ServerLookup\((\Gamma(s), q_1) = (\text{Ref}, R)\))
From the assumption of \( q_1 \sim_C q_2 \) we know that
\[
\text{ServerLookup}(\Gamma(s), q_1) = \text{ServerLookup}(\Gamma(s), q_2)
\]
Substituting on the left, we get:
\[
\text{ServerLookup}(\Gamma(s), q_2) = (\text{Ref}, R)
\]
Expanding the definition of \( \approx \), we get
\[
\text{real}(\text{ServerLookup}(\Gamma(s), q_2)) = \text{real}(\text{(Ref, R)})
\]
Simplifying on the right:
\[
\text{real}(\text{ServerLookup}(\Gamma(s), q_2)) = (\text{Ref},\text{real}(R))
\]
This equality can only hold if: ServerLookup\((\Gamma(s), q_2) = (\text{Ref}, R')\) and also \( \text{real}(R') = \text{real}(R) \).

Since both cases will resolve using the Ref case, we can compute therefore, we have:
\[
\text{real}(\text{Resolve}(s, q_1, (S, \Theta, \Gamma, \Omega), i)) = \text{real}(\bigcup_{r \in T} (\text{real}(R), N_5) \text{Resolve}(\Omega(\text{ans}(r)), q_1, (S, \Theta, \Gamma, \Omega), i-1))
\]
\[
\text{real}(\text{Resolve}(s, q_2, (S, \Theta, \Gamma, \Omega), i)) = \bigcup_{r \in T} (\text{real}(R'), N_5) \text{Resolve}(\Omega(\text{ans}(r)), q_2, (S, \Theta, \Gamma, \Omega), i-1))
\]
From the definition of real, we can distribute over set union:
\[
\text{real}(\text{Resolve}(s, q_1, (S, \Theta, \Gamma, \Omega), i)) = \bigcup_{r \in T} (\text{real}(R), N_5) \text{real}(\text{Resolve}(\Omega(\text{ans}(r)), q_1, (S, \Theta, \Gamma, \Omega), i-1))
\]
\[
\text{real}(\text{Resolve}(s, q_2, (S, \Theta, \Gamma, \Omega), i)) = \bigcup_{r \in T} (\text{real}(R'), N_5) \text{real}(\text{Resolve}(\Omega(\text{ans}(r)), q_2, (S, \Theta, \Gamma, \Omega), i-1))
\]
From the inductive hypothesis, and the fact that \( q_1 \sim_C q_2 \), then we can conclude that for each \( r \in \text{real}(R) = \text{real}(R') \):
\[
\text{Resolve}(\Omega(\text{ans}(r)), q_1, (S, \Theta, \Gamma, \Omega), i-1) = \text{Resolve}(\Omega(\text{ans}(r)), q_2, (S, \Theta, \Gamma, \Omega), i-1)
\]
Since the components are pointwise equal, the set unions are also equal, so we obtain the desired result:
\[
\text{Resolve}(s, q_1, (S, \Theta, \Gamma, \Omega), i) = \text{Resolve}(s, q_2, (S, \Theta, \Gamma, \Omega), i)
\]

**Case 3** (otherwise)
The final case is immediate from the assumption of \( q_1 \sim_C q_2 \). In particular, this means:
\[
\{\text{ServerLookup}(\Gamma(s), q_1)\} = \{\text{ServerLookup}(\Gamma(s), q_2)\}
\]
and since real is applied pointwise over sets:
\[
\text{ServerLookup}(\Gamma(s), q_1) = \text{ServerLookup}(\Gamma(s), q_2)
\]
and by the definition of Resolve:
\[
\text{Resolve}(s, q_1, (S, \Theta, \Gamma, \Omega), i) = \text{Resolve}(s, q_2, (S, \Theta, \Gamma, \Omega), i)
\]
\(\square\)

**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 \( \alpha \) 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. \(\square\)
**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 \( 127 \times n \), 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. \( \square \)