UNIVERSITY OF CALIFORNIA

Los Angeles

# Formal Methods for a Robust Domain Name System

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Siva Kesava Reddy Kakarla

2022

ABSTRACT OF THE DISSERTATION

Formal Methods for a Robust Domain Name System

by

Siva Kesava Reddy Kakarla
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Todd D. Millstein, Co-Chair
Professor George Varghese, Co-Chair

The Domain Name System (DNS), one of the foundations of the modern-day Internet, primarily translates domain names into IP addresses, enabling easy access to online services. DNS name resolution appears simple at a high level but has evolved into a complex and intricate protocol over time. Errors in either DNS configurations or DNS implementations have far-reaching disruptive consequences. This is evident from past DNS issues that have rendered popular services such as GitHub, Twitter, HBO, LinkedIn, Yelp, and Azure inaccessible for extended periods. In this dissertation, I describe my work toward making the DNS as robust as possible by combining formal methods with DNS-specific insights to provide strong guarantees.

This dissertation presents the first formalization of DNS semantics, which I developed from multiple RFCs; this serves as the foundation for the rest of my work. Second, I detail a new technique, SCALE, for finding RFC compliance errors in DNS nameserver implementations via automatic test generation. SCALE symbolically executes the developed DNS formal model to jointly generate both the test queries and configurations (zone files).

Using SCALE, I identified 30 new bugs in 8 popular open-source DNS implementations such as BIND, POWERDNS, KNOT, and NSD, including 3 previously unknown critical security vulnerabilities. Third, I describe GROOT, the first verification tool for DNS configurations. Given the DNS zone files of an organization and a property of interest, GROOT automatically verifies that the property holds for all possible DNS queries or provides all counterexamples. GROOT performs exhaustive and proactive static analysis of DNS configuration files using an efficient algorithm for finding the equivalence classes of all possible queries to guarantee key correctness properties. I conclude with a theoretical analysis of the DNS to categorize its complexity and expressive power.

The dissertation of Siva Kesava Reddy Kakarla is approved.

Ryan Beckett

Ravi Arun Netravali

Yuval Tamir

Todd D. Millstein, Committee Co-Chair

George Varghese, Committee Co-Chair

University of California, Los Angeles

2022

*To Amma, Nanna and Chaitu*

TABLE OF CONTENTS

# LIST OF FIGURES

LIST OF TABLES

# ACKNOWLEDGMENTS

I am thankful to many supportive and encouraging people who made my grad school journey a smooth and happy ride. I was advised to pick my grad school mainly based on the advisor above anything else. I did, and as it turned out, I got lucky and had not one but two amazing advisors who had nothing but my well-being and success on their minds. Looking back, I am genuinely content with my decision to join UCLA and can not ask for anything more and will cherish the grad school memories forever.

I am beyond fortunate to have Todd and George, who supported me, taught me invaluable lessons and skills, and introduced me to multiple industry collaborations over the years. I could not have asked for a better combination for co-advising — big picture vision and meticulous with details. They always considered what was best for me, even if it meant taking a hit on their side. They believed in me and encouraged me to explore fundamental problems by giving me enough time and space. I had the unique opportunity to intern with three major cloud providers, and it would not have been possible without them actively reaching out to the researchers at various places.

Todd has always been a Slack message away to chat about anything. I had fun discussing and brainstorming ideas on the whiteboard with him and the impromptu discussions we had when he used to walk into the lab. I wholeheartedly thank Todd for spending countless hours reviewing my document drafts and slides and giving me constructive feedback. I have learned how to communicate ideas effectively in my presentations, mainly from him. I am also thankful to Todd for encouraging discussion of new ideas in the reading group, initiating inter-disciplinary research with other PL students in the lab, and organizing group get-togethers every year to cheer us up.

I was highly skeptical about choosing a very senior professor as my Ph.D. advisor in my undergrad. Oh boy! I have never been so wrong. George completely amazed me with his enthusiasm and excitement towards research. I have realized George is forever young at heart.

He uplifts the environment in the room with his positive energy. He is always thinking about students, what's best for us and how we feel from our perspective. I had numerous phone calls with George, sometimes even at late hours, discussing almost everything — research, crazy ideas, getting impact for the work, publicizing the work, career advice, how to dream big and even personal aspects. I will very much miss my mobile ringing to George's calls and Thanksgiving lunch at his home. Thank you, George, for everything, and I am deeply indebted to you for it.

I like to thank Ryan Beckett, Yuval Tamir, and Ravi Netravali for agreeing to serve on my committee.

Ryan has been nothing short of another advisor to me. This dissertation wouldn't have happened if Ryan did not reach out with an internship at MSR and my advisors inclining towards it. Whenever I was stuck and stressed, storming into Ryan's office and talking with him somehow magically calmed me, and we would come up with a solution in a short time. Thanks, Ryan, for continuing to be part of my journey after the internship, and even with your busy schedule, always available to discuss and brainstorm ideas.

Behnaz has been a great mentor and support for me from the MSR internship. Thanks to her and Ryan, I had a good time in the internship. Thanks to both of their efforts, I was able to submit the GRoot paper in time for SIGCOMM. I have also learned an important skill of how to not make slides over that internship!

I enjoyed working with many other outstanding colleagues over the grad school: John Backes, Karthick Jayaraman, Gavin McCullagh, Jayaram Mudigonda, Kyle Schomp, Anees Shaikh, Yuval Tamir, Alan Tang, and Ennan Zhai. I sincerely thank them for all the insightful discussion and helping me drive impactful research. I am also thankful to all the DNS operators, developers and the DNS-OARC community for their valuable feedback on my results.

Working with Yuval for my first project, SelfStarter, I learned the power of perseverance. I spent many Friday afternoons in his office going over spreadsheets and learning how to digest the data to ask important research questions. I appreciate and thank him for all the time he provided me. I thank Ravi for agreeing to a last-minute practice talk during the NSDI conference and for providing me with helpful feedback for improving my talk. I thank Karthick for his efforts in getting approvals for the Microsoft dataset for my SelfStarter paper. Kyle has worked very patiently with me over GRoot and helped us by suggesting a key property in GRoot. Jayaram and Anees were quite accommodating to applying my tools at Google during the Google internship and were flexible in what I could work on.

Thanks to John and Gavin, I had an excellent opportunity to integrate my tool, Ferret, into the Amazon continuous development pipeline. That internship really helped me demonstrate the tool's usefulness and how academic rigor can address practical problems. John was a fantastic manager who went beyond during and after the internship.

I highly appreciate my undergraduate advisor, Sandip Chakraborty at IIT Kharagpur, for being supportive and helping me with my grad school applications by writing letters at the last minute and finally in my grad school decision process.

The journey would not have been so exciting and memorable without the company of my friends at UCLA: Aishwarya, Saswat, Akshay, Pradeep, Murali, Arjun, Navjot, Vidushi, Ashutosh, Aditya, Sashank, Vishrant, Zhiyi, Parthe, Pratik, Gulzar, Alan, Aayush, Tianyi, Taqi and Poorva. I am thankful for the deep conversations in the lab and fun-filled ones outside. Aditya and Sashank made my first year comfortable, and I enjoyed going to many movies with them in that year. Thanks to Aish and Sawat, who helped me grow in several personal aspects. I have countless fond memories of living with them in the Keystone apartment. They pulled my leg every chance they got but were also equally supportive, encouraging, and patient with me.

Aish and I shared our thoughts about research and lab, which helped me gain a new perspective. I miss sitting beside her in the lab and having random conversations. Saswat

encouraged critical thinking in everything and introduced me to various exciting topics beyond my research. Thank you, Saswat, for the website and mulitple latex templates. I miss knocking on his room door and asking him to go over my proofs and implementation or explain to me how to do something. The board game nights with them and with Akshay, Arjun, Navjot, and Pratik are something I always looked forward to when we also had many philosophical discussions. Thanks, Akshay, for being ready all the time to organize birthday celebrations or to go out for food. With Pradeep and Arjun, I liked going out to the movies and cooking with them. I will miss their cooking. Thanks, Murali, for various short trips in and around LA.

This thesis is dedicated to my parents (Krishna and Sridevi) and my sister (Chaitu). I am grateful to them for their unending love, support, and faith in me. I am here because of the countless sacrifices my parents have made to orient everything for my sake. To my mother, who showed me that if you have strong determination, even a disability will not stop you from reaching your goal. She has been a constant source of encouragement, pushing me to explore further, ultimately leading me to opt for a Ph.D. To my father, for instilling the importance of discipline and conscientiousness. He is the all-rounder of the house, taking care of everyone and everything. Thanks to my sister and brother-in-law (Nagarjuna) for taking me on many trips and pampering me in the last five years.

# VITA

| | |
|---|---|
| 2011 – 2013 | Kishore Vaigyanik Protsahan Yojana (KVPY) Fellowship from Department of Science and Technology, India. |
| 2013 – 2017 | Jagadis Bose National Science Talent Search (JBNSTS) Scholarship. |
| 2015 | Intern, Indian Institute of Science (IISc), Bengaluru, India. |
| 2016 | Intern, LinkedIn, Bengaluru, India. |
| 2017 | B. Tech. in Computer Science and Engineering with Honors, IIT Kharagpur. |
| 2017 | UCLA Graduate Dean's Scholar Award (GDSA). |
| 2018 – 2019 | UCLA Dean's Graduate Student Research (GSR) Fellowship. |
| 2019 | Intern, MNR Group, Microsoft Research, Redmond, Washington. |
| 2019 | Teaching Assistant ("Computer Network Fundamentals" course), Computer Science Department, UCLA. |
| 2019 – 2020 | Applications Developer (part-time remote), Microsoft Research. |
| 2020 | Intern (remote), Google, California. |
| 2020 | Best Student Paper Award, SIGCOMM 2020. |
| 2021 | Facebook PhD Fellowship Finalist. |
| 2021 | Artifact Evaluation Committee Member, SIGCOMM 2021. |
| 2021 | Intern (remote), Amazon Web Services (AWS), California. |
| 2021 – 2022 | UCLA Dissertation-Year Fellowship. |

PUBLICATIONS

**Siva Kesava Reddy Kakarla**, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir and George Varghese. "(SelfStarter) Finding Network Misconfigurations by Automatic Template Inference." In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 99-1013. 2020.

**Siva Kesava Reddy Kakarla**, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. "GRoot: Proactive Verification of DNS Configurations." In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pp. 310-328. 2020.

Alan Tang, **Siva Kesava Reddy Kakarla**, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir and George Varghese. "Campion: Debugging Router Configuration Differences." In Proceedings of the 2021 ACM SIGCOMM 2021 Conference, pp. 748-761. 2021.

**Siva Kesava Reddy Kakarla**, Ryan Beckett, Todd Millstein, and George Varghese. "How Complex is DNS?" In Proceedings of the 20th ACM Workshop on Hot Topics in Networks (HotNets), pp. 116-122. 2021.

**Siva Kesava Reddy Kakarla**, Ryan Beckett, Todd Millstein, and George Varghese. "SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers." In Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 307-323. 2022.

# CHAPTER 1

# Introduction

The Internet has become an integral part of our live as most users access their critical data and services online. The Internet has proven to be a centerpiece during the COVID pandemic, with remote doctor visits and virtual meetings, sometimes even described as life-changing [1920, KP20]. The Internet is a network of networks communicating with each other through standardized rules, called *protocols*. A protocol is a standardized way of formatting and processing data so that two or more devices implementing the same protocol can communicate with and understand each other. The Internet operates by a collection of protocols, collectively termed the Internet protocol suite, defined in *Request for Comments (RFCs).* The RFC document of a protocol is a natural language (English) document that contains technical specifications and organizational notes for that protocol that people use to develop working implementations.

A key Internet protocol that enables the modern-day Internet is the Domain Name System (DNS). The DNS provides the essential service of translating website (domain) names like `ucla.edu` to their IP addresses, `127.97.27.37.` Each device or a service end-point on the Internet is given a 32-bit or 128-bit IP address; information packets from one device to another are routed using these IP addresses. Unfortunately, it is hard for humans to remember the device's or service's IP address; this is similar to how hard it is to remember phone numbers. The *phone directory* idea was used in the ARPANET era (1970's) to map human-friendly and memorable names of devices to their machine-friendly numerical addresses by maintaining a centralized text file called `HOSTS.TXT` at Stanford Research Institute. The human-friendly

names are called domain names. This centralized version worked for a decade, but by the 1980s it became unwieldy; the file was getting bigger with frequent updates taking too much bandwidth, prompting a strong need for an alternative decentralized model. Paul Mockapetris took on this arduous task and created the Domain Name System in 1983.

The DNS relies on a hierarchical database distributed across multiple *nameservers*, which are maintained by several different organizations to scale well for modern-day Internet traffic. A nameserver includes a collection of *zone files*, each of which contains DNS *resource records*, which in their simplest form are a map from a domain name to a corresponding IP address. The client interacts with these nameservers to traverse the hierarchical database to resolve a domain name to an IP address. The process or the software that performs this traversal on the client side is called a *resolver*. The DNS is on the critical path of every application we use today, as every client application first needs to resolve the name to an IP address to communicate with application servers. Unfortunately, the DNS is a fragile protocol; in particular, it is easy to make two classes of correctness errors in DNS that cause it to often fail.

The first class of errors are DNS protocol *implementation errors*. There are many popular nameserver implementations of the DNS protocol in the wild, both via open-source [Con86, Lab02a, CZ11, HC02] and in public or private clouds [Ama10, SBK20, Goo22, Mic22]. The DNS protocol, which started with two RFCs [Moc87a, Moc87b] in 1987, has evolved into a complex protocol, currently spread across more than 30 RFCs [sta22]. It is difficult to write an efficient, high-throughput, multi-threaded implementation that is also bug-free and compliant with these RFC specifications. Multiple implementations must interoperate with each other as different organizations can use different software. The implementations can suffer from incorrect or implementation-specific behavior that can cause crashes, outages, security vulnerabilities, and more.

The second class of errors are *errors in DNS configurations*. Operators within organizations manage the DNS by specifying how DNS nameservers 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 nameserver using server-level configuration files called *zone files*. While some automation exists — for example in primary-secondary replication of servers — many records in zone files 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 [Aka20]. Authoring and maintaining correct DNS configurations is quite challenging. Zone files tend to have thousands of records with multiple record types and complex dependencies distributed across nameservers, making the DNS intricate and subtle. Reasoning about how a single query will get resolved in the DNS (which is inherently nondeterministic due to multiple nameservers serving the same zone) is in itself a daunting task; worse, operators must ensure that *all possible* queries behave as intended.

Consequently, errors in either zone files or software that lead to performance or connectivity issues are widespread in practice [Zel13, Yor15, Inf19, New10, Tun19, Spe21, Sta21, Cla21]. To make matters worse, errors in DNS are often highly disruptive due to its global presence and residual caching effects from resolvers. For example, a recent DNS zone file misconfiguration at Microsoft resulted in a global outage impacting all Azure customers for 2 hours [Tun19]. Similarly, bugs in nameserver and resolver software lead to security vulnerabilities [Kov18, Ras16] that an attacker can exploit remotely to cause severe damage [Bin22, Pow22, NSD22, Fri08]. In summary, DNS is a single point of failure for Internet systems, making it the Achilles' heel of the Internet [Pom17].

To prevent these two classes of errors, we rely today on a mix of techniques such as monitoring, manual testing, fuzz testing, and manual review. However, these approaches can neither find functional correctness issues (software not adhering to the specifications (RFCs) or configurations violating user-defined properties, nor provide any strong guarantees — the system may still have bugs even after checking with conventional methods. This is because it is impossible to test for every possible input, given the huge state space, even more than the number of atoms in the universe. Further, monitoring-based approaches can only catch

errors after they have already been introduced into a live system: they cannot find *latent, hidden bugs* that can only be triggered in specific scenarios.

This thesis addresses the two classes of correctness errors in DNS — protocol implementation errors and organization-specific configuration (zone file) errors — with the help of formal methods to provide *provable correctness guarantees* to make DNS robust. Formal methods [But01, Col98], techniques based on mathematical logic, have helped researchers answer the seemingly impossible question: "can any input to a program result in that program producing an incorrect output?" They provide guarantees such as correctness with respect to a specification by formally proving the desired properties. Traditionally, formal methods have been extensively applied to verify hardware [KG99] and software systems [God97, Sch01].

Researchers have started exploring the use of formal methods for networking in the past decade with the rise of cloud networking. There is a large body of work on verifying the configurations at the network routing layer, and researchers have proposed numerous techniques to perform such verification generally, efficiently, and incrementally [KVM12, KZC12, MKA11, LBG15, FSF16, GVA16, BGM17, FFP15, BBC19, JBP19, ZLY20, BG22]. *Unfortunately, there are no such existing techniques for DNS.* 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. More generally, while there are 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*. We therefore need new techniques that address these challenges. These new techniques can also be generalized in the future to apply to other protocols of the network ecosystem that have similar challenges. Moreover, much of the work in the network verification area has focused on router configurations, leaving to individual router vendors the task of checking whether network devices implement their protocols correctly. My thesis on using formal methods for DNS is a contribution on multiple fronts that can serve as the basis of future work in the broader network verification area, as I explain below.

To develop techniques using formal methods, we need to precisely reason about the behavior of the DNS. Although existing RFC standards specify the behavior of the DNS, these standards are informal and described in English. Therefore, as part of this thesis, we first present a mathematical formalization of the DNS that allows for the development of *automated* techniques with strong guarantees. The formal model of DNS that we developed from multiple RFCs is the *first formal model* of DNS and serves as the foundation for the rest of my thesis.

Using the formal model, I developed the first automatic test generator for finding RFC compliance errors in DNS nameserver implementations to handle protocol implementation errors. The test generation approach handles the large space of possible scenarios and the need to generate both user queries and zone files. It does so by creating an executable model of the above formal DNS resolution semantics and then symbolically executing it for *all* paths through the model up to a bound, but that still generates tests that cover many distinct nameserver behaviors. The next part of this thesis describes how to perform proactive verification of DNS configurations; this work is the first work on DNS configuration verification. Given the DNS zone files of an organization and a property of interest, GRoot can automatically verify that the property holds *for all possible DNS queries*, or provides all counterexamples.

Next, we observe that the DNS, which started as a simple distributed key-value store to get the IP address for a domain, quickly gained popularity and began to be used to retrieve other kinds of information to enable different use-cases [RW12a, DNS22a, DNS22b, Mea02]. That was made possible by adding new resource record types to the DNS protocol, each with its accompanying semantics [Aut22]. Unfortunately, adding new record types also increases the complexity of the protocol, leading to a decrease in the ease with which humans and machines can analyze the system. In the final part of my thesis, I analyze the theoretical complexity of the DNS protocol to categorize its power. I show that DNS has surprising complexity with the power to express regular languages and pushdown systems.

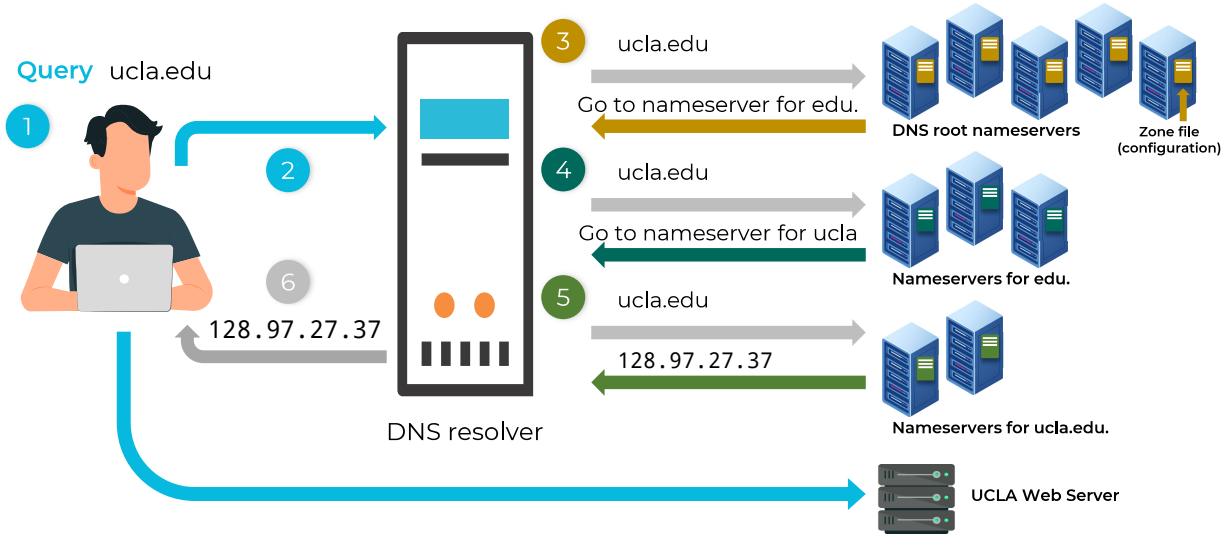



**Figure 1.1:** The resolution process for the domain name `ucla.edu` (with no caching).

In the remainder of this introduction, I start in Section 1.1 by giving an overview of how a domain name is resolved to an IP address in the DNS. In Section 1.2, I give an example and background information about DNS configurations – the zone files. Section 1.3 mentions some major outages due to DNS errors to show how costly and common the errors in the DNS are. Finally, Section 1.4 describes the thesis statement and overview the techniques that I have developed to make DNS robust.

## 1.1 Domain Name Resolution

The DNS, as I have described earlier, is a distributed database that operates in hierarchical fashion. Intuitively, each nameserver has a chunk of the entire database in the form of zone files. The root nameservers are the starting point for any domain name resolution. They can either return the information the user requested (if they have it), or point to downstream nameservers which can help resolve the name. The process happens recursively until the user reaches a nameserver that has the final response.

For instance, consider the example resolution in Figure 1.1. To access the `ucla.edu` website, the user's browser first needs to resolve it to an IP address (step 1). To get the IP address, the browser sends a DNS query to a local resolver, *e.g.* from the user's ISP (step 2). The resolver is responsible for initiating and going through the full DNS resolution process to resolve the domain name on behalf of the user to return the resource sought, which in this case is an IP address. The resolver starts by sending the query to one of the root nameservers, of which there can be many, to ensure redundancy and availability of the DNS information. Each resolver is bootstrapped with the IP addresses of the root nameservers to avoid circular dependencies, as the root nameservers are also identified with domain names.

The resolver chooses one of the root nameservers nondeterministically to send the query. The root nameserver does not know the IP address of `ucla.edu` as there is no record for it in its zone file, but it has downstream nameserver records for `edu`. These records mean that for any information about `edu` domain or any domains ending with `edu`, the resolver must contact the nameservers mentioned in those records (step 3). The process repeats (step 4). The resolver caches the responses returned by nameservers to reduce the resolution time for the future queries. Eventually, the query reaches the server with the IP address record, which it returns to the resolver (step 5), which returns it to the user's device (step 6) and the user can reach the UCLA web server.

But who decides which chunk of the database is allocated to particular nameservers and how these databases are updated?

The Internet Corporation for Assigned Names and Numbers (ICANN), a non-profit organization, helps coordinate the DNS. It maintains one of the root nameservers and entrusts the operation of the rest of the root nameservers to various organizations, including NASA, and Verisign. Each top-level domain (TLD), which is immediately below the root domain like '.com', '.org', '.uk', and '.edu', is maintained and serviced by an administrative organization called a registry. Each registry delegates the commercial sales of domain name registrations in their TLD domain space to registrars. For example, when a registrar sells a `campus.edu`

domain registration to an end-user or organization, the registrar must notify EduCase — the registry for '.edu' domain.

The organization which registered the `campus.edu` domain now has the autonomy to decide which nameservers serve the `campus.edu` domain and further controls how the `campus.edu` namespace is organized and what information will be returned. The EduCase registry updates all of the '.edu' nameservers' zone files with the information that to seek any information related to `campus.edu` and its subdomains, query the nameservers picked by the organization. A subdomain of `campus.edu` is any domain name that ends with `campus.edu`, such as `cs.campus.edu`, `www.ee.campus.edu`, *etc.* The organization can create subdomains of `campus.edu` and delegate control of newly created subdomains to other nameservers.

## 1.2 DNS Configurations (Zone files)

Each organization that registers a domain namespace maintains and organizes it with the help of configuration files, which are called zone files in the DNS terminology. The nameserver software uses the zone files to answer user information requests, called DNS queries. For example, information about `ucla.edu` domain name space is available to the users through the following four nameservers - `ns1.dns.ucla.edu`, `ns2.dns.ucla.edu`, `ns3.dns.ucla.edu`, and `ns4.dns.ucla.edu`, where each of them has the (likely same) `ucla.edu` zone file. Most nameservers will disallow a user query to transfer the entire zone file for security reasons; instead, they only respond to individual resource queries.

At the highest level, each zone file is a collection of resource records, whose format is fixed by the RFCs. A resource record has 5 fields - domain name, type, time to live, class and type-specific data. An example zone file for the `campus.edu` zone at the nameserver `ns1.com` is shown below.

Every zone file must have a resource record of type `SOA` (start-of-authority), which contains administrative information about the zone. In our example, we assume the administration

```
   campus.edu.  SOA  500  IN  ns1.com. admin.uni.edu. 11 600 30 400 500
 cs.campus.edu.  NS   500  IN  ns1.campus.edu.
ns1.campus.edu.  A    500  IN  1.2.3.4
```

**Figure 1.2:** Example zone file for `campus.edu` zone.

decides to cede the responsibility of the `cs` subdomain to the Computer Science department; to do so, it creates a delegation (`NS`) in the zone file. The CS department DNS admin will create a new `cs.campus.edu` zone file and manage its subspace. The CS admin has decided to use the `ns1.campus.edu` nameserver for their zone, which they must communicate and update in the `campus.edu` zone file. The `ns1.campus.edu` nameserver is under (or more formally, a subdomain of) `campus.edu`; therefore, the DNS administrator should also add its IP address (`A`) to the zone file. Without that bootstrap record, the user cannot resolve the `ns1.campus.edu` name to contact it to retrieve any information about the `cs` zone.

Note that in this example even a simple configuration requires coordination between two admininstrators; a slight mistake or error from the CS admininstrator can render the `cs` website and all the resources under it unreachable. When the nameserver is changed or assigned a new IP address, the admin must remember and update it in the parent `campus.edu` zone file. Things get more complicated when record types that rewrite the input query are employed, making it even harder for the administrator to reason about all possible behaviors.

The DNS nameserver software developers must carefully consider various record types and interactions to respond correctly to user queries. For example, for the user query, `www.cs.campus.edu`, the DNS nameserver implementation should return the `NS` record; the implementation can even go further and proactively return the `A` record to avoid extra round-trips from the user to the same nameserver. Any error in understanding the semantics of record types and accompanying nuances will result in the *implementation* returning incorrect responses, even though the zone file is correct.

## 1.3 The Cost of DNS Errors

Given the delicate nature of the DNS, even a minor error in the zone files or the nameserver implementation can render a vital service unavailable for extended periods causing a widespread outage. An unfortunate consequence of the fragility of the DNS is that such outages are all-too-common, with DNS-related outages making the headline news frequently. There is even a popular joke among network operators that DNS is always to blame when a network outage happens, even when it initially did not seem that way. The Haiku shown in Figure 1.3 is widely popular among network engineers and sysadmins. It shows how common DNS-related outages are and how unforgiving they can be. The following is a small subset of the real incidents caused by DNS errors.

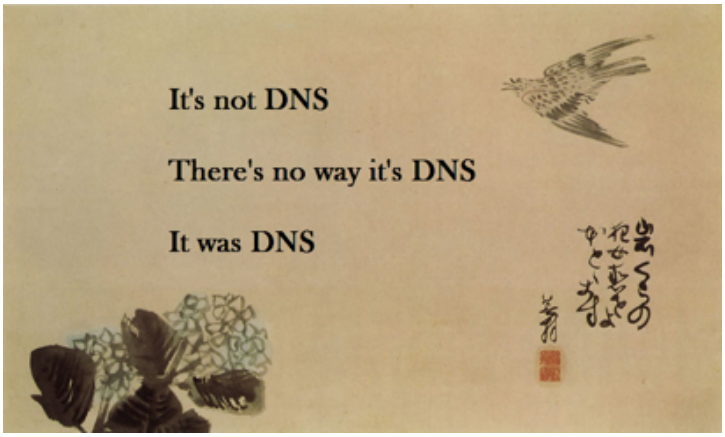



**Figure 1.3:** A Haiku (Japanese short poem) that SSBroski (a network engineer) came up with to describe his experience diagnosing network issues.

- In 2021, Slack had an outage that lasted for 24 hours and impacted around 1% of their online user base (>10 Million). It was triggered by a subtle bug in Amazon Route 53 DNS implementation. Slack uses the Amazon Route 53 DNS nameserver implementation to serve their zone files. Slack was impacted by the bug and rolled back. The rollback caused massive impact to Slack users [Raf21].

- In 2021, Salesforce made an erroneous DNS configuration change which impacted multiple data centers, affecting the Salesforce service for customers and Salesforce authenticator, and limited internal access for the Salesforce Technology team and

Subject Matter Experts (SMEs) investigating the disruption. In addition, the status.salesforce.com Trust site became unavailable during the incident [Spe21].

- In 2019, Microsoft experienced one of its worst outages due to DNS which lasted for many hours due to caching effects. The outage affected many services within Microsoft Azure such as Compute, Storage, App Service, Azure Active Directory, and Azure SQL Database services. Additional Microsoft services were affected as well, including Microsoft 365, Dynamics, Azure DevOps, and more [Tun19].

- In 2018, a misconfiguration for the JavaScript Node Package Manager (NPM) caused some users to lose access to the service world-wide for almost 48 hours [npm18].

- In 2014, a misconfiguration at GitHub resulted in a loss of access to open source repositories. As a result of this outage, GitHub customers experienced 42 minutes of downtime of services along with an additional 1 hour and 35 minutes of downtime within a subset of repositories as they worked to restore full service[Fry14].

- In 2008, Dan Kaminsky discovered a serious vulnerability in almost all the DNS implementations. This vulnerability could allow an attacker to launch cache poisoning attacks redirecting network clients from a legitimate web site to a fake one without the web site operator or end-user knowing, presumably for nefarious ends. This caused quite a furor in the security community, and led to a mad dash to patch DNS servers worldwide [Fri08].

While these are a few of the large DNS-related incidents that are publicly visible, less extreme errors are also quite common; there are inconsistencies in almost all the newer TLD zone files and over 13 million second-level domains [SJR20, BG17, SMJ20]. Such inconsistencies in the zone files affect query load distribution and resolution latency, and pose a risk to the availability of the zone. The recent TsuNAME vulnerability showed how misconfigured zone files with cyclic dependencies could be exploited to carry out distributed

denial-of-service (DDoS) attacks against critical DNS infrastructure like large TLDs or ccTLDs, potentially affecting country-specific services [MCH21]. Researchers have also demonstrated the possibility of various DDoS attacks against different DNS implementations that take advantage of the DNS semantics of referrals, rewrite record types, and wildcard records [BR18, ABS20]. An attacker can remotely exploit such DNS features if the implementations do not adequately safeguard against them.

## 1.4 Thesis Statement and Contributions

For a proper working Internet, it is crucial to ensure the *robustness* of DNS. This specifically involves checking that there are no inputs that can lead to bad or incorrect DNS behavior. My thesis is that *we can combine formal methods, which have proved successful in other areas, with DNS-specific insights to develop techniques that can provide strong correctness guarantees that help achieve a robust DNS.* Recall that DNS, as described above, has two classes of correctness errors: protocol implementation errors and organization-specific configuration errors. My goal is to develop techniques based on formal methods that address *both* kinds of errors and make DNS robust.

I have made significant progress on the proposed goals during my doctoral program. In this thesis, I make four contributions (Figure 1.4) to improve the DNS. The following four subsections briefly describe each of the four contributions, which are described in detail in the subsequent chapters. Section 1.4.1 describes the need and challenges involved in developing a formal model for the DNS. Next, in Section 1.4.2, I present a technique called SCALE and a tool based on this technique Ferret to tackle DNS protocol implementation errors in DNS nameserver implementations. I present a configuration verifier, GRoot, in Section 1.4.3 that can proactively catch any errors in organization-specific configurations. As a final contribution, I present a theoretical complexity analysis of the DNS protocol in Section 1.4.4 to help better understand the DNS and categorize its power.

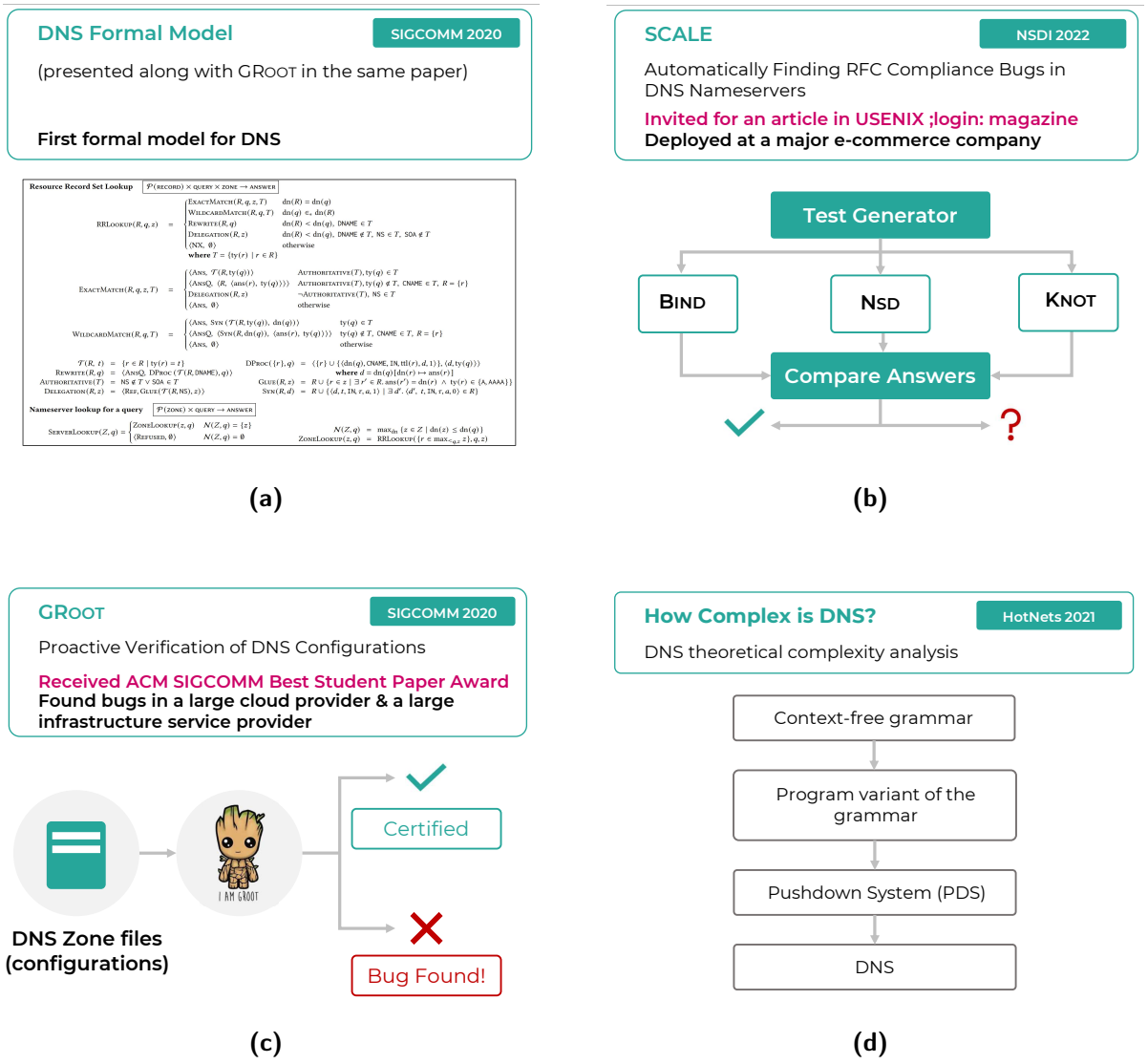



**Figure 1.4:** My four contributions through this thesis for improving the DNS.

### 1.4.1 Developing a DNS Formal Model

To this day, DNS implementers must read and interpret tens of hundreds of RFC specification pages. They find it hard to interpret these specifications mainly because they are in an easy-to-misinterpret colloquial English that can be ambiguous. When faced with ambiguity,

the choices they make can have broad ramifications. This is because implementation must interoperate with other implementations written at a different time. The fact that there is an errata for the majority of the published RFCs highlights the ambiguity and problems caused by RFCs written in informal English [For22].

To address these issues, multiple protocol specification languages have been proposed over the years to provide formal descriptions of the protocol [BD87, KKM99, MZ19b, MBJ20]. However, there are both technical and social reasons for RFCs still being specified in English over formal descriptions. Technical limitations include the incompleteness of the specification languages, whereas the social or non-technical roadblocks include a lack of familiarity and expertise with formal techniques by engineers.

Having an unambiguous formal description of the protocol not only helps protocol implementation developers but is a prerequisite to developing any tools that provide correctness guarantees. Therefore, to exhaustively verify the behavior of the DNS, this thesis starts by formalizing its behavior. Ideally, the formal model should be precise about the key behavioral aspects of the protocol but is also reasonably abstract; this allows developers the freedom to make decisions on the implementation's low-level details (for example, the kind of data structures used) and hence, the model should apply to many different implementations.

Using a formal model to provide strong guarantees dates back at least half a century when techniques like Floyd-Hoare logic [Flo67, Hoa69] and Dijkstra's predicate transformer semantics [Dij75] were used to reason about software programs rigorously. Formal techniques have been traditionally used to verify computer hardware like formal verification of chip correctness [Ben01, FM18] and software like program verification [God97, Sch01, BH14] and developing a verified kernel implementation [KEH09]. Formal models were also used, though in a limited fashion, in networked systems to verify a routing protocol [BOG02] and some aspects of TCP and UDP [BFN05].

Unfortunately, there is no existing formal model of DNS behavior that we can use to develop techniques with strong guarantees to check for correctness. Therefore, we took on the

time-consuming task of going through multiple DNS RFCs to build a formal, mathematical semantics for DNS, including both nameserver lookup and recursive resolution. We did not use any specific protocol specification language; instead, we developed it as a language-agnostic, declarative, mathematical (paper) formalism.

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. There were times when we developed an initial model after reading the RFCs, only to realize we had misinterpreted the RFC after testing a few examples with an actual implementation like BIND [Con86]. To our knowledge, this is the first formal model of DNS; we hope that in the future, researchers can build on this model to more precisely reason about DNS behavior.

### 1.4.2 Handling Protocol Implementation Errors

A DNS nameserver implementation is developed like any other large software. Developers implement the logic described in the specifications and validate the correctness of the implemented logic using a small set of manually written unit tests. Manually writing tests is onerous for all software developers: they have to write good tests that cover a wide range of program behaviors. The situation is worse for DNS implementers as a DNS test case consists of *both a query and a configuration (zone file)*. The DNS developer must come up with both of them, keeping in mind that the query must be related to the zone file for the test to reach the core query resolution logic.

This part of the thesis automatically generates high-coverage query and zone file inputs to stress test query resolution of DNS nameserver implementations to find behavioral errors. Doing so is challenging because zone files must satisfy several structural constraints to be well-formed, and because zone files have many different kinds of record types and features and subtle interactions among them. Existing standard automated test generation approaches

like fuzz testing [Pat22, Foo15, Cam19, St10], and symbolic execution [Kin76, SCP14, RE15] of the implementations are not suitable for our needs. As we will argue, they both have difficulty with generating complex *structured* inputs requirements like zone files. It is hardly surprising that existing techniques do not generate zone files.

I will illustrate why it is hard to generate good tests for DNS nameservers with the help of an important test our approach generated that exposed a previously unknown performance bug in BIND [Con86], one of the widely used DNS implementations. The zone file part of the test is shown in Figure 1.2 and the test query name was `www.cs.campus.edu`. Both the test zone file and the query were completely auto-generated by our approach. In this test case, the query matches the `NS` record in the zone file. NSD [Lab02a], KNOT [CZ11], and POWERDNS [HC02] correctly return the `NS` record along with the `A` record, avoiding extra round-trips to determine the nameserver's IP address, while BIND returns only the `NS` record, which the developers agreed to be an erroneous behavior.

Even though the zone file we generated has only a few records, it has complex dependencies that must be met to trigger this behavior in BIND. First, there must be a delegation of the query to another nameserver. Second, that nameserver must be in the same zone. Third, that nameserver must be a sibling domain. Fourth, there must be an `A` record for that nameserver in the zone. Given these dependencies, it is understandable that prior testing techniques did not uncover the bug and how quickly it becomes intractable for humans to write tests manually.

This thesis presents a new approach for automated testing of DNS nameserver implementations, call **SCALE** (Small-scope Constraint-driven Automated Logical Execution), which *jointly* generates zone files and the corresponding queries, does so in a way that is targeted toward covering many different RFC behaviors, and is applicable to black-box DNS nameserver implementations. The key insight underlying SCALE is to use the formal model of the logical behaviors of the DNS resolution process to guide test generation. Specifically, we have created an *executable* version of the formal semantics of DNS described in Chapter 3 of the

thesis, which we then symbolically execute to generate tests for black-box DNS nameservers — each test consisting of a well-formed zone file and a query that together cause execution to explore a particular RFC behavior.

We have built a tool called **FERRET** based on this approach and applied it to test 8 open-source DNS implementations, including popular implementations such as BIND [Con86], POWERDNS [HC02], KNOT [CZ11], and NSD [Lab02a]. We identified and reported 30 new unique bugs from these test cases, including at least one bug in every implementation, of which 21 have already been fixed. Many of these bugs existed in even the most popular DNS implementations, including a critical vulnerability in BIND that attackers could easily exploit to crash DNS resolvers and nameservers remotely. The FERRET tests were also used to test the DNS nameserver implementation of a major e-commerce company, and the tests are integrated as unit tests in their internal continuous integration and development (CI/CD) pipeline. Due to confidentiality reasons, I will not discuss these results in this thesis.

### 1.4.3 Handling Organization-specific Configuration Errors

Today, many operators use blackbox techniques for checking DNS configuration correctness (*e.g.*, live testing and monitoring). For example, operators can monitor for ongoing problems through offerings from commercial vendors, such as ThousandEyes [Kep20], CheckHost [Hos20] or research tools [PFM04]. 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. Further, these approaches are *reactive* in that they can detect errors after they have been introduced into a live system, in which case, the damage has already begun.

This thesis investigates an alternative approach, namely *proactive verification* of the DNS configurations - the zone files. To the best of our knowledge, GROOT is the first verification tool for DNS configurations. GROOT performs static analysis of DNS zone files, enabling an organization to use GROOT before deploying their zone files to find DNS misconfigurations

and is, therefore, proactive. Some example properties that GROOT can check on the zone files are - "Is there any query that will be rewritten in a loop for the input zone files?", or "Is there any query for which different DNS executions will result in different answers?"

For all such properties, GROOT provides all queries that violate the property and when no counter-example is found, it is guaranteed that the system is robust with respect to zone files; therefore, GROOT is exhaustive and so is a verifier and not a tester. While the number of possible DNS queries is huge, I observe that the number of distinct behaviors is much smaller and is a function of the zone files. Based on this insight, GROOT first performs an analysis of the zone files to partition all possible queries into *equivalence classes* (ECs) each of which captures a distinct behavior. GROOT then performs a *symbolic execution* of each EC to produce its set of answers and check the given property. Our formal model of the DNS resolution is crucial for the efficient symbolic execution in GROOT. I provide various proofs in Chapter 5 to show GROOT performs sound and complete verification, and also generally provides a massive reduction in complexity by generating a small number of equivalence classes.

GROOT was applied to zone files obtained from a large campus network, a large infrastructure service provider, and a large cloud provider. The campus zone files have over a hundred thousand records, and GROOT revealed 109 new bugs in under 10 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 (query is eventually rewritten and does not resolve to an IP address) in 3 minutes, which initiated a cleanup of the zone files. On the large cloud provider zone files, GROOT found many property violations, including four cyclic zone dependencies. Due to confidentiality reasons, I will not be describing the issues found in the cloud provider further in this thesis. Finally, on a synthetic dataset that I created from over 65 million real DNS records [DNS13] I found that GROOT can scale to networks with tens of millions of records spread across tens of thousands of zones.

### 1.4.4 Analysing DNS Theoretical Complexity

Internet pioneer Geoff Huston [Hus20] once made the following profound remark about DNS: "The DNS is simple in the same way that Chess or Go are simple. They have all constrained environments governed by a small set of rigid rules, but they all possess astonishing complexity." After going through multiple RFCs to develop the tools to handle both kinds of errors mentioned above and make DNS robust, I completely agree with his statement. DNS is significantly more complex than people realize. I was intrigued and motivated by the recent results that show that Internet protocols can be surprisingly complex; in particular, that BGP [RHL06] is Turing complete [CCD13]. I asked the same question for the Domain Name System (DNS). This seems like an important fundamental question because DNS is at least as pervasive and essential as BGP in the global Internet infrastructure.

Besides the scientific interest, the complexity of DNS can have implications for new applications (that can utilize the unsuspected power of DNS), for security (to understand how attackers can exploit DNS via new vectors and how to defend against it), and for verification (to understand basic complexity limits and suggest new verification algorithms). In this thesis, I show how DNS can recognize arbitrary regular languages using a single zone file at a single nameserver, which can be used to build a system for controlling domain access (of which parental control is a particular case).

Next, I demonstrate that the power of DNS extends beyond regular languages by showing how pushdown systems (PDS) can be encoded in DNS. A pushdown system is a transition system equipped with a finite set of control locations and a stack. The stack length is unbounded, and hence, a PDS may have infinitely many reachable states, making them more powerful than a regular language recognizer. I take advantage of the inherent nondeterminism in DNS due to nameserver delegation to encode a PDS in DNS and use this as a subroutine to generate strings of arbitrary context-free grammars. Consequently, the verification of DNS configurations is likely to require time cubic in the number of DNS records in the worst case.

I believe DNS has the same power as a PDS, and therefore, DNS is not Turing-complete. The proof for reducing DNS to PDS is a work in progress and will, unfortunately, not be part of this thesis. I do, however, sketch the proof intuition in Figure 7.1.

## 1.5 Additional Comments

The work in this thesis is a revised and extended presentation of research developed through collaborative work. The chapters herein are based on a series of co-authored papers [KBA20a, KBM22, KBM21]. Chapter 3 of the foundational DNS formal model was presented as part of the paper published in SIGCOMM 2020 [KBA20a], which won a best student paper award. Chapter 4 on finding RFC compliance bugs (protocol-implementation errors) in DNS nameserver implementations is based on a paper from NSDI 2022 [KBM22]. The configuration verification work presented in Chapter 5 was part of the SIGCOMM 2020 [KBA20a] paper that also introduced the first formal model for DNS. Finally, Chapter 6 on DNS complexity is based on work that appeared in HotNets 2021 [KBM21].

# CHAPTER 2

# DNS Background

In this chapter I give an overview of the DNS, summarizing the key ideas and concepts defined across various RFCs necessary to understand our work. This includes a description of the domain namespace (Section 2.1), the fields of DNS resource records, resource records types used throughout this thesis, their intended use (Section 2.2), and how they influence the query resolution process at a nameserver (Section 2.3). I also briefly describe wildcard records (Section 2.4) in this chapter. Readers familiar with these DNS concepts can skip ahead to the DNS formal model chapter (Chapter 3).

## 2.1 Domain Namespace

A domain name consists of one or more parts, technically called labels, that are conventionally concatenated and delimited by dots (`.`), such as `ucla.edu`. These labels form a tree-like hierarchy with the root as an empty label and `edu` as a child of it, and so on. Each label may contain zero to 63 characters, and the label with zero characters is reserved for the root node. The hierarchy of domains descends from the right to left - the label immediately left to another is a child node of the label on the right. The left label is called a subdomain of the domain on the right. A small part of the domain namespace is shown in Figure 2.1. For example, the domain name `ucla.edu` is represented by a thick brown line in the figure showing how the labels are arranged in a hierarchical order. The full domain name may not exceed the length of 253 characters, including the `.`, when represented in its textual form.

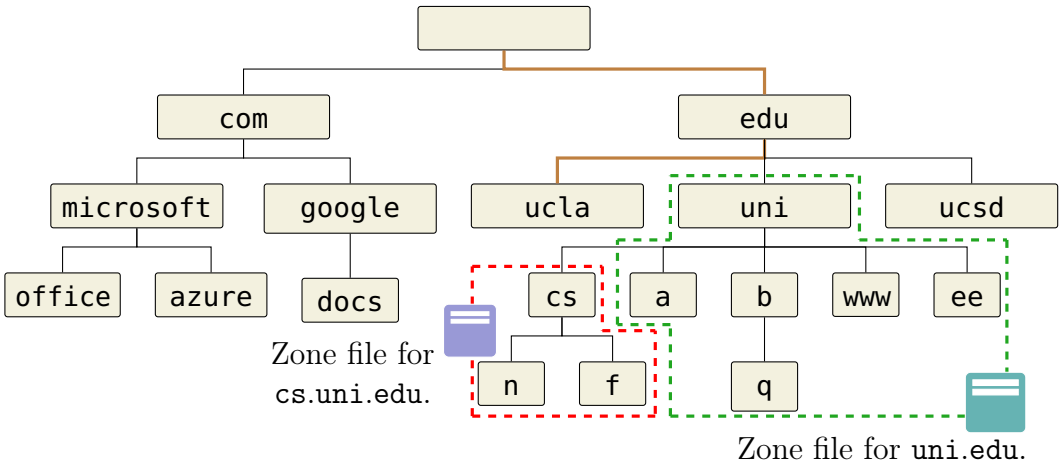



**Figure 2.1:** Example domain namespace with some commonly used domain names. The `uni` and all of its subdomains except `cs` can be managed by one team, where as a specialised team can handle `cs` and the subdomains under it.

Ideally, the domain name should end with the trailing `.`, but it is generally ignored when written in the text form for brevity.

## 2.2 Resource Records and Zones

Each label at any level in the hierarchy can contain information, and the user obtains that information by querying the domain name formed by joining the labels from that node to the root. An example sequence of steps of how a user retrieves the information with the help of a resolver is described in Section 1.1. Data is stored as DNS *resource records* (RRs) where each record has the following fields:

1. **Name:** The name of the node to which this record belongs to. It is generally represented as the full domain name starting with the node in the tree and going up to the root node while concatenating the nodes on the path.

| Example Record | | | Description |
|---|---|---|---|
| uni.edu. | SOA | ns1.com. admin 11 600 30 400 500 | Start of authority record |
| a.uni.edu. | A | 1.2.3.1 | IPv4 record |
| q.b.uni.edu. | AAAA | 1:db8::2:1 | IPv6 record |
| ∗.uni.edu. | TXT | "Awesome" | Wildcard Text record |
| cs.uni.edu. | NS | n.cs.uni.edu. | Delegation record |
| n.cs.uni.edu. | A | 5.4.2.7 | Glue IPv4 record |
| www.uni.edu. | CNAME | uni.edu. | Canonical name |
| ee.uni.edu. | DNAME | elec.com. | Domain redirection |

**Table 2.1:** uni.edu. zone file showing common DNS record types (the class and TTL fields are omitted).

2. **Type:** The type of the resource in this resource record. Types refer to abstract resources. It indicates the format of the data and it gives a hint of its intended use. For example, the A record is used to translate from a domain name to an IPv4 address.

3. **Class:** The class of a record, which defines an independent namespace. For all our purposes, we fix the class to IN (for Internet), which is the most commonly used class, though the techniques I present are applicable to others as well.

4. **Time To Live (TTL):** The TTL describes how long a RR can be cached by resolvers and others querying the records before it should be discarded and queried again.

5. **Rdata:** The type-specific data, such as the IP address for address records, text information for TXT records.

The namespace database tree is divided into a large number of *zones* using "cuts". Cuts (separation) in the namespace can be made between any two adjacent nodes. After all cuts are made to the namespace, each group of connected namespace is a separate zone. A zone is a collection of records that share a common end domain name. For example, the uni.edu zone has only records ending with uni.edu. There is a cut between cs and uni nodes, effectively causing the cs.uni.edu to be a new zone as shown in Figure 2.1. The zone cut is also evident

from the zone file as there are `NS` records for `cs.uni.edu` naming the nameservers for the `cs` subzone.

DNS supports many record types [Aut22], including records for IP addresses, text records, domain aliases, delegation records, and more. Table 2.1 shows a few example records in the zone file of `uni.edu` zone. Here, the domain name `uni.edu` is called zone domain or zone apex for this zone file. When a domain name is shown as part of a zone file without the trailing `.`, then it is considered as a *relative* domain, and the zone domain is appended to it to form the complete domain. We describe each record type behavior for record types frequently used through out this thesis.

- `SOA` (start of Authority) [Moc87b] - Each zone file will have exactly one record of this type which contains administrative information about the zone. The name field will be the name of the zone. The rdata field of the record has several sub-fields like the primary server responsible for the zone (`ns1.com`), the email address of the administrator responsible for the zone (`admin` is a relative domain name and should be treated as `admin.uni.edu`; therefore the email address is `admin@uni.edu`), followed by five integers that specify several timers relating to refreshing the zone.

- `A` (address) [Moc87b] - This is the most commonly used record type, which is used to map the domain name specified by the record's name field to a 32-bit IPv4 address.

- `AAAA` (quad A) [KHT03] - Similar to the `A` record type, `AAAA` is used to obtain an IPv6 address for a given domain name.

- `TXT` (text) [Moc87b] - This record type is used to provide any additional information about a domain name like site verification, framework policies, *etc.*

- `NS` (name server) [Moc87b] - This record type specifies which nameserver is authoritative, *i.e.*, has the information for the namespace specified in the name field of the record. To obtain any information for the computer science domain (`cs.uni.edu`) or any of its

subdomains, one needs to contact the server mentioned in the rdata field of the record (`n.cs.uni.edu`).

- `CNAME` (canonical name) [Moc87b] - A record type that maps one domain name (an alias) in the name field to another domain name (the canonical name) specified in the rdata field. A common example is the `www` subdomain which is provided as an alias to the zone domain name - users accessing `www.uni.edu` are referred to the zone apex "`uni.edu`".

- `DNAME` (domain redirection or delegation name record) [RW12a] - A `DNAME` record creates an alias for an entire subtree of the domain name tree, whereas the `CNAME` record creates an alias for a single name and not its subdomains. Zone administrators may want subtrees of `ee.uni.edu` and `elec.com` to contain the same information in DNS, in which case they can use a `DNAME` record.

If a nameserver uses a `CNAME` or `DNAME` record to respond to a query then the query is first rewritten into a new one. Consider the `CNAME` and `DNAME` records in Table 2.1. For the `CNAME` to apply, the input query domain name has to be the same as the `CNAME` record name (`www.uni.edu`), and the content of the record completely replaces the query name (`uni.edu`). For a `DNAME` to apply, on the other hand, the query domain name must be a subdomain (for example, `x.y.ee.uni.edu.`) of the name in the `DNAME` record (`ee.uni.edu`). The new query will preserve the subdomain and replace the ending part that matches the `DNAME` record name (producing the new query name `x.y.elec.com.`). The `DNAME` record's ability to rewrite the ending part of the query, regardless of what comes before it, turns out to be surprisingly powerful.

### 2.2.1 Glue Records

A zone file should typically not have any resource records below a zone cut [HSF19]. The example `uni.edu` zone file shown in Table 2.1 shows a zone cut at `cs.uni.edu`; the zone should

not have any records below it. DNS implementations should ignore such records even if such records are present in a user-provided zone file. Users should contact the computer science nameserver for any information about the `cs` sub-space. However, this leads to a circular dependency as the nameserver for `cs.uni.edu` is `n.cs.uni.edu`, which is in the `cs` sub-space, so contacting it to resolve its own name is not possible. To resolve this dependency in the in-domain nameserver instances, the zone file needs to have *glue records* to avoid failure of name resolution [Moc87a]. A glue record is an IPv4 or IPv6 record of the in-domain nameserver (`n.cs.uni.edu`) added to the zone file to help resolve DNS nameservers.

In-domain glue records and delegation records are not part of the authoritative records of the zone. The authoritative data for a zone is simply all of the RRs attached to all of the nodes from the top node (or apex) of the zone down to leaf nodes or nodes above cuts around the bottom edge of the zone [Moc87a]. A nameserver can return the glue records to the resolver only when returning the corresponding delegation `NS` records. Suppose the user sends a query for `n.cs.uni.edu` to the nameserver, which has the `uni.edu` zone file shown in Table 2.1. In that case, the nameserver should return the `NS` record since the queried name is below a zone cut, and additionally, return the glue `A` record and mark both the records as not authoritative. The nameserver should not return the `A` record alone, in which case, it signifies an authoritative response (and is wrong). Even though it has the IP address for `n.cs.uni.edu`, the resolver should technically contact the `n.cs.uni.edu` nameserver using the glue IP address to get an authoritative response. This is because the glue record is a hint or suggestion to avoid circular dependency and the actual records are with the `n.cs.uni.edu` nameserver.

### 2.2.2 Empty Non-Terminals (ENTs)

Empty non-terminals are domain names that own no resource records but have subdomains that do own records [HSF15]. In the `uni.edu` zone file shown in Table 2.1, the domain name `b.uni.edu` has no resource records in the zone file, but its subdomain `q.b.uni.edu` has

an IPv6 record. Implementations must handle ENTs properly; otherwise, it can lead to unexpected behaviors. If the query name is `b.uni.edu`, then the nameserver implementations must respond with an empty response but with no-error code (`NOERROR`). The nameserver should not respond with a non-existent `NXDOMAIN` error code, as it signals that the query name and its subdomains do not exist, which is incorrect [BH16].

## 2.3 Nameservers and Query Resolution

As mentioned in the introduction, a distributed collection of organizations manage the zones and provide the translation service through publicly accessible DNS servers, called nameservers, which are in turn identified by a domain name. Each nameserver serves one or more zones. Multiple servers also serve the same zone to ensure redundancy and availability. For example, the `ucla.edu` zone is available from servers like `ns1.dns.ucla.edu`, `ns2.dns.ucla.edu`, `ns3.dns.ucla.edu`, and `ns4.dns.ucla.edu`. Each nameserver can provide the data requested for a domain name directly or point to other nameservers. The user sends requests for information in the form of DNS queries which have three parts - **(a)** the resource target domain name, **(b)** the type of the resource, and **(c)** the class of the resource. For our purposes, the class is `IN` always, so the query is considered as a 2-tuple.

When a query arrives at a nameserver, it first checks the available zones to select the best matching zone and then uses the best matching records from that zone to answer the query [RW12a]. If no matching records are found, the nameserver returns an empty response with a non-existent domain (`NXDOMAIN`) error code signaling that the query domain name and any subdomains do not exist. If the selected best records are of type `A` or `AAAA` or `TXT`, then the resolver which sent the query gets the intended response. If the nameserver responds with an `NS` record, then the resolver must contact another nameserver. If there is a glue record associated with the `NS` record, then it is also returned along with the delegation record. The original query is rewritten if the best records are of type `CNAME` or `DNAME`. The rewrite

semantics of both the types were explained in the previous section with an example. After a rewrite, the nameserver first checks if it can resolve the new query. It restarts the query resolution process if it can; otherwise returns the rewritten query to the resolver.

## 2.4 Wildcard Resource Records

The resource records whose name starts with an asterisk label "∗" are given special treatment and are called wildcard resource records. Wildcard RRs can be thought of as instructions for synthesizing RRs. When the appropriate conditions are met, the nameserver creates RRs with an owner name equal to the query name and contents taken from the wildcard RRs [Moc87a]. At the highest level, the wildcard RRs match a query domain name that are not matched by other records in the zone.

The "appropriate conditions met" clause is nuanced and intricate. I will describe it with the help of several example input queries to a nameserver serving the `uni.edu` zone file shown in Table 2.1 which has a wildcard text record (∗.`uni.edu`):

- ⟨`n.uni.edu`, `TXT`⟩ - The nameserver uses the wildcard record for synthesis as there is no `n.uni.edu` record in the zone file. The nameserver synthesizes a new record using the wildcard record - "`n.uni.edu TXT` "Awesome"". The user is oblivious that the returned resource record is a synthesized record.

- ⟨∗.`uni.edu`, `TXT`⟩ - The nameserver returns the wildcard record as is without any synthesis as the query name matches exactly with the wildcard name. This should not be considered as a wildcard match, but rather an exact match.

- ⟨`n.uni.edu`, `A`⟩ - The query name matches the wildcard record but cannot be used for synthesis as the queried type does not match with record type.

- ⟨x.y.z.uni.edu, TXT⟩ - The nameserver uses the wildcard record for synthesis. The wildcard label, "∗" can match any number of labels, on the condition that all the labels in the wildcard record after "∗" are the same in the query domain name.

- ⟨x.a.uni.edu, TXT⟩ - The query name matches the wildcard record but the wildcard record cannot be used for synthesis because a.uni.edu exists in the zone file.

- ⟨x. ∗ .uni.edu, TXT⟩ - The nameserver cannot use the wildcard record because ∗.uni.edu exists in the zone file. To have a wildcard synthesis for this query, the zone file should have a wildcard record with the name ∗. ∗ .uni.edu.

For further detailed description of the wildcard records, the interested reader may refer the RFC 4592 [Lew06].

# CHAPTER 3

# DNS Formal Model

As mentioned in Section 1.4.1, we must first formalize the behavior of DNS to verify it exhaustively and to develop tools with solid correctness guarantees. This chapter provides formal, mathematical semantics for DNS, including both nameserver lookup and recursive resolution. I start by defining domain names, zones, resource records, DNS queries, answers, and record priority for a query using the mathematical notation in Section 3.1. In Section 3.2, I provide several constraints a zone file must satisfy to be considered a well-formed zone file. The DNS formal semantics are described in Section 3.3.

## 3.1 Definitions and Notations

In this section, I define the concepts explained in English and through examples in Chapter 2 in a formal notation using mathematical symbols. This helps avoid the ambiguity of the natural language and provides a precise way of understanding the semantics.

### 3.1.1 Domain Names

The DNS namespace which is a tree-like hierarchy naturally forms a partial order for domain names. For example, we say that `uni.edu` is the parent of `cs.uni.edu`, and accordingly `cs.uni.edu` is a child of `uni.edu`. Further, we consider `cs.uni.edu`, `uni.edu` and `edu` to be **prefixes** of `cs.uni.edu`. We model a domain name as a sequence of zero or more labels. The domain name `uni.edu` contains the labels `uni`, `edu`, and an implicit empty label ($\epsilon$) for the

(1) Domain prefix match

$$d_1 \simeq_j d_2 \stackrel{\text{def}}{=} (0 \leq j \leq \min(|d_1|, |d_2|)) \wedge (\forall i.\, 0 < i \leq j \implies d_1[i] = d_2[i])$$

(2) Maximal prefix match

$$\max_{\simeq}(d_1, d_2) \stackrel{\text{def}}{=} \max\ \{j \mid d_1 \simeq_j d_2\}$$

(3) Domain partial order

$$d_1 \leq d_2 \stackrel{\text{def}}{=} d_1 \simeq_{|d_1|} d_2$$

(4) Domain wildcard match

$$d_1 \in_* d_2 \stackrel{\text{def}}{=} (|d_2| \leq |d_1|) \wedge (d_1 \simeq_{(|d_2|-1)} d_2) \wedge d_1[|d_2|] \neq * = d_2[|d_2|]$$

(5) Resource record matches query

$$\textsc{Match}(r, q) \stackrel{\text{def}}{=} \mathrm{dn}(r) \leq \mathrm{dn}(q) \vee \mathrm{dn}(q) \in_* \mathrm{dn}(r)$$

(6) Resource record Rank

$$\textsc{Rank}(r, q, z) \stackrel{\text{def}}{=} \langle \mathcal{I}\left(\textsc{Match}(r, q)\right), \mathcal{I}\left(\mathrm{ty}(r) = \texttt{NS} \wedge \mathrm{dn}(r) \neq \mathrm{dn}(z)\right), \max_{\simeq}(\mathrm{dn}(r), \mathrm{dn}(q)), \mathcal{I}\left(\mathrm{dn}(q) \in_* \mathrm{dn}(r)\right)\rangle$$

(7) Resource record order

$$r_1 <_{q,z} r_2 \stackrel{\text{def}}{=} \textsc{Rank}(r_1, q, z) \lhd \textsc{Rank}(r_2, q, z)$$

**Figure 3.1:** Common DNS definitions, and notations.

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 $\epsilon$, which represents an empty string (*e.g.*, `uni.edu` is written as `uni` $\circ$ `edu` $\circ\ \epsilon$). The sequence with only the empty domain name ($\epsilon$) is called the root domain. 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 3.1 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 $\epsilon$). For example, $\texttt{uni} \circ \texttt{edu} \circ \epsilon \simeq_1 \texttt{edu} \circ \epsilon$. 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, $d_1 \leq d_2$ iff $d_1$ is a prefix of $d_2$ ($d_1 \simeq_{|d_1|} d_2$). We use the notation $d_1 \in_* d_2$ to mean that $d_1$ matches the wildcard domain $d_2$ (4). A domain name $d_1$ matches a wildcard domain $d_2$ if three conditions are met: **(a)** $d_2$ is of same length or shorter than $d_1$, **(b)** $d_1$ has the same labels as in $d_2$ after the "$*$" label, and **(c)** $d_1$ does not have a "$*$" in the same position where $d_2$ has the wildcard "$*$". There is a subtle difference between wildcard matching and synthesis. We can decide whether a wildcard domain matches a domain name using the wildcard record alone, whereas whether the nameserver will use that wildcard record for synthesis to respond to a query depends on the other resource records in the zone file. Section 2.4 provides some examples for wildcard record matching and synthesis.

### 3.1.2 Zones and Resource Records

A DNS zone $z \in \textsf{ZONE}$ is a set of resource records ($\textsf{ZONE} = \mathcal{P}(\textsf{RECORD})$). We use the symbol $\mathcal{P}(\textsf{RECORD})$ 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. Section 3.2 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 $r \in \textsf{RECORD} = \langle d, t, \tau, a, b \rangle$ as a tuple with five components: **(a)** a domain name $d \in \textsf{DOMAIN}$, **(b)** a record type $t \in \textsf{TYPE} = \{\texttt{A}, \texttt{AAAA}, \texttt{MX}, \texttt{NS}, \texttt{DNAME}, \texttt{CNAME}, \texttt{SOA}, \ldots\} \cup \{\texttt{N}\}$ representing either the kind of data the record holds (*e.g.*, `AAAA` for an IPv6 address) or the type `N` to represent empty data, **(c)** 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, **(d)** the answer $a \in \Sigma^*$ which gives the DNS result as a string, and **(e)** finally a boolean value $b$ ($T$ or $F$) that marks whether a record was synthesized from

another. All the resource records are assumed to be of the `IN` class, and therefore the class is not modeled as one of the tuple components.

We explicitly model empty non-terminals [Lew06, Moc87a, HSF15] as resource records containing the type `N` to avoid the pitfalls mentioned in Section 2.2.2. We write $\text{dn}(r)$ for the domain name of record $r$, $\text{ty}(r)$ for the type, $\text{ttl}(r)$ for the TTL, $\text{ans}(r)$ for the record answer, and $\text{synth}(r)$ for whether the record was synthesized.

### 3.1.3 DNS Queries

A DNS query $q = \langle d,\ t \rangle$ is a tuple containing a domain name $d \in \textsf{DOMAIN}$ and a query type $t \in \textsf{TYPE}$. A user that needs the IPv4 address might send a query $\langle \texttt{www.uni.edu}, \texttt{A} \rangle$ 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.

### 3.1.4 DNS Answers

We model a DNS answer $a = \langle x, y \rangle$ as a pair of a tag $x \in \{\textsc{Ans}, \textsc{AnsQ}, \textsc{Ref}, \text{NX}, \textsc{Refused}, \textsc{ServFail}\}$, and data $y$. Each tag indicates the type of answer and is described below:

- Ans (Answer): The nameserver returns a set of resource records after the query lookup. The data $y$ of the answer would have resource records $R$ holding pertinent information to the query.
- AnsQ (Answer + Query Rewrite): The nameserver rewrites the query using a set of resource records $R$ resulting in the new query $q'$. The data $y$ field of the answer, in this case, would be a pair $\langle R, q' \rangle$.
- Ref (Referral): The data $y$ of the answer is a set of delegation (`NS` and glue) resource records $R$ indicating the query is delegated to other nameservers.

- NX (NXDOMAIN): The nameserver does not return any records ($y = \emptyset$), but indicates that the queried domain does not exist with this tag.
- REFUSED (Refused): The nameserver refuses to respond to the query as there is no relevant zone file ($y = \emptyset$).
- SERVFAIL (Server Failure): There is no nameserver available to handle the query, and the DNS is assumed to return a failure message with this tag ($y = \emptyset$).

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).

### 3.1.5 Resource Record Priority for a Query

The remaining definitions (5) – (7) in Figure 3.1 are used to define the order in which DNS prioritizes resource records for a given query. MATCH (5) determines if a record is relevant for a given query (*i.e.*, a potential match). A resource record is relevant for a query if the domain name of the record is a prefix of the query domain name, or the record is a wildcard record and the query domain name matches the wildcard domain. 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 (7) 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 (Section 3.3.1)
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. finally whether it is a wildcard match as a tiebreaker.

The first condition ensures that a nameserver uses only matching records to create an answer for the input query. The second condition is necessary for avoiding returning glue records, as authoritative records as the zone cut `NS` records take priority in those cases (Section 2.2.1 explains it with the help of an example). The third condition allows selecting the record with the most number of labels in common with the query. The final condition prioritizes a wildcard match when everything else is the same.

## 3.2 Zone Well-formedness Constraints

A collection of resource records $R$ is considered as a well-formed zone $z$ if it satisfies all of the following conditions:

(1) There should be exactly one `SOA` record.

$$|\{r \in R \mid \text{ty}(r) = \texttt{SOA}\}| = 1$$

(2) No record can be a synthesized one.

$$\langle d, t, \tau, a, b\rangle \in R \implies b = F$$

(3) The domain name of the `SOA` record should be a prefix of the domain name of all the records in $R$.

$$\langle d, \texttt{SOA}, \tau, a, b\rangle \in R \wedge \langle d', t, \tau', a', b'\rangle \in R \implies d \leq d'$$

(4) The answer of a `CNAME`, `DNAME` and an `NS` record should be a valid domain name.

$$\langle d, t, \tau, a, b\rangle \in R \wedge t \in \{\texttt{CNAME}, \texttt{DNAME}, \texttt{NS}\} \implies a \in \textsf{DOMAIN}$$

(5) There can be only one `CNAME` record for a domain name.

$$\langle d, \texttt{CNAME}, \tau, a, b\rangle \in R \wedge \langle d, \texttt{CNAME}, \tau', a', b'\rangle \in R \implies \tau = \tau' \wedge a = a' \wedge b = b'$$

(6) If there is a `CNAME` record for a domain name, then there cannot be any other record type for that domain name.

$$\langle d, \texttt{CNAME}, \tau, a, b\rangle \in R \wedge \langle d, t, \tau', a', b'\rangle \in R \implies t = \texttt{CNAME}$$

(7) There can be only one DNAME record for a domain name.

$\langle d, \texttt{DNAME}, \tau, a, b\rangle \in R \wedge \langle d, \texttt{DNAME}, \tau', a', b'\rangle \in R \implies \tau = \tau' \wedge a = a' \wedge b = b'$

(8) A domain name cannot have both DNAME and NS records unless there is an SOA record.

$\langle d, \texttt{DNAME}, \tau, a, b\rangle \in R \wedge \langle d, \texttt{NS}, \tau', a', b'\rangle \in R \implies \langle d, \texttt{SOA}, \tau'', a'', b''\rangle \in R$

(9) If there is a DNAME record for a domain name $d$, then there cannot be any records with domain names for which $d$ is a proper prefix.

$\langle d, \texttt{DNAME}, \tau, a, b\rangle \in R \wedge \langle d', t, \tau', a', b'\rangle \in R \wedge d \neq d' \implies d \not\leq d'$

(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.

$\langle d, \texttt{NS}, \tau, a, b\rangle \in R \wedge \neg\exists\ \tau_s, a_s, b_s.\ \langle d, \texttt{SOA}, \tau_s, a_s, b_s\rangle \in R \wedge \langle d', t, \tau', a', b'\rangle \in R \wedge d < d' \implies t \neq \texttt{NS}$

(11) Wildcard domain names can not have a DNAME or an NS record.

$\langle d, t, \tau, a, b\rangle \in R \wedge d[|d|] = * \implies t \neq \texttt{DNAME} \wedge t \neq \texttt{NS}$

(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.)

$\langle d, t, \tau, a, b\rangle \in R \wedge \langle d_s, \texttt{SOA}, \tau_s, a_s, b_s\rangle \in R \wedge d \neq d_s \implies \exists \langle d', t, \tau, a, b\rangle \in R \wedge d' < d \wedge |d'| = |d| - 1$

## 3.3 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.*, `n.cs.uni.edu`). We leave nameservers as opaque objects and associate them with other information through functions.
- $\Theta \subseteq S$ is a set of "root" nameservers for $S$.
- $\Gamma : S \to \mathcal{P}(\texttt{ZONE})$ 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.

### 3.3.1 Authoritative Nameserver semantics

Given a set of zone files $Z$ and a query $q$, the definition of ServerLookup at the bottom of Figure 3.2 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 zone files 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. The function $\mathcal{N}$ returns not more than one zone file as the nameserver cannot have two zone files with the same zone domain.

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

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

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

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

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

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

$$\begin{aligned} \textsc{ServerLookup}(Z,q) &= \begin{cases} \textsc{ZoneLookup}(z,q) & \mathcal{N}(Z,q) = \{z\} \\ \langle \textsc{Refused}, \emptyset \rangle & \mathcal{N}(Z,q) = \emptyset \end{cases} \\ \mathcal{N}(Z,q) &= \max_{\mathrm{dn}} \{z \in Z \mid \mathrm{dn}(z) \leq \mathrm{dn}(q)\} \\ \textsc{ZoneLookup}(z,q) &= \textsc{RRLookup}(\{r \in \max_{<_{q,z}} z\}, q, z) \end{aligned}$$

**Figure 3.2:** Authoritative DNS lookup semantics.

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 (7) in Figure 3.1. The set of records passed to RRLookup will necessarily have the same domain name, *i.e.*, $\mathrm{dn}(r_1) = \mathrm{dn}(r_2)$ for any $r_1$, $r_2$ in the set. However their types may differ. Thus, for such a set $R$, we simply write $\mathrm{dn}(R)$ 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:

- an answer (Ans), if the resource records $R$ are sufficient to answer the query $q$
- a referral (Ref), if records $R$ cannot answer the query $q$ but indicate who might have the answer
- 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 the resource record $r'$
- an error message (NX), indicating that the domain does not exist.

RRLookup implements the DNS resolution process described as the server algorithm in RFC 6672 [RW12a]. Note that we exclude a few steps of the server algorithm since the formal model does not capture dynamic elements like caches. The search for the nearest zone to the query is captured by $\mathcal{N}$ stated earlier. When the records' domain name exactly matches the query, the ExactMatch function is applied. Otherwise, if it is a wildcard domain that matches the query domain, the WildcardMatch case will apply. 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. If no such record exists, DNS will delegate the query to another nameserver if it has an `NS` record (Delegation). Finally, if all else fails, the nameserver will return `NXDOMAIN` (non-existent domain).

The ExactMatch and WildcardMatch cases are both broken down further into several cases. For the ExactMatch case, if there is an authoritative record with the same

type as the query, then the nameserver will simply return this record. Of all the records passed to RRLookup, a zone $z$ is authoritative for all the records except for `NS` records (zone cut) not accompanied by an `SOA` record and the glue records (Section 2.2.1). Otherwise, if there is a `CNAME` record, 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. Finally, if all else fails, it will simply return an answer with no information ($\emptyset$).

The WildcardMatch case is similar to the ExactMatch 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 $*.\texttt{com}$ generates a (cachable) synthesized record for `email.com`.

The Rewrite case for `DNAME` records returns AnsQ with a tuple containing **(a)** the `DNAME` record and a synthesized `CNAME` record, and **(b)** 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 ($\text{dn}(q)[\text{dn}(r) \mapsto \text{ans}(r)]$). For example, RRLookup of $(\{r_1 = \langle \texttt{ee} \circ \texttt{uni} \circ \texttt{edu} \circ \epsilon,\ \texttt{DNAME},\ 500,\ \texttt{elec} \circ \texttt{com} \circ \epsilon,\ F\rangle\}, \langle \texttt{foo} \circ \texttt{ee} \circ \texttt{uni} \circ \texttt{edu} \circ \epsilon, \texttt{A}\rangle, z)$ will return $\langle \text{AnsQ}, \langle R', \langle \texttt{foo} \circ \texttt{elec} \circ \texttt{com} \circ \epsilon, \texttt{A}\rangle\rangle\rangle$, where $R' = \{r_1\} \cup \{\langle \texttt{foo} \circ \texttt{ee} \circ \texttt{uni} \circ \texttt{edu} \circ \epsilon,\ \texttt{CNAME},\ 500,\ \texttt{foo} \circ \texttt{elec} \circ \texttt{com} \circ \epsilon,\ T\rangle\}$. The new query name would be the answer field of the `CNAME` record, which is obtained by substituting the matching prefix ($\texttt{ee} \circ \texttt{uni} \circ \texttt{edu} \circ \epsilon$) of the original query ($\texttt{foo} \circ \texttt{ee} \circ \texttt{uni} \circ \texttt{edu} \circ \epsilon$) with the answer field ($\texttt{elec} \circ \texttt{com} \circ \epsilon$) of the `DNAME` record. The DNS adds these `CNAME` records to the answer to facilitate caching — future queries are rewritten based on the cached `CNAME` record. The Delegation case returns the `NS` records along with the necessary `A` and `AAAA` glue records.

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

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

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

**Figure 3.3:** DNS Recursive resolution semantics.

### 3.3.2 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 3.3) explained earlier with an example in Section 1.1. 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 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 $\bot$ due to a nameserver lookup failure in $\Omega$ 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 ($\{a\}$).

# CHAPTER 4

# Finding RFC Compliance Bugs in DNS Nameservers via Automatic Test Generation

Writing an efficient, high-throughput, multithreaded DNS nameserver implementation that is also bug-free and compliant with the multiple DNS RFC specifications is difficult and challenging. As a result, nameserver implementations frequently suffer from incorrect or implementation-specific behavior that causes outages [Fry14, Tun19, Yor15], security vulnerabilities [Kov18, Ras16], and more [Bin22, Pow22, NSD22].

In this chapter, I present the first approach for identifying RFC compliance (protocol implementation) errors in DNS nameserver implementations, by automatically generating test cases that cover a wide range of RFC behaviors. The key technical challenge is the fact that a DNS test case consists of both a query and a zone file, which is a collection of resource records that specify how queries should be handled. Zone files are highly structured objects with various syntactic and semantic well-formedness requirements, and the query must be related to the zone file for the test even to reach the core query resolution logic.

Existing standard automated test generation approaches are not suitable for our needs, as illustrated on the top of Figure 4.1. Fuzz testing is scalable but has well-known challenges in navigating complex semantic requirements and dependencies [GKL08, CC18], which are necessary to generate behavioral tests for DNS. As a result, fuzzers for DNS only generate queries and hence are used only to find parsing errors [Pat22, Foo15, Cam19, St10]. Symbolic execution [Kin76] can, in principle, generate DNS tests that achieve high code coverage but, in practice, suffers from the well-known problem of "path explosion" [CDE08, GKL08, CC18] that

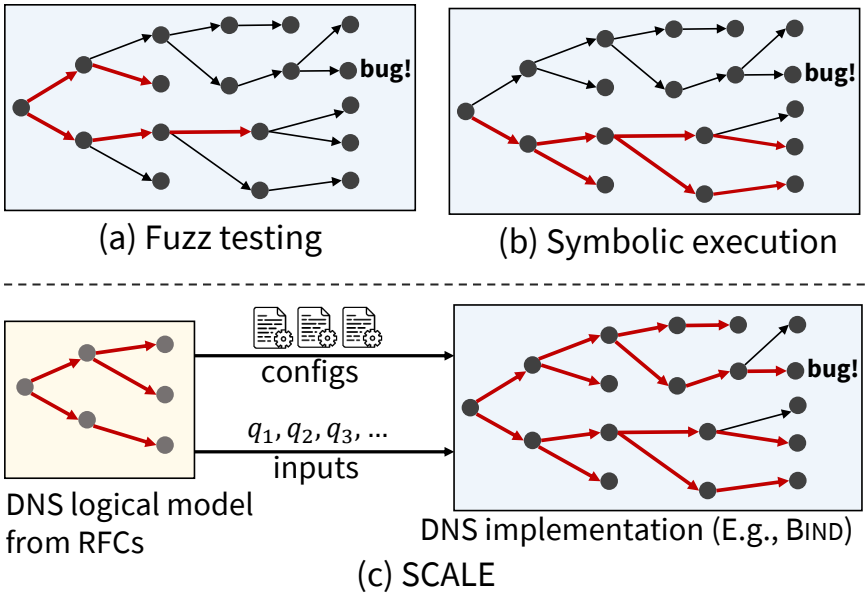



**Figure 4.1:** Overview of different automated testing approaches. Tested implementation paths are shown in red. (a) Fuzz testing is scalable but is often unable to navigate complex input requirements. (b) Symbolic execution can solve for input conditions but suffers from path explosion and has difficulty with complex data structures and program logic, and will thus only typically explore a small subset of possible program paths. (c) SCALE uses a logical model of the DNS RFCs to guide symbolic search toward *many* different *logical behaviors*.

limits scalability and coverage. As a result, symbolic execution has only been used to identify generic errors like memory leaks in individual functions within nameserver implementations, again avoiding the need to generate zone files [RE15].

My approach to automated testing for DNS nameservers, which I call **SCALE** (Small-scope Constraint-driven Automated Logical Execution), *jointly* generates zone files and the corresponding queries, does so in a way that is targeted toward covering many different RFC behaviors, and is applicable to black-box DNS nameserver implementations. The key insight underlying SCALE is that we can use the existing RFCs to define a model of the logical behaviors of the DNS resolution process and then use this model to guide test generation.

SCALE symbolically executes a program version of the formal semantics of DNS described in Chapter 3 to generate well-formed tests for black-box DNS nameservers — each test consisting of a well-formed zone file and a query that together cause execution to explore a particular RFC behavior.

Symbolic execution of our logical model is still fundamentally unscalable — there are an unbounded number of possible execution paths, they grow exponentially in the size of the zone file, and expensive constraint solvers must be used to generate a test case for each path. I therefore bound the generated zone files to contain a very small number of resource records and short domain names — a maximum of 4 for each of these in our experiments, which is much smaller than real-world zone files. However, I provide experimental evidence of the existence of a *small-scope* property [Jac02], meaning that many interesting behaviors can be covered with small tests. First, each return point in our logical model can be reached with a test where the length of domain names and the number of records in the zone file is at most 3. Each return point represents a distinct RFC-specified scenario for DNS resolution (*e.g.*, a particular flavor of query rewrite). Second, while increasing this constant from 2 through 4 increased the number of errors that the tool identified, no new errors were found in a sample of paths that required size 5. This finding makes sense because, while zone files can contain a large number of records, the number of records that are relevant to any particular query tends to be small.

I have used the SCALE approach as the basis for a tool called FERRET[1] for automated testing of DNS nameserver implementations (Figure 4.2). FERRET generates tests using our logical model, which we have implemented in a modeling language called Zen [BM20] that has built-in support for symbolic execution. FERRET then performs *differential* testing by running these tests on multiple DNS nameserver implementations and comparing their results to one another. In this way FERRET can identify RFC violations, crashes, as well as situations where the RFCs may be ambiguous or underspecified, leading to implementation-

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

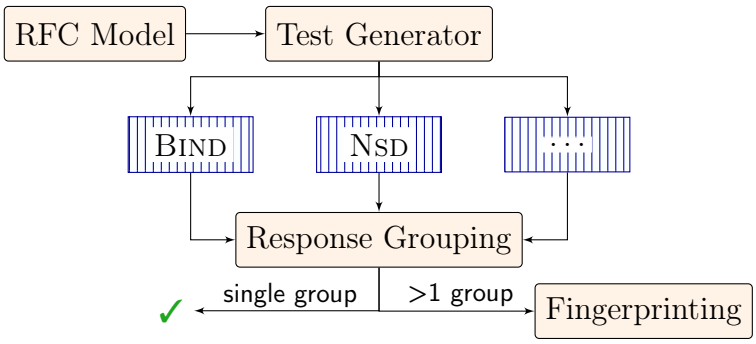



**Figure 4.2:** FERRET system architecture

dependent behavior. Because DNS implementers strive for behavioral consistency among their implementations [Pel20], any test that produces divergent results among the implementations represents a likely error. However, there can be orders-of-magnitude fewer root causes than divergent tests, so as a final step we provide a simple but effective technique to help users with bug deduplication. I create a *hybrid fingerprint* for each test, which combines information from the test's path in the Zen model with the results of differential testing, and then group tests by fingerprint for user inspection.

Using FERRET, in just a few hours I generated over 12.5K valid test cases[2] with a maximum zone-file size of 4 records. Running these tests on 8 different open-source DNS nameserver implementations, I found that the implementations' behaviors only completely agreed on 35% of the tests. Our fingerprinting technique reduced the remaining cases to roughly 75 groups. Because my executable model includes a specification of the well-formedness conditions for zone files, I also leveraged Zen to systematically generate zone files that violate one of these conditions (detailed in Section 4.3). I generated 900 invalid zone files of which 184 resulted in some difference among implementations.

Inspecting tests from each fingerprinted group resulted in the discovery of 30 unique bugs across the different implementations. Developers have confirmed all of them as actual bugs

[2] Test cases: `https://github.com/dns-groot/FerretDataset`

and fixed 21 of them, at the time of writing. The most severe bug FERRET found was a subtle combination of zone file and query that an attacker could easily use to crash both BIND nameservers *and* resolvers remotely. I engaged in a secure disclosure process, after which the developers fixed the issue and then publicly disclosed the vulnerability, through a CVE (CVE-2021-25215) [GKD21, Dat21] rated with high-severity.

In summary, I make the following major contributions:

(§ 4.2) I present the first automated approach to identifying RFC violations in black-box DNS nameservers. A unique feature of my approach, SCALE, is the joint generation of zone files and queries to produce high-coverage behavioral tests.

(§ 4.4) I describe an implementation of my approach in FERRET that combines SCALE with differential testing.

(§ 4.4) I present a novel fingerprinting approach for bug deduplication that takes advantage of my RFC model to help triage bugs.

(§ 4.5) I present an evaluation from testing 8 different open-source DNS nameserver implementations with tests generated by FERRET consisting of over 13.5K zone files, which resulted in the discovery of 30 new unique bugs and no false positives. Section 4.6 explain some of the bugs FERRET found and the positive responses from developers of different DNS implementations on my bug reports.

## 4.1 Finding DNS Errors with FERRET

The goal of FERRET is to automatically generate high-coverage query and zone file inputs to find behavioral errors in DNS nameserver implementations. In this section, I illustrate both the challenges in doing so and FERRET's capabilities through two example errors that it automatically found in BIND.

### 4.1.1 Bug #1: BIND sibling glue records bug

We have already seen a brief version of the following bug in Section 1.4.2.

FERRET generated the following test case, which identified a previously unknown performance bug in BIND [KAK21b].[3]

```
    campus.edu.   SOA   ns1.com. admin.uni.edu. 11 600 30 400 500
 cs.campus.edu.   NS    ns1.campus.edu.
ns1.campus.edu.   A     1.2.3.4
```

**Query:** ⟨`www.cs.campus.edu.`, `A`⟩

In this test case, the query matches the `NS` record in the zone file, which delegates the query to another nameserver, `ns1.campus.edu`. However, that nameserver happens to be a sibling of `cs.campus.edu` (as they are both directly under `campus.edu`), and the zone file contains a glue record (Section 2.2.1), for the nameserver's IP address. NSD, KNOT, and POWERDNS correctly return the `NS` record along with the glue record, avoiding extra round-trips to determine the nameserver's IP address, while BIND returns only the `NS` record. Returning the sibling glue record is not compulsory, but our test case exposed two unrelated errors that can negatively affect the performance of many queries.

After we filed the issue the BIND developers confirmed the bug saying, "This report turns out to be very interesting..." Briefly, BIND uses a "glue cache" to speed up the identification of glue records, but it had two bugs. First, if the cache lookup fails, then glue records are supposed to be searched for in the zone file, but this was not happening. Second, glue records for siblings domain nameservers were accidentally never searched for at all.

This example illustrates the challenges of identifying nameserver behavior errors. Even though the zone file has only a few records, they have complex dependencies. First, there must be a delegation of the query to another nameserver. Second, that nameserver must be

[3] Note that we have renamed the labels for all the example bugs for clarity.

in the same zone. Third, that nameserver must be a sibling domain. Fourth, there must be a glue record for that domain in the zone. Given these dependencies, it is understandable that prior testing techniques did not uncover these bugs. Further, by comparing the outputs from multiple implementations, FERRET is able to identify this test case as potentially buggy behavior despite receiving a valid response from BIND.

### 4.1.2 Bug #2: BIND crash

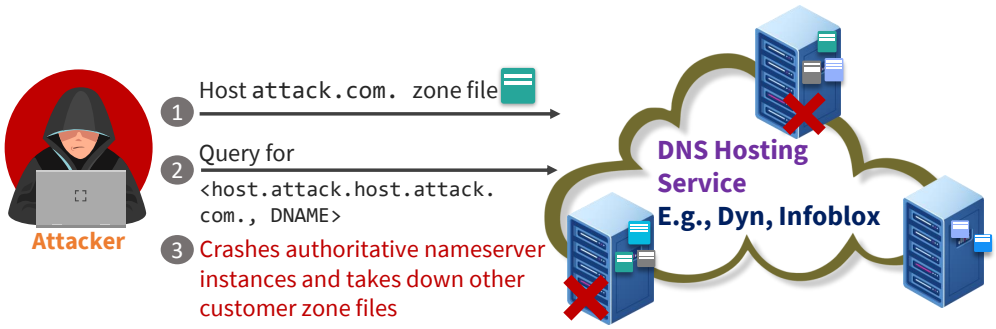
As another, more dire example, consider the following zone file that FERRET generated. The zone file is invalid due to having two identical records, but BIND, NSD, and KNOT accept the zone file and make it valid by ignoring the duplicate record.

```
      attack.com.   SOA     ns1.com. admin.uni.edu. 11 600 30 400 500
      attack.com.   NS      ns1.outside.com.
      attack.com.   NS      ns1.outside.com.
host.attack.com.    DNAME   com.
```

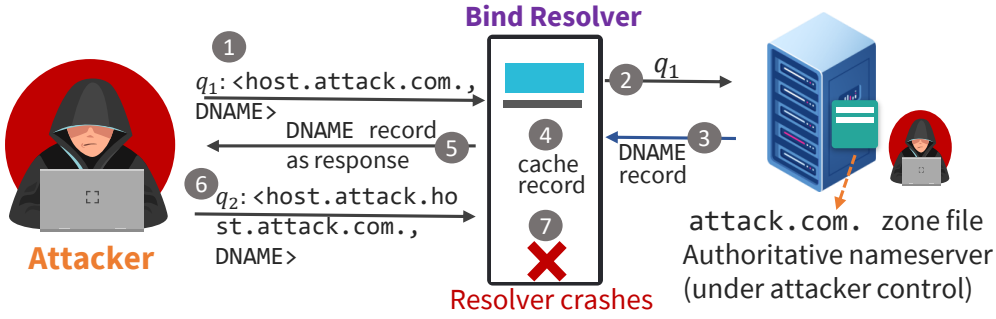
**Query:** ⟨`www.cs.campus.edu.`, `A`⟩

FERRET generated multiple queries for this zone file (Section 4.3) and the one showed above caused BIND to crash.

In this test case, the `DNAME` record is applied to rewrite any queries ending with `host.attack.com` to end with just `com`, so the query that FERRET generated is rewritten to the new query `host.attack.com`. The nameservers add the `DNAME` record and rewritten query to the response before resolving the new query. The new query exactly matches the same `DNAME` record, so implementations are expected to return the current response. All implementations except BIND behaved as expected. BIND did not respond, and the query timed out. Inspecting the logs, we found that the server crashed with an assertion failure due to an attempt to add the same `DNAME` record to the response twice.




**(a)** Attack on a DNS hosting service using Bind nameserver implementation.




**(b)** Attack on a public Bind DNS Resolver.

**Figure 4.3:** DNAME attack targeting the DNS hosting services (a) and the public BIND based recursive resolvers (b).

This error constitutes a critical security vulnerability. I next describe two scenarios to show how this failed assertion check can be exploited remotely by an attacker.

**Scenario 1 - Attack on a DNS hosting service that uses Bind:** DNS hosting services using BIND's authoritative nameserver implementation (*e.g.*, Dyn [Inc22]) are vulnerable to this attack. An attacker can upload the above zone file to the authoritative server instances through the hosting service. Then, when the above query is requested, the server instances will crash as shown in Figure 4.3(a). Since a server instance will generally be serving zone files from multiple customers, such a crash will take down the zones for all customers hosted

at that nameserver. This provides a method for attackers to trivially and remotely initiate a denial of service attack against customers hosted by such a service.

**Scenario 2 - Attack on a public Bind DNS resolver:** In this second scenario, the attacker can crash any public DNS resolver based on BIND, thereby constituting, as stated by the BIND security team, an "easily-weaponized denial-of-service vector." As illustrated in Figure 4.3(b), the attacker purchases, registers, and controls the `attack.com` zone and its authoritative servers. The attacker then simply requests the `DNAME` record from a public recursive resolver running BIND, which attempts to fetch the result from the attacker's authoritative server. This record is cached, and then the test query is sent to the resolver. The resolver uses the cached `DNAME` record and ultimately crashes as described earlier. In some estimates, BIND accounts for over half of all DNS resolvers in use [KHB15], which means that attackers could effectively initiate a simple distributed denial of service (DDoS) attack against the numerous ISPs and public resolvers available to end users.

**Disclosure:** After discovering the `DNAME` attack, I initiated a responsible disclosure procedure with the BIND maintainers. Understanding the attack severity, they requested that I keep the issue confidential until they worked through their process to patch and then disclose the bug to the relevant parties in a controlled manner. BIND released a Common Vulnerabilities and Exposure (CVE-2021-25215) [Dat21, GKD21], with a "high severity" rating and asked developers and users to upgrade to the patched version. The attack affected all maintained BIND versions, which in turn affected RHEL, Slackware, Ubuntu, and Infoblox.

## 4.2 Methodology

In this section I overview my methodology for generating high-coverage tests for DNS nameserver implementations and discuss how I address several technical challenges.

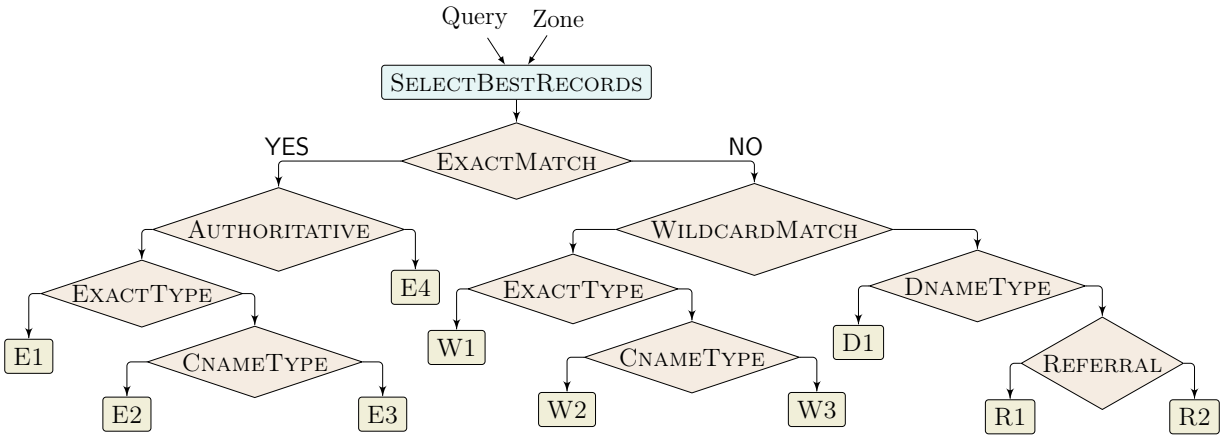



**Figure 4.4:** Abstract representation of the Authoritative DNS decision tree used to respond to a user query.

### 4.2.1 SCALE Approach

As illustrated by the examples in the previous section, the inputs to a DNS nameserver — a query and a zone file containing a set of records — are highly structured. Further, records can be of many different types and have many different kinds of dependencies among them. Therefore, an effective approach to automatically identifying RFC violations must be able to generate *valid* inputs that meet the required structural and semantic constraints of the domain, and it must also be able to explore different combinations of record types and features in a systematic way. To solve this joint generation problem, our approach, SCALE (Small-scope Constraint-driven Automated Logical Execution) leverages a specification of the DNS nameserver logic to drive test generation. Specifically, we have created an *executable* version of the DNS specification described in Chapter 3 and generate tests through *symbolic execution* [Kin76] on this executable specification. Symbolic execution is a static analysis technique that enumerates execution paths in a program and uses automated constraint solvers to produce an input that will take each enumerated path, thereby generating tests that cover many different program behaviors.

While the end-to-end behavior of a DNS query lookup can require contacting many nameservers, we employ a *compositional* approach that only generates tests for a single nameserver in isolation. Because our formal model considers the space of all inputs to the nameserver that could be produced by the rest of the system, and because the "next step" delegation of the resolution process is captured in the output at a single nameserver, this approach still allows us to generate tests for all behaviors of the end-to-end DNS. In other words, any implementation bug that exists in a DNS nameserver implementation can be found using our approach. In general, a downside of compositional testing is that it can lead to false positives if the tester considers input states that are, in reality, unreachable with respect to the rest of the system. However, in the case of DNS, nameservers keep no internal state — the response they provide is based only on the supplied query and configuration. This stateless nature implies that compositional testing will not incur any false positives.

Hence our formal semantics focuses on query lookup at a single nameserver, which we model as a stateless function that takes a user query and a zone file and produces a DNS response. This function was introduced as ZoneLookup in the formal model Chapter 3 and an abstract view of it is shown in Figure 4.4. Given the input query and zone, DNS will first select the closest matching records in the zone for the query using the SelectBestRecords function and then follow the decision logic laid out in the figure using these records. Each leaf node represents a unique case in the DNS. For example, the tree shows four different cases of exact matches, labelled E1 through E4. E1 corresponds to the first case in the definition of the ExactMatch function in Figure 3.2. E2 - E4 also map to the remaining cases of the definition in a similar way. Symbolic execution of our query-lookup function generates inputs that drive the function down different execution paths, thereby enabling us to systematically explore the space of DNS behaviors and feature interactions.

*Example:* Consider the path in Figure 4.4 to the leaf labelled R1. In order to reach that leaf, the selected records must not contain one with either an exact match or a wildcard match on the query domain name. Further, there should not be a `DNAME` match but should be one of

```
Zen<Response> QueryLookup(
  Zen<Query> q,
  Zen<Zone> z)
{
  var records = SelectBestRecords(q, z);
  var rname = records.At(0).Value().Name();
  var types = records.Select(r => r.Type());

  return If(
    rname == q.Name(),
    ExactMatch(records, q, z),
    If(
      IsWildcardMatch(q.Name(), rname),
      WildcardMatch(records, q, z),
      If(
        types.Any(t => t == RType.DNAME),
        Rewrite(records, q),
        If(
          And(types.Any(t => t == RType.NS),
              Not(types.Any(t => t == RType.SOA))),
          Response(Tag.R1,
            Delegation(records, z), Null<Query>()),
          Response(Tag.R2, empty, Null<Query>())
    ))));
}
```

**Figure 4.5:** Record lookup model in C# using Zen.

type `NS` (Referral). Finally, while not shown in the figure, when preparing a response to the query the function will also search for a glue record if the `NS` target is in the same zone. Solving all of these constraints caused symbolic execution to automatically generate the first test case shown in Section 4.1, which identified two errors in Bind.

### 4.2.2 An Executable Model of DNS

I have created an implementation of the formal semantics of query lookup presented in Chapter 3 as a program in a modeling language called Zen [BM20], a domain-specific language (DSL) embedded in C#. To illustrate this approach, I show several components

```
26 Zen<Response> WildcardMatch(
27   Zen<IList<ResourceRecord>> rrs,
28   Zen<Query> q,
29   Zen<Zone> z)
30 {
31   var exact = rrs.Where(r => r.Type() == q.Type());
32   var record = rrs.At(0).Value();
33   var newQuery = Query(record.RData(), q.Type());
34   var exactSyn = RecordSynthesis(exact, q.Name());
35   var cnameSyn = RecordSynthesis(rrs, q.Name());
36
37   return If(
38     exact.Length() > 0,
39     Response(Tag.W1, exactSyn, Null<Query>()),
40     If(
41       rrs.Any(r => r.Type() == RType.CNAME),
42       Response(Tag.W2, cnameSyn, Some(newQuery)),
43       Response(Tag.W3, empty, Null<Query>())
44     ));
45 }
```

**Figure 4.6:** Wildcard match model in C# using Zen.

of my model. Figure 4.5 shows the model's main query-lookup function, as depicted in Figure 4.4. The function first selects the best records (Line 5) and then tests if the query domain name is equal to the records' domain name (Line 10). If so, then this is an exact match and the model calls out to a helper function to specifically handle the `ExactMatch` subcase (Line 11). Similarly, if the query domain name is a wildcard match for the record domain name (Line 13), then I invoke the `WildcardMatch` subcase (Line 14). I show the implementation of wildcard matching in Figure 4.6. This function implements the case where the best matching record is a wildcard, properly handles interactions with `CNAME` records, and synthesizes the correct records for use in the resolver cache.

Our complete executable model consists of 520 lines of C# code. The model can also easily extend to new DNS RFCs that would be added in the future. Similarly, if an organization

has a particular way of resolving RFC ambiguities or purposely deviates from the RFCs in specific ways, the organization can modify the logical model to reflect that intent.

I chose to implement our formal model in Zen because it has built-in support for symbolic execution. In Zen, certain inputs can be marked as *symbolic*, and the ferret will then leverage SMT solvers [DB08] to produce concrete values for these inputs that drive the program down different execution paths. In our code examples, the `Zen<T>` type for inputs has the effect of marking them as symbolic. The tests produced by symbolic execution can then be used to test any DNS nameserver implementation. However, making symbolic execution effective required us to address several challenges, which I describe in the rest of this section.

### 4.2.3 Generating Valid Zone Files

The first challenge that I encountered is that zone files must satisfy several constraints in order to be considered well-formed. For instance, if there is a `DNAME` record in a zone file for `math.uni.edu`, then no other records below this domain name may exist, for any record type (*e.g.*, an `A` record for `fun.math.uni.edu` is not allowed). The DNS RFCs define many such constraints as a way to eliminate ambiguous or useless zones, as shown in Table 4.1. The table presents a summary of the conditions presented earlier using formal notation in Section 3.2. Naively performing symbolic execution will produce many zone files that are not well formed. Further, DNS implementations typically preprocess zone files to reject ill-formed zones, thereby failing to test the intended execution path of the query lookup logic.

Fortunately, the SCALE approach admits a natural solution to this problem. I have formalized all of the DNS zone validity conditions as predicates in Zen. Whenever Zen's symbolic execution engine produces a constraint representing the conditions under which the query lookup function takes a particular execution path, I conjoin these predicates to that constraint before Zen passes it off to an automated constraint solver. In this way I ensure that all test cases will have well-formed zone files by construction.

| | Validity Condition | RFC Document |
|---|---|---|
| i. | All records should be unique (there should be no duplicates). | 2181 [EB97] |
| ii. | A zone file should contain exactly one `SOA` record. | 1035 [Moc87b] |
| iii. | The zone domain should be prefix to all the resource records domain name. | 1034 [Moc87a] |
| iv. | If there is a `CNAME` type then no other type can exist and only one `CNAME` can exist for a domain name. | 1034 [Moc87a] |
| v. | There can be only one `DNAME` record for a domain name. | 6672 [RW12a] |
| vi. | A domain name cannot have both `DNAME` and `NS` records unless there is an `SOA` record as well. | 6672 [RW12a] |
| vii. | No `DNAME` record domain name can be a prefix of another record's domain name. | 6672 [RW12a] |
| viii. | No `NS` record can have a non-`SOA` domain name that is a prefix of another `NS` record. | 1034 [Moc87a] |
| ix. | Glue records must exist for all `NS` records in a zone. | 1035 [Moc87b] |

**Table 4.1:** Summary of DNS zone file validity conditions specified in various RFCs.

### 4.2.4 Data Representation

In our Zen model, I represent zone files as a list of resource records, where each resource record contains a domain name, record type, and data fields. I represent user queries similarly as consisting of a domain name and a query type. Record and query types are represented using enums, which Zen translates to integer values.

One challenging decision I ran into was how best to represent and model domain names, for both zone records and record data, in a manner that permits fully automatic and scalable analysis. For instance, a natural way to encode domain names would be as string values (a domain name is just a '`.`' separated string). Indeed, modern SMT solvers like Z3 [DB08] support the logical theory of strings, so this is a natural approach to consider. However, the theory of strings is in general undecidable [CCH17, GMS12]. Moreover, this encoding would require us to define complex predicates for manipulating domain names, including extracting

each of the labels of a domain name and checking whether one domain name is a prefix of another.

Therefore, rather than model domain names as strings, I take advantage of the observation that the particular character values in a domain name label string do not matter for DNS lookup. Instead, all that matters is whether two labels are equivalent to one another and whether a label represents a wildcard. As such, I encode a domain name in Zen as a list of integers and use a specific integer value to represent the wildcard character '$*$'. This allows us to use simple, efficient integer operations and constraints to manipulate domain names according to the formal model.

### 4.2.5 Handling Unbounded Data

A final challenge associated with symbolic execution for my formal model is the fact that there are several sources of *unboundedness*. For example, a zone file can contain an unbounded number of records, and a domain name can contain an unbounded number of labels. Our Zen model contains an unbounded number of paths, since the number of resource records in a zone file is unbounded and the function to select the best records must examine all of them and compare them to one another. SMT constraint solvers have limited support for unbounded data structures such as lists, and in general, reasoning about such constraints requires quantifiers, which lead to undecidability [RBC16]. Therefore, in our Zen implementation I only consider inputs that have a bounded size, *e.g.*, at most $N$ records in a zone file, and hence only produce test cases that respect these bounds. The size of inputs is a parameter that is configurable by the user. While the SCALE approach can therefore fail to detect some errors, I provide experimental evidence of the existence of a *small-scope* property [Jac02], meaning that many interesting behaviors, and behavioral errors, can be exercised with small tests (Section 4.5.1).

## 4.3 Generating Tests for Invalid Zone Files

While it's critical to be able to generate well-formed zone files for testing, bugs can also lurk in implementations' handling of ill-formed zones. Many DNS implementations use zone-file preprocessors to perform syntactic and semantic checks. For example, BIND uses `named-checkzone` [Con02], KNOT uses `kzonecheck` [Lab22], and POWERDNS uses `pdnsutil` [BV22]. The implementations either reject an ill-formed zone or accept it but convert it to a valid one by ignoring certain records that cause it to be semantically ill-formed.

Many security vulnerabilities for software lie in the incorrect handling of unexpected inputs (*e.g.*, in parsers [Zal13]), and DNS software should be no different. Since my executable model includes a formulation of the validity conditions for zone files, I leverage Zen to systematically generate zone files that violate one of these conditions. For example, I ask Zen to generate a zone file in which all but the $7^{\text{th}}$ condition in Table 4.1 is violated and the rest are satisfied.

If an invalid zone is rejected, then there is no issue, but if it is accepted, then there can be errors in how the zone is used for DNS lookups. To test for such errors we must also be able to generate queries for these zones. However, the formal model is only well defined for valid zone files so we cannot use it to generate queries. Instead, I use my technique on zone-file verification presented in Chapter 5 to partition queries into equivalence classes (ECs) relative to a given zone file. An equivalence class is a set of queries with the same resolution behavior, assuming a correct underlying DNS implementation, and the ECs are generated through a simple syntactic pass over a zone file. FERRET generates these ECs and then uses one representative query from each EC as a test. Though the number of ECs can vary widely, depending on the records in a zone file, in practice a zone containing four records will typically induce tens of ECs.

## 4.4 System Overview

FERRET is divided into several components, which are depicted in Figure 4.2. First it uses my Zen model described above to generate test inputs. Because domain names are encoded in Zen using lists of integer labels (see Section 4.2.4), FERRET includes a shim layer that translates the generated zone files and queries into meaningful domain names by mapping these labels to a collection of predefined strings (*e.g.*, `com`). FERRET uses the equivalence-class (EC) generation algorithm of GROOT (Chapter 5) to generate test queries for invalid zone files.

FERRET uses Docker [Mer14] to construct a working container image of each implementation. I cloned the implementations' code as of October 1st, 2020 [Con86, Lab02a, CZ11, HC02, Gc16, EUR12, FC15, Tre01], from their open-source repositories on GitHub [PWH08] and GitLab [SZ14]. FERRET starts a container for each image, and each container serves one zone file at a time as an authoritative zone. The containers expose the DNS port 53 to the host system by mapping an unused host port and enable the host to send queries and receive responses from the DNS server running in that container. Thus each implementation gets a unique host port that maps to the container's DNS port 53. FERRET uses a Python library `dnspython` [HC05] to construct queries and send them to each implementation's container. The library sends the query to each mapped port and then collects the responses. The Docker daemon running on the host machine handles the packet forwarding and mapping seamlessly. For each test case, the Python script prepares the container by stopping the running DNS nameserver, copying the new zone file and the necessary implementation-dependent configuration files to the container, and then restarting the DNS nameserver.

Using containers allows us to easily isolate each implementation. My initial implementation of FERRET spun up new containers for each test case, but this did not scale well. Instead, FERRET reuses the existing implementation container by restarting the nameserver after each test case; it only restarts the container when a test case causes a server to experience

an internal error (*e.g.*, server crash or unresponsiveness). FERRET is open-source and my container approach makes it easy and convenient to integrate new nameserver implementations into FERRET. Other DNS developers can generate an image of their implementation and can quickly compare it with existing ones.

Finally, FERRET performs response grouping followed by fingerprinting to deduplicate errors that are likely to have the same root cause. For each test case, two DNS responses are considered equivalent, and hence in the same group, if they have the same response flags, return code, answer, and additional sections. FERRET only compares the authority section in two responses when their answer sections are empty. I do this because implementations are free to add additional records like a zone's `SOA` or `NS` records along with the requested records. I then fingerprint tests that result in more than one group and thereby represent a likely error.

The fingerprint for a valid test is a tuple consisting of **(a)** the case in the formal model (the leaf label in the decision tree from Figure 4.4), and **(b)** the response groupings. An example fingerprint is $\langle$`R1`, $\{\{$NSD, KNOT, POWERDNS, YADIFA$\}$, {BIND, COREDNS}, {TRUSTDNS, MARADNS}$\}\rangle$. The fingerprint for an ill-formed test is similar but I use the validity condition being violated instead of the model case.

## 4.5 Results

### 4.5.1 Testing Using Valid Zone Files

Using FERRET, I generated thousands of tests and used them to compare the behavior of 8 popular open-source authoritative implementations of DNS. Table 4.2 shows the 8 implementations, the languages they are implemented in, and a brief description of their focus or how they are used. I constrained FERRET to generate tests where the length of each domain name and the number of records in the zone was at most 4. I ran FERRET on a 3.6GHz 72 core machine with 200 GB of RAM and it generated a total of 12,673 valid

| Implementation | Language | Description |
|---|---|---|
| BIND [Con86] | C | *de facto* standard |
| POWERDNS [HC02] | C++ | popular in N. Europe |
| NSD [Lab02a] | C | hosts several TLDs |
| KNOT [CZ11] | C | hosts several TLDs |
| COREDNS [Gc16] | Go | used in Kubernetes |
| YADIFA [EUR12] | C | created by EURid (.eu) |
| TRUSTDNS [FC15] | Rust | security, safety focused |
| MARADNS [Tre01] | C | lightweight server |

**Table 4.2:** The eight open-source DNS nameserver implementations tested by FERRET. FERRET can test implementations implemented in any language.

test cases, one per path in our Zen model that is consistent with the length constraints, in approximately 6 hours. Users can run the tests in parallel, so the runtime depends heavily on the user resources for parallelization. Each test takes around 10 seconds to run on average, and most of the time is spent setting up the zone file and necessary configuration files.

As described in Section 4.4, FERRET runs each test against all 8 implementations and groups their responses. Out of 12,673 tests, FERRET found more than one group in the majority (8,240) of tests. Table 4.3 shows the number of tests generated for each case in the model (Figure 4.4), the number of tests where there was more than one group, and the number of unique fingerprints formed for each model case.

In total the 8,240 tests with more than one group were partitioned into 76 unique fingerprints, for a reduction of more than two orders of magnitude. For 24 of these fingerprints there exists only a single test case, while one fingerprint has 1892 corresponding tests. These 76 fingerprints can over-count the number of bugs since a single implementation issue can cause errors on multiple model paths. For example, YADIFA, TRUSTDNS, and MARADNS do not support `DNAME` records; so any generated test containing this feature will cause them to give the wrong answer or fail to respond. However, two tests can also have the same

| Model Case | #Tests | #Tests Failing | #Fingerprints |
|---|---|---|---|
| E1 | 3180 | 239 | 7 |
| E2 | 12 | 10 | 5 |
| E3 | 96 | 12 | 3 |
| E4 | 6036 | 5312 | 11 |
| W1 | 60 | 33 | 8 |
| W2 | 24 | 21 | 9 |
| W3 | 18 | 16 | 1 |
| D1 | 230 | 65 | 4 |
| R1 | 2980 | 2529 | 27 |
| R2 | 37 | 3 | 1 |

**Table 4.3:** Test generation statistics for $n = 4$. The model case refers to the leaves in Figure 4.4. Even though the number of failed tests is higher, the number of fingerprints is small.

fingerprint despite different implementation root causes; so the number of fingerprints can also under-count the number of bugs.

For these reasons, I manually examined the test cases matching each fingerprint, examining them all when the fingerprint has 4 or fewer tests and otherwise examining a small random sample. By doing this I identified 24 unique bugs, as summarized in Table 4.5 (all except the ones marked with ✧). All of these have been confirmed as actual bugs (no false positives) and developers have fixed 15 of them at the time of writing.

### 4.5.2 Testing Using Invalid Zone Files

FERRET generated 900 ill-formed zone files, 100 violating each of the validity conditions in Table 4.1, in 2.5 hours. I used these zone files to test the four most widely used DNS implementations — BIND, NSD, KNOT, POWERDNS— as these have a mature zone-file preprocessor available.

| BIND | NSD | KNOT | PDNS | #Zones | Condition violated | #Zones with a difference |
|---|---|---|---|---|---|---|
| A | A | A | R | 100 + 100 + 1 | i or viii or ix | 11 + 94 + 1 |
| A | A | R | R | 100 + 61 | vi or ix | 8 + 3 |
| A | R | A | R | 17 + 100 | ii or iii | 1 + 6 |
| A | R | R | A | 60 | vii | 53 |
| R | A | R | A | 34 | ix | 7 |
| A | R | R | R | 39 | vii | - |
| R | A | R | R | 4 | ix | - |
| R | R | R | A | 95 + 1 | v or vii | - |
| R | R | R | R | 83 + 100 + 5 | ii or iv or v | - |

**Table 4.4:** Invalid zone file statistics. The second row shows that 100 (61) zone files that violate condition vi (ix) are accepted by only BIND and NSD, and 8 (3) of them resulted in some difference between the two implementations.

There is no practical limit on the number of invalid zone files the ferret can generate. I limited it to 100 for each violation in our experiments, but one could use FERRET to generate many more such tests if desired. Similarly, though I only explored violations of single well-formedness rules, it is straightforward to use FERRET to generate tests that violate a combination of rules. As a first step, FERRET checked all of the zone files with each implementation's preprocessor: `named-checkzone` [Con02] for BIND, `kzonecheck` [Lab22] for KNOT, `nsd-checkzone` [Lab02b] for NSD, and `pdnsutil` [BV22] for POWERDNS. Each implementation can either reject or accept the invalid zone file and Table 4.4 shows the statistics of how different implementations treat the zone files.

All together there are 573 invalid zone files (the first five rows in the table) that are accepted by more than one DNS implementation and so are amenable to differential testing. Our formal model relies on zones to be well-formed: so we cannot use it to generate queries for these zones. Instead I leverage GROOT (Chapter 5), which generates query equivalence classes (ECs) of the form $\langle \texttt{example.com}, t \rangle$ for a given zone file, one for each DNS record type

| Impl. | Bugs Found | Bug Type | Status |
|---|---|---|---|
| BIND | Sibling glue records not returned [KAK21b] | Wrong Additional | ✓ |
| | Zone origin glue records not returned [KA21] | Wrong Additional | ✓ |
| | `DNAME` recursion denial-of-service✧ [KAK21a] | Server Crash | ☑ |
| | Wrong `RCODE` for synthesized record✧ [KAK20] | Wrong `RCODE` | ☑ |
| NSD | `DNAME` not applied recursively [KW21b] | Wrong Answer | ☑ |
| | Wrong `RCODE` when $*$ is in Rdata [KW21a] | Wrong `RCODE` | ☑ |
| | Used `NS` records below delegation✧ [KW21c] | Wrong Answer | ☑ |
| | Wrong `RCODE` for synthesized record✧ [KW20] | Wrong `RCODE` | ☑ |
| POWERDNS | `CNAME` followed when not required [KD20a] | Wrong Answer | ✓ |
| | `pdnsutil` check-zone `DNAME`-at-apex✧ [KD20b] | Preprocessor Bug | ☑ |
| KNOT | Incorrect record synthesis [KPS21a] | Wrong Answer | ☑ |
| | `DNAME` not applied recursively [KPS21c] | Wrong Answer | ☑ |
| | Used records below delegation [KPS21b] | Wrong Answer | ☑ |
| | Error in `DNAME-DNAME` loop KNOT test [KPS20] | Faulty KNOT Test | ☑ |
| | Wrong `RCODE` for synthesized record✧ [PS20] | Wrong `RCODE` | ☑ |
| COREDNS | `NXDOMAIN` for existing domain [KG21b] | Wrong `RCODE` | ☑ |
| | Wrong `RCODE` for `CNAME` target [KO20] | Wrong `RCODE` | ☑ |
| | Wildcard `CNAME` loops & `DNAME` loops [KG21a] | Server Crash | ☑ |
| | Wrong `RCODE` for synthesized record [KOY20] | Wrong `RCODE` | ☑ |
| | `CNAME` followed when not required [KOY21] | Wrong Answer | ☑ |
| | Sibling glue records not returned [KG21c] | Wrong Additional | ✓ |
| YADIFA | `CNAME` chains not followed [Kye20b] | Wrong Answer | ☑ |
| | Wrong `RCODE` for `CNAME` target [Kye20a] | Wrong `RCODE` | ☑ |
| | Used records below delegation [Ky21] | Wrong Answer | ☑ |
| MARADNS† | AA flag set for zone cut `NS` RRs | Wrong Answer | ✓ |
| | Used records below delegation | Wrong Answer | ✓ |
| TRUSTDNS† | Wildcard match only one label [KF21] | Wrong Answer | ✓ |
| | Used records below delegation [KFB20] | Wrong Answer | ✓ |
| | AA flag set for zone cut `NS` RRs [KF20b] | Wrong Flag | ✓ |
| | `CNAME` loops crash the server [KF20a] | Server Crash | ☑ |

**Table 4.5:** Summary of the bugs found by FERRET across the eight implementations. Status column represents whether the developers responded and acknowledged (✓) and also fixed (☑) to the filed bug report. The † symbol denotes implementations with unreported issues due to missing or unimplemented features. The ✧ symbol denotes the bugs found exclusively using testing with invalid zone files. We reported all the bugs FERRET identified to the respective developers before publishing this paper.

$t$, and does not require the zone to be semantically well-formed. I used 7 query types: `A`, `NS`, `CNAME`, `DNAME`, `SOA`, `TXT`, `AAAA`. I excluded 19 zone files as GRoot generated over 200 ECs for each of them due to multiple interacting `DNAME` loops (explained in Section 6.3). For the remaining 554 zone files, the average number of ECs is 21*7 *i.e.*, 21 domains names and each domain name is paired with the 7 types, and I chose one representative query from each EC.

The last column in Table 4.4 shows the results of differential testing. For example, 106 out of the 201 zone files in the first row exhibited differences among the three implementations during testing. I manually inspected all differences for the zone files that violated conditions of i, ii, iii, vi, and ix, as there were 12 or fewer such differences in each category, and we inspected a random sample for the others. By doing this I identified 6 new errors as shown in Table 4.5 with the ✧ symbol and all of them are fixed. Some of the errors identified earlier were also present here but are not double-counted.

### 4.5.3 Small-scope Property Validation

Finally, I performed an experiment to validate the small-scope property that justifies our approach — many interesting behaviors can be covered with small tests. I used Ferret to generate valid tests where the length of each domain name and the number of records in the zone were limited to $n$, for different values of $n$. Table 4.6 shows the results. For example, when $n = 2$ there are 52 feasible paths through the model. Ferret generated the corresponding 52 tests in 10 minutes, out of which 12 had more than one group, and these 12 fell into 9 fingerprints. By inspecting those failed tests, I identified 4 unique bugs, which are a subset of the ones identified by our evaluation described in Section 4.5.1, where $n = 4$.

My experiment identifies two distinct forms of small-scope property. First, the DNS query resolution protocol itself, as represented by our logical model, has a small-scope property. In particular, when $n = 2$ all leaf nodes in Figure 4.4 are covered by at least one test, except for the R1 leaf, and all leaf nodes are covered when $n$ is 3 or higher. Hence, although we are

| **Max Length ($n$)** | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| **No. of Tests** | 52 | 618 | 12673 | 646K (51K tested) |
| **Test generation time** | 10m | 40m | 6h | 14d |
| **No. of Tests Failing** | 12 | 224 | 8240 | 41173 |
| **No. of Fingerprints** | 9 | 22 | 76 | 115 |
| **No. of Bugs** | 4 | 14 | 24 | 27 |

**Table 4.6:** Results summary for different length bounds.

restricted to generating small zones, we can still cover all return points in our formal model, each of which represents a distinct RFC behavior.

Second, the DNS nameserver implementations have a small-scope property. In part the fact that I have identified dozens of subtle new errors is evidence that small tests can explore interesting behaviors. The results in Table 4.6 add further evidence. As I increase the size of $n$ from 2 to 3 to 4, the number of bugs identified goes from 4 to 14 to 24. In the $n = 5$ case, FERRET generated over 646K tests and took almost 14 days to finish. The distribution of tests across model cases is similar to the $n = 4$ breakdown shown in Table 4.3, where the majority of tests fall into the E1, E4 and R1 cases. I randomly sampled 50K tests to run from these three cases, according to their proportions. The other cases totalled to around 1000 tests, so we ran all of them. Out of the resulting 115 fingerprints, 50 fingerprints were in common with the fingerprints of $n = 4$. I therefore decided to examine the remaining 65 fingerprints to search for new bugs. For these 65 fingerprints, the median number of tests in each fingerprint was 3, and the mode was 1. I found three bugs that we did not find with $n = 4$, but all three bugs were covered by the tests for invalid zones with $n = 4$ (Section 4.5.2). In other words, increasing $n$ from 4 to 5 has so far not uncovered any new errors in the DNS nameserver implementations.

## 4.6 Example Bugs

I now provide a detailed description of some of the bugs from Table 4.5. Two of them were already described in Section 4.1.

### 4.6.1 Bug #3: CoreDns crash

Ferret generated the following test that causes CoreDns, the recommended nameserver for Kubernetes, to crash. It was subsequently confirmed and fixed by the CoreDns developers.

```
  example.   SOA     ···
*.example.   CNAME   foo.example.
```

**Query:** ⟨baz.bar.example., CNAME⟩

In this example the zone file has a wildcard `CNAME` record that rewrites any query ending with the label `example` to `foo.example`. This rewritten query will then match the wildcard record again and so on, causing CoreDns to loop and consume resources until, eventually, the server crashes with the following message:

```
"runtime: goroutine stack exceeds 1000000000-byte limit"
"runtime: sp=0xc03c6c0378 stack=[0xc03c6c0000, ...]"
"fatal error: stack overflow"
```

Interestingly, CoreDns correctly guards against `CNAME` loops that do not involve wildcard; so only a test that combines `CNAME` and wildcards will trigger the bug. After our bug report, the developers fixed the issue by adding a loop counter and breaking the loop if the depth exceeds nine. They commented: "Note the answer we're returning will be incomplete (more cnames to be followed) or illegal (wildcard cname with multiple identical records). For now it's more important to protect ourselves than to give the client a valid answer."

Crashes like this represent serious security vulnerabilities, particularly in multi-tenant settings such as the attack described earlier in Figure 4.3(a).

### 4.6.2 Bug #4: CoreDns apex only zone error

Ferret generated the following test that caused CoreDns to return a response that misinforms the resolver that a domain does not exist when it does exist, in fact. Such responses can lead to an outage similar to the recent one faced by Microsoft Azure [Tun19], where the nameserver replied with the non-existence of domains, thereby disconnecting users from services globally.

```
cs.clg.  SOA  ...
cs.clg.  NS   ns1.net.
```

**Query:** ⟨cs.clg., A⟩

The above zone file only has records at what is called the zone apex. For this test, the response should have an empty answer section but with a `NOERROR RCODE`. CoreDns returned an empty answer but with an `NXDOMAIN`. The `RCODE` is important as resolvers can use QNAME minimization as described in RFC 7816 [Bor16] when resolving names like `email.cs.clg` and will wrongly conclude that there is nothing at `cs.clg` as well as below it.

When we filed the bug report with CoreDns, they acknowledged the problem and also determined that this only happens when the zone does not contain any other data, *i.e.*, it is an apex-only zone. The root cause of the issue was explained as, "... This stems from the optimization of putting the APEX record (SOA+NS) not in the tree (less and less convinced that was a good idea)."

### 4.6.3 Bug #5: Wrong `RCODE` for synthesized `CNAME`

Ferret generated a zone that violates condition vii in Table 4.1:

```
       test.com.   SOA     ...
   foo.test.com.   DNAME   bar.test.com.
cs.foo.test.com.   AAAA    1:db8::2:1
```

**Query:** ⟨www.foo.test.com., CNAME⟩

BIND and POWERDNS accepted the zone file but NSD and KNOT did not. FERRET chose the above query as the representative from the query EC ⟨$\alpha$.foo.test.com., CNAME⟩ generated by GROOT, where $\alpha$ represents any sequence of labels that does not start with cs. BIND responded with:

```
"rcode NXDOMAIN",
";ANSWER",
"foo.test.com. 500 IN DNAME bar.test.com.",
"www.foo.test.com. 500 IN CNAME www.bar.test.com.",
```

The response from POWERDNS was the same but with a NOERROR RCODE. The RCODE is important as resolvers can use QNAME minimization (RFC 7816 [Bor16]) to wrongly conclude domain (non-)existence if an incorrect RCODE is returned. However, since the RFCs do not describe this subtle case, the intended behavior is unclear. Since the query is not relevant to the AAAA record, which violates the validity condition, to further investigate this issue we decided to remove that record and check the responses from NSD and KNOT. Both responded with the same response as BIND, leading us to (wrongly) conclude that the issue was with POWERDNS.

To our surprise, after reporting the issue to POWERDNS they responded: "The PowerDNS behavior looks correct to me. Are you sure BIND, NSD, and knot all return NXDOMAIN on a CNAME query in this context?" BIND and KNOT noticed the issue we filed on POWERDNS's GitHub and fixed the bug almost immediately, even before we filed reports on their repositories. After some back and forth with the NSD developers they concurred saying: "If you are right

that the other implementations do this, then we can do that too; that makes less unexpected surprises in packet responses."

### 4.6.4 Bug #6: PowerDns `pdnsutil` bug

Ferret generated the following test case and PowerDns returned an incorrect response, exposing a bug in its zone-file preprocessor.

```
      dept.com.   SOA     ...
      dept.com.   DNAME   dept.edu.
host.dept.com.    A       1.1.1.1
```

**Query:** ⟨host.dept.com., A⟩

The zone file is considered invalid as it violates condition vii in Table 4.1. `nsd-checkzone` and `kzonecheck` preprocessors reject the zone file but `named-checkzone` and `pdnsutil` do not raise any errors or warnings and accept the zone file. When queried for the `A` record, PowerDns returned this record even though it should have used the `DNAME` record. PowerDns has a long-standing open issue about handling `DNAME` occlusion (records below a `DNAME`, which should be ignored), and `pdnsutil` generally gives a warning but did not in this specific case. We filed a bug report for this test and the developers confirmed a bug in `pdnsutil` when the `DNAME` is at the apex of the zone. This is now fixed and `pdnsutil` gives a warning as in other occlusion cases.

### 4.6.5 Bug #7: `DNAME` loops

As another example, Ferret generated a test case that creates a loop using `DNAME` records such that, as in the wildcard loop example in Section 4.6.1, certain input queries will loop indefinitely.

| | | |
|---|---|---|
| corp. | SOA | ... |
| corp. | NS | ns1.com. |
| corp. | DNAME | us.corp. |

**Query:** ⟨www.corp., NS⟩

In the test, the DNAME record is applied to the query once, which results in a synthesized CNAME record (for caching) that is specialized for the user query:

```
"www.corp. 500 IN CNAME www.us.corp."
```

The nameserver then proceeds to lookup the response for the new, rewritten query that has the domain name www.us.corp. This process can lead to infinite recursion.

In this case, implementations are free to choose how they will respond. It is important that implementations guard against these cases, which can otherwise lead to heavy resource consumption by the nameserver. BIND applies the DNAME multiple times and stops when the limit reaches 17. COREDNS did not respond to this test, and the server crashed as with the wildcard loop bug shown earlier. The developers have since patched this bug. POWERDNS returns an error message through the SERVFAIL RCODE, while KNOT and NSD apply the DNAME only once and return the response. However, we found through another test that the KNOT and NSD behavior was also incorrect, because forcing the DNAME to only be applied once prevents legitimate cases where the same DNAME record must be applied multiple times (Section 4.6.7).

As a final interesting byproduct, this test also found an error in the KNOT test suite that compares KNOT's responses with BIND. A comment indicated that a test targeted a DNAME-DNAME loop scenario as in our test. On inspection, however, we realized they did not construct the zone file correctly, and so the test case did not create a loop. When we reported the issue, the developers agreed and fixed it.

### 4.6.6 Bug #8: Yadifa `CNAME` chains not followed

FERRET generated a test case that revealed a bug in YADIFA, which resulted from a side effect of a previous fix their developers made. The bug causes YADIFA to return an incomplete response forcing the resolvers to make multiple unnecessary round-trips to the same nameserver.

```
       dept.com.   SOA     ...
www.cs.dept.com.   CNAME   cs.dept.com.
    cs.dept.com.   CNAME   dept.com.
       dept.com.   A       2.2.2.2
```

**Query:** ⟨www.cs.dept.com., A⟩

For the test query, the expected response is to rewrite the query twice using both the `CNAME` records and finally return the IP address. YADIFA rewrote the query only once and did not return the IP address, which was clearly in violation of the RFC rule, "CNAME chains should be followed" [Moc87a].

`CNAME` chains are used extensively in practice as a form of indirection, for instance to optimize traffic in CDNs [SBK20]. In this case, YADIFA simply returned the incomplete answer. They acknowledged the issue and responded: "The rerun of the query was incorrectly disabled, the issue is fixed and will be updated on github on our next update of the code." They also mentioned that it was a side effect of a previous fix they made and added a regression test to keep such things in check.

### 4.6.7 Bug #9: `DNAME` not applied recursively

As an another example of an interesting bug that FERRET was able to find, consider the following test case that it generated:

```
sig.edu.   SOA     ...
sig.edu.   NS      ns1.outside.edu.
sig.edu.   DNAME   edu.
```

**Query:** ⟨sig.sig.sig.edu., NS⟩

This zone file creates a `DNAME` record that rewrites query prefixes repeatedly. For example, on the generated query the `DNAME` record will match and be applied two times. For the above test case, NSD and KNOT incorrectly did not apply the same `DNAME` twice. This test does not apply to YADIFA, MARADNS, and TRUSTDNS as they do not support `DNAME`. The response (answer section) from BIND, POWERDNS, and COREDNS was:

```
";ANSWER",
"sig.edu. 500 IN DNAME edu.",
"sig.sig.sig.edu. 500 IN CNAME sig.sig.edu.",
"sig.sig.edu. 500 IN CNAME sig.edu.",
"sig.edu. 500 IN NS ns1.outside.edu.",
";AUTHORITY",
";ADDITIONAL"
```

where as the response (answer section) from NSD and KNOT was:

```
";ANSWER",
"sig.edu. 500 IN DNAME edu.",
"sig.sig.sig.edu. 500 IN CNAME sig.sig.edu.",
```

If, in reality, a zone similar to this is served by either KNOT or NSD, then the resolver would be making two round-trips instead of a single-round trip it would have if it were others. Increased round-trips by the resolver can affect performance and increase query-response delay to the user.

KNOT agreed that the response has to be similar to BIND's response and fixed it. The initial reaction from NSD was, "It is a feature, not a bug. The reason is that it is seen as a DNAME loop because the same DNAME is applied twice. If it was two different DNAMEs then there would be two DNAMEs in the answer." Later, they responded as, "But fixed it anyway, because the same behavior as BIND is desirable. The fix tests if the CNAME can be added. That results in the NS record returned in the example you gave. Thanks for the report!"

### 4.6.8 Bug #10: NSD wrong `RCODE` when * is in Rdata.

As the final example, FERRET generated a test case that revealed a bug in how NSD sets the `RCODE` when `CNAME` target has * in its domain name and does not exist in the same zone. As said earlier, `RCODE` is important as resolvers use it to determine whether domains exist or not. The test case that triggered this issue was the following:

```
    booksonline.  SOA    ...
buy.booksonline.  CNAME  www.*.booksonline.
```

**Query:** ⟨buy.booksonline., NS⟩

For the test query, NSD returns the `CNAME` record in the response but sets `RCODE` to `NOERROR` where as others set it to `NXDOMAIN`, which is expected as `www.*.booksonline` does not exist. The interesting aspect here was, if there was no * in the domain name then NSD sets the `RCODE` properly.

When reported, the developers acknowledged and fixed it. Their response was, "It has to do with the internal data structure for storing domains in the memory of NSD, there a domain struct is created for the right hand of the CNAME, and it is set to be non-existing. The is_existing was not checked for the wildcard expansion, and this is fixed by the commit. So this fix is only for CNAMEs to a wildcard right hand, where that wildcard right hand does not exist in the zone. Thanks for the report!"

## 4.7 Discussion

The SCALE approach worked surprisingly well at identifying subtle errors in implementations. This was not obvious from the beginning, since each implementation can have very different control logic compared to one another and compared to our formal model. And yet seemingly the tests derived from paths through my formal RFC model frequently uncover bugs in rare control paths for these implementations.

On the other hand, this approach is not a panacea. I found situations where one path in the model corresponds to multiple paths in an implementation due to the internal data structures that it uses to represent different record types, which can lead to FERRET missing some issues. This showed up, for example, with empty non-terminals (ENTs) — domain names that own no resource records but have subdomains that do. Since there is no explicit branch that differentiates empty non-terminals in the model, FERRET did not generate test cases where the zone file had both an ENT and a query targeting that ENT. However, by manually testing a few such cases, I found two more bugs in COREDNS. In one case, COREDNS returns an incorrect response when there is an empty wildcard non-terminal, and in the other case, it applies wildcard to the cases it should not as they match the ENT.

Going forward it may be possible to extend FERRET to find more cases like this. One way to do so would be to manually add additional non-semantic branches to the model to expose behavior thought to be error-prone. Yet another approach would be to generate more than one input per path in the model by attempting to vary the query types.

## 4.8 Related Work

FERRET and SCALE are related to several lines of prior work in DNS and in automated testing.

**Verified DNS implementations.** One approach is to build, from scratch, a nameserver implementation verified to be correct. This approach has found some success in other domains, for example, in operating system microkernels [KEH09] using proof assistants such as Coq [Let04]. IRONSIDES [CF12] is an implementation of a DNS resolver and authoritative nameserver that uses SPARK [Bar12] to prove the absence of dataflow errors such as buffer overflows. While this work is promising, it does not formalize the DNS RFC semantics and thus cannot provide any functional correctness guarantees. Moreover, open source implementations such as BIND [Con86] are already used pervasively in the Internet. Providing a new verified implementation does not help these existing deployments.

**Models for DNS.** In my first work on the DNS [KBA20a] I presented the first formalization of DNS semantics. However, it was a paper formalism and was only used to prove the correctness of the equivalence-class generation algorithm that forms the core of GROOT's approach to verifying zone files. Indeed, GROOT assumes that DNS implementations conform to the DNS RFCs. This work is therefore complementary, but I used GROOT's logical model as a basis for my executable Zen model. I also leveraged GROOT's equivalence-class generation algorithm to create queries for invalid zone files.

**Fuzz testing.** Fuzz testing is the technique in which a program is bombarded with many randomly generated inputs. As mentioned in the beginning of the chapter, fuzzing cannot easily be used in our setting due to the need to navigate complex constraints and dependencies, and hence existing fuzzers for DNS [Pat22, Cam19, St10] are limited to testing DNS parsers and use a fixed zone file. Fuzz testing with semi-random and/or grammar-based or mutation tests has seen success in recent years for certain code bases [Hoc07, Edd04, LSF15, BPR16, Zal13, God20, LS18, GCB16]. Mutation testing [LS18, GCB16] mutates well-formed inputs randomly or based on coverage feedback. The success rate of mutating well-formed seed zone files to get new zone files would be close to zero as mutation will likely make them not well-formed. Users must also instrument the implementation to use coverage feedback, which will be on an implementation basis and not based on RFCs. While grammar-based

approaches [Edd04, AS19] would ensure that generated inputs are syntactically well-formed, they still wouldn't address the issues of ensuring semantic well-formedness or relating the query to the zone files.

**Symbolic execution.** Symbolic execution [GKL08, GKS05], which systematically solves for inputs that take different execution paths in a program, has also been successful [CDE08, CJV11]. However, as described in the beginning, due to the scale and complexity of DNS nameserver implementations, symbolic execution has been used only on individual functions and has avoided the need to generate zone files [RE15]. My SCALE approach uses symbolic execution to drive test generation, but it does so on an executable model of the RFC behavior, which is significantly smaller and simpler than an implementation and has carefully chosen data representations that are amenable to symbolic execution. As a result, symbolic execution on my model is tractable and allows me to jointly generate (small) zone files and DNS queries that exercise interesting behaviors.

**Model- and specification-based testing.** In model-based testing (MBT) [BMM18, NR95, VCG08, PA09] a user builds an abstract model of the system to test (*e.g.*, a finite state machine [BMM18, VCG08]). A tester implementation then generates paths through this abstract model and creates concrete tests by "filling in" missing information from the abstract example. For static network dataplane verification, [SPN16] developed a network modeling language to model different network boxes. Closest to my work are model-based testers for black-box network functions (*e.g.*, [FYT16, SWD20]), which also use symbolic execution to generate tests. However, they respectively use finite-state machine models [FYT16] and a domain-specific language for specifying network function behavior [SWD20], while I have implemented a full functional model of DNS in a general modeling language [BM20]. Further, their setting does not require generating configurations, which is the key technical challenge for testing protocols like DNS. Specification-based testing leverages a user-provided specification of the valid inputs to a function. Most commonly, tests are generated by finding inputs that satisfy a given precondition [BKM02]. Like SCALE, these approaches typically rely on a

small-scope hypothesis [Jac02] and hence focus on generating small inputs. Recent work has developed an approach to automated testing for QUIC implementations [MZ19b, MZ19a] that leverages a formal specification, but in a very different way than in our approach. Specifically, the specification models the party that is interacting with the implementation being tested and is used to generate valid responses.

## 4.9 Comments from DNS Community

I presented SCALE and FERRET at a DNS-specific workshop called DNS OARC (DNS Operations, Analysis, and Research Center), attended by DNS operators, implementors (for example, BIND, KNOT, POWERDNS, and Amazon Route 53 DNS developers), and researchers. My work was well received by the community. The following are some of the comments I received after the presentation:

- The DNS-OARC workshop tweeted the following from their official handle — "Incredible reception from the audience on @SivaKesavaRK presentation. The automation tool received great compliments from the DNS experts."
- Peter Van Dijk, a senior POWERDNS developer commented — "This is awesome, thank you for this work, and thank you for your very clear bug reports, both to us (PowerDNS) and to other projects. I was not kidding about the excellent bug reports, by the way - (link to a bug report)."
- Vicky Risk, director of marketing at Internet Systems Consortium, said — "I was skeptical because I thought – why should I believe his tests, but he proved them by running against so many DNS servers through them. So, possibly new RFCs should come with their own logic diagram which can be used to generate the tests"

- Paul Hoffman, a principal technologist at ICANN and author or co-author of many DNS-related RFCs from the IETF, agreed with Vicky Risk, saying, "I was skeptical at the beginning but blown away by the end."

- There were similar comments on the work from Casey Deccio, assistant professor at Brigham Young university, Joe Abley, chief technology officer at public interest registry, Gavin McCullagh, principal systems development engineer in Amazon Route 53, and others.

## 4.10 Summary

In this chapter, I introduced FERRET, the first automatic test generator for RFC compliance of DNS nameserver implementations. The SCALE approach underlying FERRET uses symbolic execution of a formal model to jointly generate configurations together with inputs. FERRET combines this technique with differential testing and fingerprinting to identify and automatically triage implementation errors. In total FERRET identified 30 new bugs, including at least two for each of the 8 implementations that we tested.

I believe my SCALE approach to RFC compliance testing and "ferreting" out bugs through (i) symbolic execution of a small formal model to jointly generate configurations together with inputs, combined with (ii) differential testing, and (iii) fingerprinting, could be useful more broadly beyond the DNS. For instance, there are many other complex and distributed protocols used at different network layers such as routing protocols like BGP and OSPF, flow control protocols like PFC, new transport layer protocols such as QUIC, and many more. I will describe some challenges and preliminary ideas on how to apply the SCALE methodology beyond DNS in the future work section.

# CHAPTER 5

# DNS Configuration Verification

As we have briefly seen in the first two chapters, configuring DNS zone files of organizations is challenging for several reasons - complex record types, nondeterminism, scale of the system and distributed management. In this chapter, I first provide a more concrete and an end-to-end example to demonstrate these challenges in Section 5.1. 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 [Fry14] (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 [npm18]. 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 [Tun19] 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 [Kep20, ZBW07], testing [Hos20], linting [Mic18] 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 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, I present GRoot in this chapter, which, to the best of my knowledge, is the first verification tool for DNS configurations. Given the DNS zone files of an organization and a property $\Phi$ of interest, my algorithm and the corresponding tool GRoot will either verify that $\Phi$ holds *for all possible DNS queries* or provide all counterexamples.

GRoot avoids verifying the huge space of DNS queries by first partitioning 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 symbolically executes the set of queries in each equivalence class to efficiently find (or prove the absence of) any bugs such as rewrite loops. GRoot relies on the formal model from Chapter 3 to automatically verify DNS configurations and detect any misconfigurations. I, indeed, use the formal model to prove that my EC generation algorithm satisfies the key correctness property described above.

I applied GRoot to the configuration files 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. I, also, show GRoot can scale to networks with tens of millions of records spread across tens of thousands of zones using a synthetic dataset that I created from over 65 million real DNS records [DNS13].

To summarize, I make the following contributions:

(§ 5.2) I describe 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.

(§ 5.3) I describe a technique for symbolically executing the ECs using the formal model and checking properties efficiently as graph properties.

(§ 5.5) I present formal proofs to show that the EC generation algorithm is sound, complete and efficient.

(§ 5.7) I present an evaluation of GROOT on production configuration files. I evaluate GROOT using data from **(a)** configurations obtained from a large campus network, **(b)** configurations obtained from a large infrastructure service provider, and **(c)** a synthetic dataset built from over 65 million Internet records, showing that GROOT is effective at finding bugs and verifying large configurations.

## 5.1 Motivating Example

I explained how a DNS query is resolved in Section 1.1 in a simple fashion without showing the zone files at the nameservers. In this section, I will explain how different record types and nondeterminism can lead to different answers and how difficult it can get to reason about the possible behaviors with the help of an example. Consider the configuration zone files shown in Figure 5.1, which are based on real records I 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. I depict each record with an accompanying label (*e.g.*, ⓐ) and refer to those labels when discussing a record.

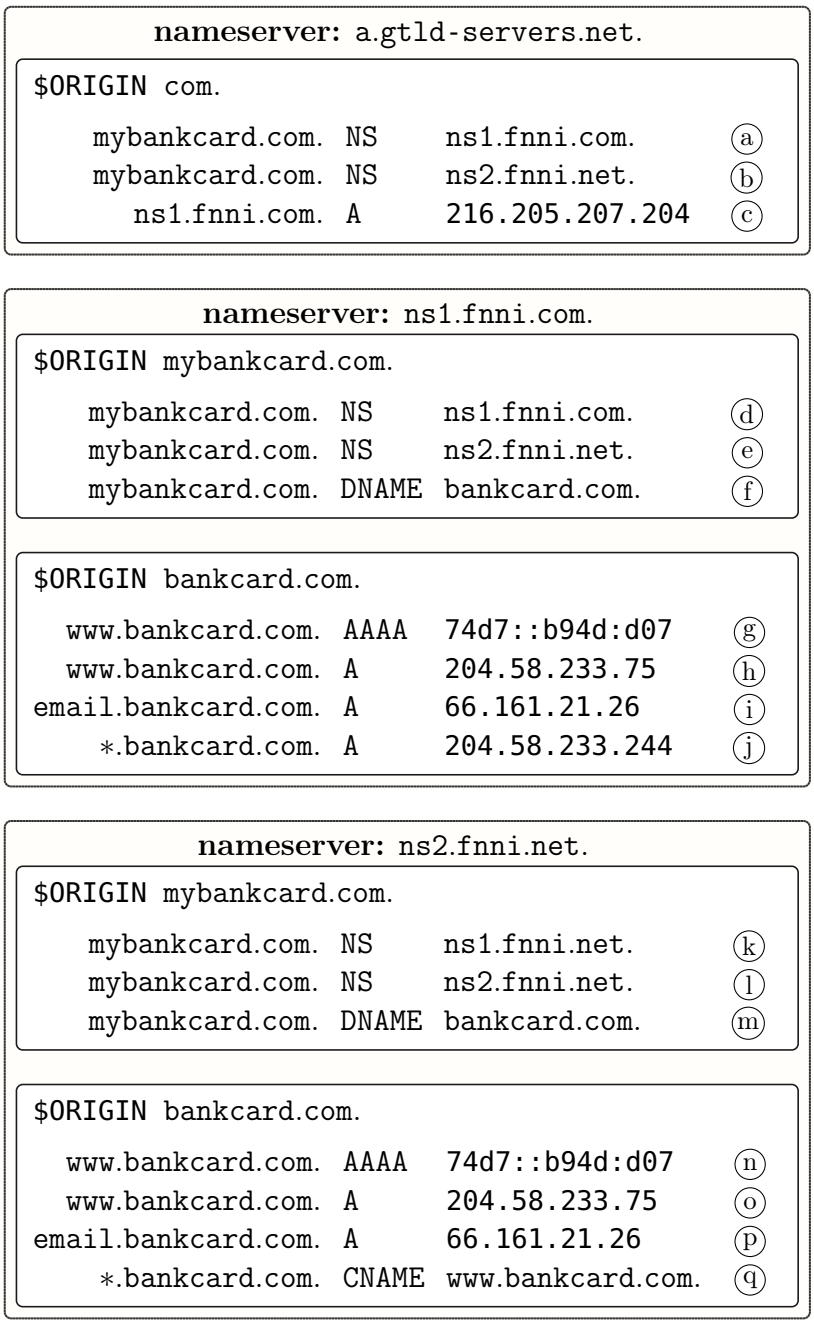



**Figure 5.1:** Example zone files for three nameservers: a.gtld-servers.net, ns1.fnni.com, and ns2.fnni.net. The query $\langle$support.mybankcard.com, A$\rangle$ has two possible executions: one for records $\langle a, f, j\rangle$ and another for $\langle b, m, q, o\rangle$.

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 $\langle$`support.mybankcard.com`, `A`$\rangle$, 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* [HSF19] (`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 `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 (Section 2.4). Thus, the nameserver will return an answer with an IP address `204.58.233.244`.

### 5.1.1 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 [SCK06, WH97, Gle18].

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 [Tun19, Fry14, Inf19, Yor15, New10, Zel13] (Section 1.3). Indeed, there are many ways in which DNS behavior can go

| 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.*, `ucla.edu`) |
| **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 $q_1 \to q_2 \to q_3 \to \ldots \to q_1$ |
| **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 5.1:** Sample subset of possible bugs. Several are taken from previous work [PFM04] while I proposed the rest.

wrong, in addition to nondeterministically returning different answers as shown above. For example, a configuration mistake might result in DNS returning `NXDOMAIN` (non-existent domain) for a popular service, which can result in a loss of connectivity, as was the case in the Azure outage [Tun19]. As another example, a query might get stuck in a rewrite loop. Table 5.1 summarizes several common kinds of DNS misconfigurations. Section 5.7 demonstrates my tool GRoot's effectiveness in finding such errors in real-world DNS configurations.

## 5.2 A Fast Verification Algorithm

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

### 5.2.1 Equivalence Class Generation

The idea with my 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. I define this notion of equivalence more formally and prove that it is correct in Section 5.5. The set of ECs our algorithm computes need not be, and indeed is not, always minimal.

Other verification tools such as Veriflow [KZC12] and Atomic Predicates [YL16, YL17] 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, I leverage the hierarchical, tree-like structure of domain names to reduce this cost. Specifically, I show in Section 5.5 that in the absence of `DNAME` rewrites, our approach computes the set of ECs in linear time.

### 5.2.2 Label graph construction

As a first step to generate query ECs, my 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 5.1: the corresponding label graph is shown in Figure 5.2. The label graph is rooted at $\epsilon$ and every domain name that appears as the key

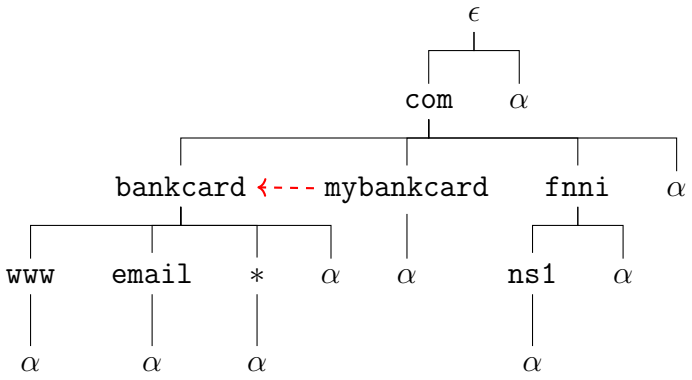



**Figure 5.2:** Label Graph used for equivalence class generation for the zone files from Figure 5.1. Note, only the domain name ($d$) field of the records are used but not the answer ($a$) field. The dotted red edge represents the `DNAME` redirection of (f).

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, I 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, I add an $\alpha$ 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.

### 5.2.3 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 $\alpha$, it constrains it to exclude its siblings. For the example in Figure 5.1, 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, I simply treat wildcards as character labels ('$*$'), as such characters are valid and will experience an exact match with a wildcard record. I instead use the $\alpha$ 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) $\langle \alpha \circ \texttt{bankcard} \circ \texttt{com} \circ \epsilon, t \rangle \quad \alpha[0] \notin \{\texttt{www}, \texttt{email}, *\}$

### 5.2.4 `DNAME` rewrites

The next paths traversed are those for `mybankcard.com`. Since `mybankcard` has a `DNAME` record, I continue enumerating paths through the dashed edge. However, since we want to capture the input query before the transformation, I 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, $\langle \alpha \circ \epsilon, t \rangle$, $\alpha[0] \notin \{\texttt{com}\}$.

### 5.2.5 `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 once the 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.

## 5.3 Symbolic Execution of ECs

To determine the behavior of the equivalence classes, I symbolically execute each EC using semantics from Figures 3.1, 3.2 and 3.3. 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 3.2 and 3.3. The result is a single graph representing multiple interpretation graphs.

Figure 5.3 shows the result of symbolic execution for the running example for three equivalence classes, which are compactly represented as: $\langle \alpha \circ \texttt{mybankcard} \circ \texttt{com} \circ \epsilon, \{\texttt{A}, \texttt{MX}, \texttt{TXT}\} \rangle$.

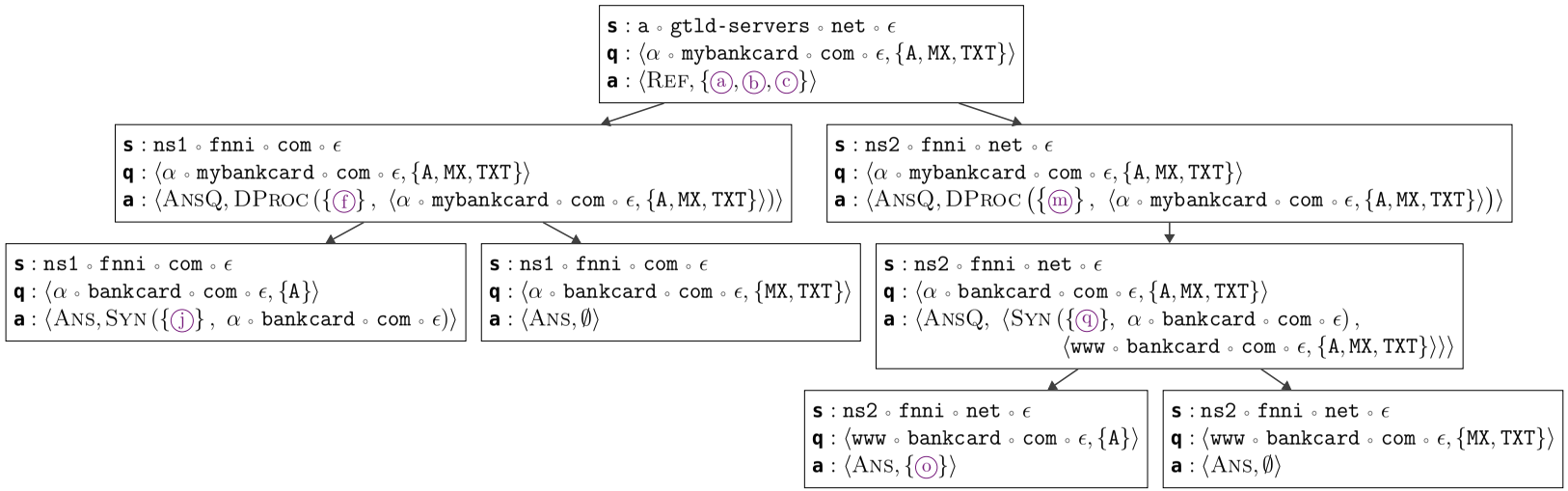



**Figure 5.3:** The interpretation graph based on the zone files shown in Figure 5.1 for types $\{\mathtt{A}, \mathtt{MX}, \mathtt{TXT}\}$ for the equivalence classes given by the schematic query: $\langle \alpha \circ \mathtt{mybankcard} \circ \mathtt{com} \circ \epsilon, \{\mathtt{A}, \mathtt{MX}, \mathtt{TXT}\}\rangle$ with $\alpha[0] \notin \{\mathtt{www}, \mathtt{email}, *\}$.

I 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.

## 5.4 Checking Properties

The representation for ECs and their interpretation graphs facilitates efficient checking for a wide variety of properties. I write property checkers as custom graph algorithms (Section 5.6) 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 5.2 summarizes the implementation of checkers for the bugs listed in Table 5.1. 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.

## 5.5 Proof of Correctness

I prove my approach with GROOT is correct in two steps. First, I show that my equivalence class generation algorithm computes classes of queries that adhere to a restrictive notion of equivalence called *strong equivalence*. Next, I 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 3.1), which generates specialized records for use in the cache. For example, two

| Bug | Description |
|---|---|
| **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 |
| **Rewrite 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 5.2:** Bug finding implementation for Table 5.1.

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

I 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 my approach is correct, is to show that the algorithm presented in Section 5.2.1 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 proof is included in the appendix. □

**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.* I 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, I also prove two other theorems about my technique. First I 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 my technique indeed performs verification: all possible queries are represented by the ECs, and all queries within an EC have the same behavior.

Second, I prove that, in the absence of `DNAME` rewrites, my 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 `DNAME`s in practice (Section 5.7), 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, my 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.

## 5.6 Implementation

GROOT is implemented in over 2100 lines of C++ code and uses the Boost Graph Library [Bem05] 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, I 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, I 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].

## 5.7 Evaluation

To evaluate GROOT, I aim to show -

1. it can find bugs in real DNS configurations, and
2. it can scale to large sets of zone files

I describe my methodology and results next. I evaluate GROOT on zone files from three networks:

**A university network.** I 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 [Inf22] 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.

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

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

I use the `campus.edu` zone file and the zone files of the 895 subdomains that are secondaried by the campus nameservers for my experiments, since I am 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 5.4(a) shows the cumulative distribution of subdomains to the number of resource records that they contain.

**An infrastructure service provider.** I 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 [DNS13]. 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, I picked the resource records corresponding to the latest timestamp. This leaves 1.05 billion resource records. I partition this set into zone files by using the `SOA` records and the DNS namespace hierarchy.

While creating the zones I 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 my experiments, I considered all the second-level domains (for example, `co.uk.`) 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 [KBA20b]. Figure 5.4(b) shows the distribution of second-level domains to the number of subzones they contain.

**Features used.** Table 5.3 shows a summary of the features used in the three datasets. For the campus network, there were 63 wildcard records and over 4000 `CNAME` records. However,

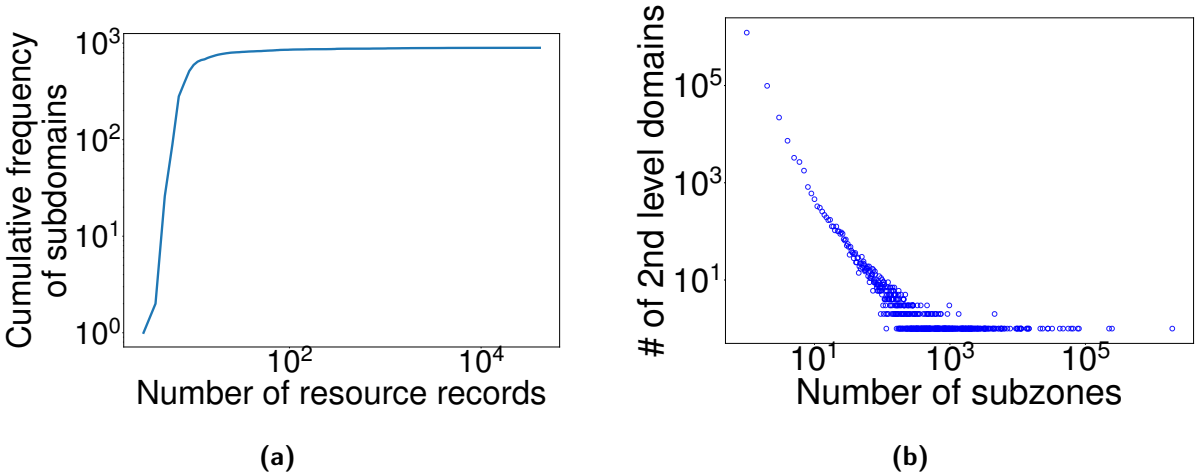



**Figure 5.4:** Dataset statistics. (a) Cumulative number of subdomains with a number of resource records in the campus network. (b) Number of 2nd-level domains with a given number of subzones for DNS census.

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

**Table 5.3:** Summary of features used in the three datasets studied.

the configurations did not make use of DNAME records. In contrast, the part of the DNS census dataset that I 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.

### 5.7.1 Functionality Experiments

#### 5.7.1.1 University network

I use the data from the university network to evaluate whether GRoot can find bugs on a real network. I performed two different classes of checks using GRoot (summarized in Table 5.4) based on properties described in Table 5.1. For the properties in Table 5.1 but not in Table 5.4, 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, I 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 5.4 represent true misconfigurations and are common operational and configuration errors described in RFC 1912 [Bar96]. I contacted operators, and those that responded confirmed our findings (because DNS management is decentralized there are many administrators responsible for these domains and I did not hear back from all of them). I discuss some example violations here:

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

| 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 $\leq 2$ | 24† |

**Table 5.4:** Properties checked on the campus network and the number of cases GRoot reported. Cases in red (⋆) are bugs while orange(†) are warnings.

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 I 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  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 5.4 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 I 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 $\alpha$.`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 [Moc87a] 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 [BR18].

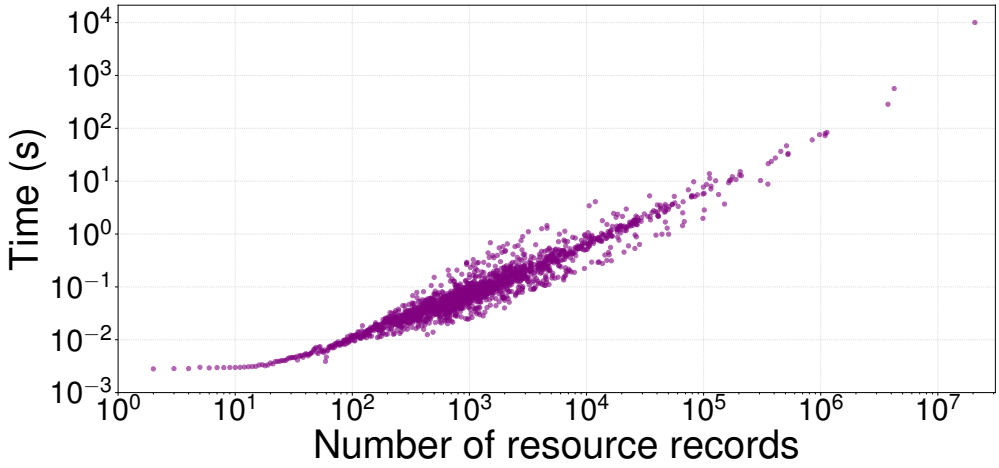



**Figure 5.5:** Total time to build label graph and check for properties for the 1,368,523 second-level TLDs. The median time is taken when multiple domains have the same number of resource records.

#### 5.7.1.2 Service Provider

I also performed checks based on the properties described in Table 5.1 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.

### 5.7.2 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.

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

Figure 5.5 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.

## 5.8 Discussion

To my 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 [Ama10, Mic20, Goo22, Aka20]. 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. I leave optimizing GROOT for small incremental changes for future work.

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

## 5.9 Related Work

GROOT is related to several prior lines of work - DNS testing, DNS modeling and network verification.

I have already mentioned how DNS testing and monitoring approaches are insufficient as they are incomplete, provides no guarantees and are reactive in Section 1.4.3 and in the beginning of the chapter. Another popular approach relevant in this space is *linting* of DNS configuration files. Tools like dnslint [Mic18] 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 non-existent domain, `NXDOMAIN`).

IRONSIDES [CF12], is a DNS server implementation that is provably robust to data flow exceptions such as unexpected exceptions. However, as said earlier in Section 4.8, 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 emerged as an exciting area at the intersection of networking, programming languages, and formal methods in the past decade due to the rapid growth in the scale and complexity of modern network and by the high frequency of outages [KVM12, KZC12, MKA11, LBG15, FSF16, GVA16, BGM17]. However, it has been limited to verifying the network routing layer. While there are some superficial similarities between routing and DNS, the details are vastly different as various routing-specific insights were used to optimize the developed tools. Therefore, those approaches are not directly suitable for DNS. For certain cases, GRoot can generate equivalence classes asymptotically faster than approaches used for routing verification due to the hierarchical structure of domain names.

## 5.10 Summary

In this chapter, I presented GRoot, the first verification tool for DNS configurations. Using my formal model of the DNS, I described, and proved 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. I demonstrated how GRoot can check wide variety of properties by modeling the properties as graph properties on the resulting symbolic execution graphs. Finally, I showed that GRoot can efficiently analyze real DNS configurations in practice, leading to the discovery of numerous misconfigurations.

My 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.

# CHAPTER 6

# DNS Complexity Analysis

In the previous chapters, I have described techniques and tools to handle DNS protocol implementation errors and configuration errors. The main reason for requiring such elaborate techniques presented in the last chapters is that DNS has grown from its intended simple distributed key-value store into a complex beast with many intricate and subtle features. New record types and protocol features are proposed frequently and added to the protocol to enable realistic use cases without considering the collective impact such extensions can have on the broader system through their mutual interactions.

The prior work has shown that network protocols indeed have surprising and accidental complexity. The best example of accidental complexity in the network protocols is in the Border Gateway Protocol (BGP) [RHL06]. Designed to enable routing over the Internet among organizations with different, often conflicting policies, BGP was created to support an extremely rich set of policies. It took many years for theoreticians to "catch up" to practice, demonstrating that the seemingly simple policy mechanisms in BGP can be used to simulate an arbitrary Turing machine [CCD13].

The theoretical complexity of a networked system is important because it has broad ramifications related to the ease with which humans and machines can analyze the system. For instance, even for finite network topologies, simply determining if BGP will converge is NP-Complete [GSW02].

In this chapter, I analyze and understand the theoretical complexity of the DNS. Arguably, DNS is as crucial and widely deployed as BGP, and understanding its complexity has

implications on the cost of verification. Further, unlike BGP, DNS's power can be directly used by applications [Bri95, CK13].

As part of my investigation, I find that the DNS has surprising complexity. I first show in Section 6.1 that `DNAME` rewriting [RW12b], a seemingly simple record type for domain redirection, allows the DNS to recognize arbitrary regular languages encoded in the string labels of the DNS query. Hence users can perform complex validation and lookup logic (*e.g.*, string validation, domain filtering, parental controls, *etc.*) in the DNS itself as part of the configurable records that are processed at authoritative nameservers.

Second in Section 6.2, I demonstrate that the expressiveness of the DNS is beyond that of regular languages. Specifically, the combination of `DNAME` records and nondeterminism due to *nameserver delegation* allows the DNS to encode both deterministic and nondeterministic pushdown systems (PDS) [BEM97, BS95, RSJ05] and hence to generate strings of arbitrary context-free grammars (*e.g.*, strings of the form $a^n b^n$). Section 6.3 discusses the consequences of these results.

## 6.1 Deterministic Finite Automata

We first define a deterministic finite automaton (DFA) and then show how a DFA can be encoded in DNS. We then give an example and conclude with potential applications.

A DFA is a machine with only a read-only tape which it reads from left to right without changing direction. The finite control allows a DFA to read one input symbol from the input tape. Then based on the machine's current state, it may change state. As part of each computational step, the input tape head is repositioned one square further to the right to allow it to read the next input symbol [Hat12]. A DFA is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the input string [HMU06]. DFAs recognize exactly the set of regular languages – languages that use regular expressions [HMU06].

Formally, a deterministic finite automaton $\mathcal{M}$ is a quintuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ such that $Q$ is a finite set of states, $\Sigma$ is a finite set of input symbols called the alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is an initial or start state, and $F \subseteq Q$ is a set of final or accept states. The *language* of $\mathcal{M}$, denoted $\mathcal{L}(\mathcal{M})$, is the set of strings whose processing by $\mathcal{M}$ ends in a final state.

In a DFA, for a particular input symbol, the machine goes to one state only. For every state, a transition is defined for every input symbol, *i.e.,* they are complete. Null (or $\epsilon$) transitions are not allowed, *i.e.,* a DFA cannot change state without any input symbol. A nondeterministic finite automaton (NFA) is one that does not need to obey these restrictions, but it is proven that NFA and DFA have the same computation power and an NFA can be converted to a DFA using Rabin–Scott powerset construction algorithm [RS59].

While a DFA is a mathematical concept, it is often implemented in hardware and software for solving specific problems such as lexical analysis in compilers and pattern matching. For example, a DFA can model software that decides whether or not online user input such as email addresses are syntactically valid.

### 6.1.1 Encoding an arbitrary DFA in DNS

Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ be any DFA. Let $Q = \{q_0, q_1, \cdots, q_n\}$, $\Sigma = \{a_0, \cdots, a_m\}$. We show that $\mathcal{M}$ can be encoded in DNS using a single zone. Let the zone file be for the domain `dfa.com`.. Intuitively, we use the DNS query to encode both the remaining input string and the current state. We then use `DNAME` records to encode the transition relation and use `TXT` records to encode the final accept/reject status.

The steps to encode a DFA $\mathcal{M}$ as a zone file $z$ are:

- **Start:** For each symbol $a_i$ in the alphabet, add a `DNAME` record of the form "`a_i DNAME a_i.q_0`", where $q_0$ is the start state.[1] These records add the start state to the beginning of the query without consuming any input.
- **Transition:** For each transition of the form $q_i \times a_j \to q_k$ in $\delta$, add a `DNAME` record "`a_i.q_j DNAME q_k`". These records consume an input symbol $a_i$ in a state $q_j$ and move to the state $q_k$. By the DNS semantics, these `DNAME` records only apply to a query if it is a strict subdomain of `a_i.q_j`, so these records have the effect of transitioning the system from the start state to the penultimate state, with one input symbol remaining.
- **Decision:** For each transition of the form $q_i \times a_j \to q_k$, add a `TXT` record - " `a_j.q_i TXT "accept""` if $q_k \in F$; otherwise add "`a_j.q_i TXT "reject""`. These records are the final step in the transition system, where $a_j$ is the last input symbol and the system is in state $q_i$.

To test whether a given string $S \in \Sigma^*$ is accepted by the DFA $\mathcal{M}$, we encode $S = s_0 s_1 \cdots s_n$ as the domain name `s_n.⋯.s_1.s_0.dfa.com.`. This query is then sent by the resolver to the nameserver that contains the zone file $z$. The text record response will contain "`accept`" if and only if the string $S \in \mathcal{L}(\mathcal{M})$.

### 6.1.2 Example

In this subsection, we show an example DFA and its encoding in DNS using the three steps mentioned above. Consider the DFA $\mathcal{M}_\emptyset$ over alphabet $\{a, b\}$ shown in Figure 6.1, which accepts all strings that contain an odd number of $a$'s.

Table 6.1 shows the encoding of DFA $\mathcal{M}_\emptyset$ shown in Figure 6.1 in DNS as a zone file $z$. To make it a valid zone, there must also exist an `SOA` and `NS` record for `dfa.com.`, which are omitted for brevity. To test whether the string *abaa* is accepted by the $\mathcal{M}_\emptyset$, we send the query

[1] For exposition purposes we use *relative* domains here, which lack the trailing ".": implicitly the zone domain `dfa.com.` is appended to form the complete domain.

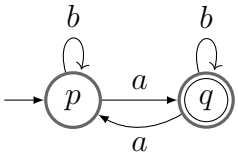



**Figure 6.1:** An example DFA $\mathcal{M}_{\mathcal{O}}$ that accepts strings only if they contain an odd number of a's

| | | |
|---|---|---|
| Start | ① | a IN DNAME a.p |
| | ② | b IN DNAME b.p |
| Transition | ③ | a.p IN DNAME q |
| | ④ | b.p IN DNAME p |
| | ⑤ | a.q IN DNAME p |
| | ⑥ | b.q IN DNAME q |
| Descision | ⑦ | a.p IN TXT "accept" |
| | ⑧ | b.p IN TXT "reject" |
| | ⑨ | a.q IN TXT "reject" |
| | ⑩ | b.q IN TXT "accept" |

**Table 6.1:** Zone file $z$ showing the enconding of DFA $\mathcal{M}_{\mathcal{O}}$ shown in Figure 6.1.

$\langle$a.a.b.a.dfa.com., TXT$\rangle$ to the nameserver serving $z$. The steps followed by the nameserver to resolve the query are shown below.

$$\langle \texttt{a.a.b.a}, \texttt{TXT} \rangle \xrightarrow{①} \langle \texttt{a.a.b.a.p}, \texttt{TXT} \rangle \xrightarrow{③} \langle \texttt{a.a.b.q}, \texttt{TXT} \rangle \xrightarrow{⑥} \langle \texttt{a.a.q}, \texttt{TXT} \rangle \xrightarrow{⑤} \langle \texttt{a.p}, \texttt{TXT} \rangle \longrightarrow ⑦$$

The nameserver returns the entire trace along with the TXT (⑦) record. Since the TXT record received contains "accept" in its content, the string is accepted by $\mathcal{M}_{\mathcal{O}}$. Thus, given any DFA, it can be encoded in DNS with a single zone file, and to test whether a string belongs to the language accepted by the DFA, one can send a DNS query to the nameserver serving the zone file.

In practice, there are zone files with millions of records; therefore, complex DFAs with many states and transitions can easily be encoded in DNS. We wrote a small script to

encode a DFA in DNS and successfully tested it with two popular DNS implementations, BIND [Con86] and NSD [Lab02a]. In DNS, the domain name has certain length restrictions; specifically, the domain name cannot be longer than 255 characters, and each label cannot be more than 63 characters. The nameserver can also limit the number of rewrites that it will perform on a query. However, various techniques can be used to overcome such limitations. For example, we can map pairs of alphabet symbols from the DFA to single labels in DNS and then change the encoding of the transition relation to consume multiple symbols at a time, thereby processing longer DFA input strings.

### 6.1.3 Applications

Regexes are frequently used to validate user input for well-formedness. For example, the regex "`^[a-zA-Z0-9+_.-]+@[a-zA-Z0-9.-]+`" is a simple validator for email addresses. Since a regex can be represented as a DFA [Tho68], using the construction detailed in the previous subsection we can validate if user input is a proper email address or not.

While the idea of using the DNS to check input well-formedness may seem far-fetched, we believe that it could have some natural use cases. For example, organizations generally want to control what domains their employees can visit while using their office devices, due to security and various other reasons. If the allowed domains can be represented as a regular expression, then this validation can be done in the DNS, *as part of the DNS lookup for the domain*. Office devices are generally configured to use specific DNS resolvers. Therefore, the resolver could first use our approach, with a local DNS nameserver implementing the policy DFA, to check that the user's DNS query is to an allowed domain, and only then send it to the outside world in order to resolve it to an IP address. A similar setup could be used for parental control in the home setting. Doing this directly in the DNS gives a single, global, always-available vantage from which to enforce policies.

## 6.2 Pushdown System

While the DNS can encode finite automata, its expressiveness goes beyond that of regular languages. In this section, I show that the DNS can express nondeterministic pushdown systems and by extension can generate strings in arbitrary context-free languages.

A pushdown system is a transition system equipped with a finite set of *control locations* and a *stack*. The stack contains a word over some finite stack alphabet; its length is unbounded. Hence, a pushdown system may have infinitely many reachable states. An important use of pushdown system is in representing sequential programs with (possibly recursive) functions [RLK07]. These programs in general cannot be modeled using finite-state machines as there is no limit on the depth of the call stack for function calls [Sch02].

**Definition 6.1.** A pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ is a quadruple, where $P$ and $\Gamma$ are finite sets called the control locations and the stack alphabet, respectively. A configuration of $\mathcal{P}$ is a pair $\langle p, w \rangle$, where $p \in P$ and $w \in \Gamma^*$, and $c_0$ is the initial configuration. The set of all configurations is denoted by $Conf(\mathcal{P})$. $\Delta$ is a finite subset of $(P \times \Gamma) \times (P \times \Gamma^*)$, which consists rules of the form $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$, where $p, p' \in P, \gamma \in \Gamma$, and $w \in \Gamma^*$. These rules define the transition relation $\Rightarrow$ between configurations of $\mathcal{P}$ as follows:

$$\textit{If } \langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle, \textit{ then } \langle p, \gamma w' \rangle \Rightarrow \langle p', w w' \rangle \textit{ for all } w' \in \Gamma^*.$$

As shown above, each step depends only on the control location ($p$) and the topmost element ($\gamma$) of the stack ($\gamma w'$). The rest of the stack ($w'$) is unchanged and has no influence on the possible next actions.

### 6.2.1 Encoding a PDS in DNS

With the help of an example, we show how a PDS can be encoded in DNS. Similar to how we encoded a DFA, we will encode both the stack and the current state in the query

and use DNAME records to implement the transition relation. We employ multiple zone files and nameservers and *delegate* among them to encode any nondeterminism in the transition relation.

Consider a PDS $\mathcal{P}$ with $P = \{\mathsf{p}, \mathsf{q}\}$, $\Gamma = \{a, b, c, d\}$, $c_0 = \langle \mathsf{p}, c \rangle$ and $\Delta$ given by:

$$\begin{aligned} r_1 &= \langle \mathsf{p}, a \rangle \hookrightarrow \langle \mathsf{q}, b \rangle \quad & r_2 &= \langle \mathsf{p}, a \rangle \hookrightarrow \langle \mathsf{p}, c \rangle \\ r_3 &= \langle \mathsf{q}, b \rangle \hookrightarrow \langle \mathsf{p}, d \rangle \quad & r_4 &= \langle \mathsf{p}, c \rangle \hookrightarrow \langle \mathsf{p}, ad \rangle \\ r_5 &= \langle \mathsf{p}, d \rangle \hookrightarrow \langle \mathsf{p}, \epsilon \rangle \end{aligned}$$

We show some transitions between different configurations of $\mathcal{P}$ starting with $c_0$ and with the rules given by $\Delta$.

$$\langle \mathsf{p}, c \rangle \xrightarrow{r_4} \langle \mathsf{p}, ad \rangle \begin{cases} \xrightarrow{r_1} \langle \mathsf{q}, bd \rangle \xrightarrow{r_3} \langle \mathsf{p}, dd \rangle \xrightarrow{r_5} \langle \mathsf{p}, d \rangle \xrightarrow{r_5} \langle \mathsf{p}, \epsilon \rangle & \text{(i)} \\ \xrightarrow{r_2} \langle \mathsf{p}, cd \rangle \xrightarrow{r_4} \langle \mathsf{p}, add \rangle \begin{cases} \xrightarrow{r_1} \langle \mathsf{q}, bdd \rangle \rightarrow & \text{(ii)} \\ \xrightarrow{r_2} \langle \mathsf{p}, cdd \rangle \rightarrow & \text{(iii)} \end{cases} \end{cases}$$

As with the DFA encoding, we assume we control the pds.com. domain and all its subdomains. We create the pds.com. zone file as shown below and place it in the server1.pds.com. nameserver. The resolver is bootstrapped with the IP address of this nameserver.

In the pds.com. zone, we first encode all the deterministic rules, $r_3, r_4, r_5$ with DNAME records, 1, 2, and 3. We then use DNS delegation (referral to a new nameserver) for each nondeterministic set of rules. This apporach leverages the nondeterminism inherent in DNS delegation – given multiple NS records, DNS implementations will chose one nondeterministically. Here we have only one set of nondeterministic rules, namely $\langle \mathsf{p}, a \rangle$ with two rules. For each rule, we create an NS record (4 and 5) and assign it a nameserver not previously assigned. For each NS record, we also add a glue record (6 and 7) to provide the IP address of the nameserver.

**nameserver:** server1.pds.com.

```
$ORIGIN pds.com.
     b.q   DNAME   d.p        [1]
     c.p   DNAME   d.a.p      [2]
     d.p   DNAME   p          [3]
     a.p   NS      server2    [4]
     a.p   NS      server3    [5]
server2    A       2.2.2.2    [6]
server3    A       3.3.3.3    [7]
```

We then create a zone file for a.p.pds.com. at each of the delegated nameservers and place a DNAME record for each nondeterministic rule at a unique nameserver. In our example we end up with two nameservers and zone files:

**nameserver:** server2.pds.com.

```
$ORIGIN a.p.pds.com.

a.p.pds.com.   DNAME   b.q.pds.com.   [8]
```

**nameserver:** server3.pds.com.

```
$ORIGIN a.p.pds.com.

a.p.pds.com.   DNAME   c.p.pds.com.   [9]
```

To execute the PDS from the initial configuration $\langle \mathtt{p}, c \rangle$, we ask the DNS query $\langle \beta.\mathtt{c.p}, \mathtt{TXT} \rangle$. As with the DFA, the query encodes the current stack followed by current state. Additionally, we start the query with a dummy subdomain $\beta$. This is necessary since DNAME records only apply to strict subdomains; doing so ensures that the DNAME records apply even when the stack contains only a single element.

One possible execution starting from the query $\langle \beta.\mathtt{c.p}, \mathtt{TXT} \rangle$ at the resolver is as follows:

1. **Resolver:** Queries the default server Server1 with the query $\langle \beta.\mathtt{c.p}, \mathtt{TXT} \rangle$.

2. **Server1:** The server first rewrites the query, and the best records for the new query are `NS` records. The server returns the rewrite, the delegation records, and the corresponding glue records to the resolver.

   (a) $R_1$ - $\langle \beta.\texttt{c.p}, \texttt{TXT} \rangle \xrightarrow{\boxed{2}} \langle \beta.\texttt{d.a.p}, \texttt{TXT} \rangle$

   (b) Delegation - 4, 5, 6, and 7

3. **Resolver:** The resolver now has a choice to contact either Server2 (4) or Server3 (5). We show the sequence of steps if the resolver sends the query $\langle \beta.\texttt{d.a.p}, \texttt{TXT} \rangle$ to Server2.

4. **Server2:** Rewrites the query and returns it. $R_2$ - $\langle \beta.\texttt{d.a.p}, \texttt{TXT} \rangle \xrightarrow{\boxed{8}} \langle \beta.\texttt{d.b.q}, \texttt{TXT} \rangle$

5. **Resolver:** Queries Server1 again.

6. **Server1:** Rewrites the query three times and returns the final rewritten query $\big(\langle \beta.\texttt{p}, \texttt{TXT} \rangle\big)$ to the resolver.

   (a) $R_3$ - $\langle \beta.\texttt{d.b.q}, \texttt{TXT} \rangle \xrightarrow{\boxed{1}} \langle \beta.\texttt{d.d.p}, \texttt{TXT} \rangle$

   (b) $R_4$ - $\langle \beta.\texttt{d.d.p}, \texttt{TXT} \rangle \xrightarrow{\boxed{3}} \langle \beta.\texttt{d.p}, \texttt{TXT} \rangle$

   (c) $R_5$ - $\langle \beta.\texttt{d.p}, \texttt{TXT} \rangle \xrightarrow{\boxed{3}} \langle \beta.\texttt{p}, \texttt{TXT} \rangle$

When the resolver gets the response from the Server1 in step 6, it is clear that the stack is empty as the domain name has only the control symbol and the dummy subdomain we added. If we put together all the rewrites ($R_1$, $R_2$, $R_3$, $R_4$, and $R_5$) starting from the first rewrite then we have the trace corresponding to the top trace (i) shown earlier in transitions between different configurations in Section 6.1.2.

Next we show how a different set of configurations can be explored if the resolver instead chose Server3 at step 4 above. The steps in that case would be:

4'. **Server3:** $\langle \beta.\texttt{d.a.p}, \texttt{TXT} \rangle \xrightarrow{\boxed{9}} \langle \beta.\texttt{d.c.p}, \texttt{TXT} \rangle$

5'. **Resolver:** Queries Server1 with the rewritten query.

6'. **Server1:** The server first rewrites the query (a) and the new query is again delegated. The server returns both the steps and the records involved to the resolver as it doesn't remember any previous exchanges.

(a) $\langle \beta.\texttt{d.c.p}, \texttt{TXT} \rangle \xrightarrow{\boxed{2}} \langle \beta.\texttt{d.d.a.p}, \texttt{TXT} \rangle$

(b) Delegation - 4, 5, 6, and 7

The resolver now again has a choice, and different options lead to the configurations (ii), (iii) shown earlier. If the number of rewrite steps becomes more than a threshold, the nameserver can stop processing further and return the rewrites with a warning message to the resolver. To overcome this limitation and explore more configurations, the resolver can send a fresh query from the last stopped configuration instead of the initial configuration.

In this way we can use the DNS to explore the reachable configurations of a PDS. Generally records returned to the resolver have a time to live (TTL) field for caching. The resolver will use the local cache when a matching query comes, thus slowing down the exploration of other configurations. We can avoid this by setting the TTL of the `DNAME` records to be small, even to 0. Another issue is that nameservers often have a limit on the number of rewrites they will perform, at which point they stop processing the query further and return it. To overcome this limitation and explore more configurations, the resolver can then send a fresh query from that last configuration.

So far, we have seen how to explore reachable configurations of a PDS using the DNS. In the next subsection we will describe how we can use this capability to *generate* strings from any context-free language.

### 6.2.2 Context-free Language Generator

A formal grammar is a set of production rules that describe all possible strings in a given formal language. A context-free grammar (CFG) is a formal grammar whose production rules are of the form "$A \to \alpha$", with $A$ being a single nonterminal symbol, and $\alpha$ a string

of terminals and/or nonterminals ($\alpha$ can be empty). A formal grammar is "context free" if its production rules can be applied regardless of the context of a nonterminal. Context-free grammars generate context-free languages, which are strictly more expressive than regular expressions. Context-free languages have many applications in programming languages; in particular, most programming language syntaxes are specified by context-free grammars.

Formally, a context-free grammar $\mathcal{G}$ is defined as a 4-tuple $\mathcal{G} = (N, \Sigma, P, S)$, where $N$ is a finite set of non-terminal symbols, and $\Sigma$ is a finite set of terminal symbols disjoint from $N$. The set of terminals is the alphabet of the language defined by the grammar $\mathcal{G}$. $P$ is the set of production rules and is a finite relation in $N \times (N \cup \Sigma)^*$. $S$ is the start symbol and is one of the non-terminal symbols in $N$.

We derive strings in the language of a CFG by starting with the start symbol and repeatedly replacing some non-terminal by the right side of one of its production rules. Consider the context-free language $L = \{a^n b^n : n \geq 1\}$. The grammar of this language, with the start symbol $S$ is:

$$S \to aSb \tag{6.1}$$

$$S \to ab \tag{6.2}$$

The string $a^3b^3$ in this language is generated by applying rule (4.1) twice followed by (4.2): $S \to aSb \to aaSbb \to aaabbb$.

We first describe a program variant of the above grammar and show how that program can be represented using a pushdown system. Then based on the encoding described in Section 6.2.1 we can implement this in the DNS.

A program that generates the strings in $L$ is given in Figure 6.2. Here $\ell_1$, $\ell_2$, and other such symbols are used to denote each program location (line of code) uniquely, which will be later used as the stack alphabet $\Gamma$ in our PDS. Since $S$ is the start symbol in the grammar,

```
   procedure S1:        procedure S2:        procedure S:
ℓ1  output a         ℓ5  output a          ℓ8  if ?
ℓ2  call S           ℓ6  output b          ℓ9   call S1
ℓ3  output b         ℓ7  return            ℓ10 else call S2
ℓ4  return                                 ℓ11 return
```

**Figure 6.2:** Program to generate strings in language $L = \{a^n b^n : n \geq 1\}$.

the `procedure S` would be called to start the program. The symbol "`?`" in $\ell_8$ represents nondeterministic choice, reflecting the nondeterminism in the original grammar.

The technique used to convert the above grammar into a program can be generalized as follows. Let $\mathcal{G}$ be a CFG. First, create a uniquely named procedure (disjoint from $\Sigma \cup N$) for each production rule in $P$, as in `S1` and `S2` in the above example. The body of the procedure then encodes the right side of the corresponding rule. Specifically, there is an "`output t`" line for each terminal symbol $t$ in the right side and a "`call A`" line for each non-terminal $A$ in the right side of the rule, in order of their appearance in the rule. Finally, for every non-terminal $A$ in $N$ create a "`procedure A`" and use an `if` statement to nondeterministically call one of the procedures created in the previous step whose corresponding rule has $A$ on the left side.

We can create a PDS that encodes all possible executions of such a program [Sch02]. The PDS has a single control location and uses the program labels as the stack alphabet. For example, a PDS $\mathcal{P}_L$ for the program shown above has $P = \{\mathsf{p}\}$, $\Gamma = \{\ell_1, \cdots, \ell_{11}\}$, and $c_0 = \langle \mathsf{p}, \ell_8 \rangle$. $\Delta$ is given by:

$$
\begin{array}{lll}
\langle \mathsf{p}, \ell_1 \rangle \hookrightarrow \langle \mathsf{p}, \ell_2 \rangle & \langle \mathsf{p}, \ell_5 \rangle \hookrightarrow \langle \mathsf{p}, \ell_6 \rangle & \langle \mathsf{p}, \ell_8 \rangle \hookrightarrow \langle \mathsf{p}, \ell_9 \rangle \\
\langle \mathsf{p}, \ell_2 \rangle \hookrightarrow \langle \mathsf{p}, \ell_8 \ell_3 \rangle & \langle \mathsf{p}, \ell_6 \rangle \hookrightarrow \langle \mathsf{p}, \ell_7 \rangle & \langle \mathsf{p}, \ell_8 \rangle \hookrightarrow \langle \mathsf{p}, \ell_{10} \rangle \\
\langle \mathsf{p}, \ell_3 \rangle \hookrightarrow \langle \mathsf{p}, \ell_4 \rangle & \langle \mathsf{p}, \ell_7 \rangle \hookrightarrow \langle \mathsf{p}, \epsilon \rangle & \langle \mathsf{p}, \ell_9 \rangle \hookrightarrow \langle \mathsf{p}, \ell_1 \ell_{11} \rangle \\
\langle \mathsf{p}, \ell_4 \rangle \hookrightarrow \langle \mathsf{p}, \epsilon \rangle & & \langle \mathsf{p}, \ell_{10} \rangle \hookrightarrow \langle \mathsf{p}, \ell_5 \ell_{11} \rangle \\
 & & \langle \mathsf{p}, \ell_{11} \rangle \hookrightarrow \langle \mathsf{p}, \epsilon \rangle
\end{array}
$$

Intuitively, $\Delta$ encodes the control flow of the program. For statements where control passes from one line to the next (here just `output`), we add rules of the form $\langle \mathsf{p}, \ell_1 \rangle \hookrightarrow \langle \mathsf{p}, \ell_2 \rangle$.

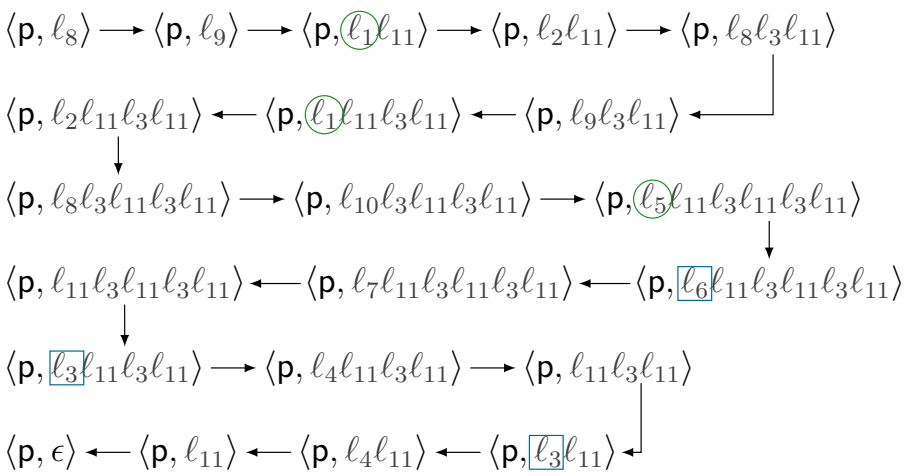



**Figure 6.3:** The full trace of PDS $\mathcal{P}_L$ that generates string $a^3b^3$. The `output` code lines that are on top of the stack are shown with circles for $a$ and squares for $b$.

A procedure call (for example, at $\ell_2$) is encoded by pushing the return point ($\ell_3$) followed by the called procedure's ($\ell_8$) starting statement. A `return` statement is encoded as a stack pop ($\epsilon$).

Finally, we can use this encoding to generate strings in our original CFG $L$. Define a *full trace* in $\mathcal{P}_L$ as the sequence of configurations starting with $c_0$ and ending with an empty stack. Given a full trace, consider the top symbol of the stack in each configuration, and retain only those symbols that correspond to an `output` code line. If we concatenate the output of those lines, then we obtain a string. The set of such strings is exactly the set of strings defined by the CFG $L$.

For example, the full trace that would generate $a^3b^3$ is shown in Figure 6.3. In the trace, among all the top stack elements only six symbols, shown with circles and squares, represent `output` code lines. If we concatenate them in the order given by the full trace then we obtain the string *aaabbb*.

In summary, we have shown how to encode a PDS in DNS in Section 6.2.1, and here we have shown how to encode a generator for a CFG as a PDS. Hence we can use the DNS system to generate strings in the language of a given CFG.

## 6.3 Discussion

I presented an initial investigation into the complexity of DNS, and there are several directions for future research.

**Impact on DNS Verification.** The most efficient algorithms known for PDS reachability — determining whether a given configuration can be reached in a given PDS — have near-cubic time complexity in the number of rules [EHR00, Cha08, CCP17, CO17, HM97]. Hence DNS zone-file verification [KBA20a], which requires reasoning about all possible query lookups, also has at least this complexity today. This has not only theoretical implications but is also a problem for real zone-file verifiers like GRoot [KBA20a].

Even a simple four-record zone file with interacting `DNAME` loops can create close to a million query equivalence classes in GRoot as `DNAME` records can change the Label graph from a tree to a graph as each `DNAME` record adds a directed edge to the label graph (Section 5.2.2). Figure 6.4(a) shows an example zone file with two interacting `DNAME` records and Figure 6.4(b) shows the corresponding Label graph GRoot constructs to generate query equivalence classes for this zone file. This is a contrived example, and `DNAME` records are not frequently used; therefore, GRoot will work very well in general cases. The loops in Figure 6.4(b) are composed of both solid and dotted edges and are guaranteed to terminate when the domain name of the path exceeds the maximum length allowed by DNS. But, the number of possible domain names with the combinations of the three labels `baz`, `bar`, and `zot` quickly blow up to more than a million. The time to check for a property on such zone files also blows up as GRoot constructs an interpretation graph for each equivalence class.

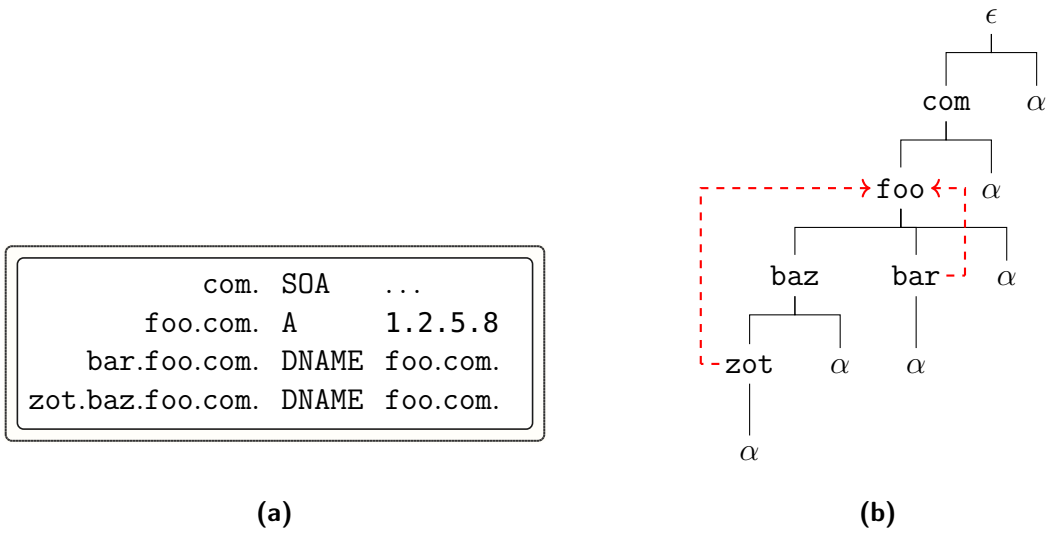



**Figure 6.4:** Example (a) zone file and the corresponding (b) Label graph GROOT constructs to generate query equivalence classes. Due to the interacting `DNAME` loops in (b), GROOT generates more than a million equivalence classes.

Interestingly, however, verification in GROOT is linear time in the absence of `DNAME` records. Can we design new verification algorithms that scale well with the number of `DNAME` records for real-world configurations? I will give an intuition for such an algorithm using model checking of PDS in Section 7.1.1.

**Tighter Bounds.** Can we show that the DNS is even more expressive than a PDS? Alternatively can we reduce the DNS to a PDS and hence show that they are equivalent? I believe DNS and PDS have the same power and are equivalent. I will briefly describe this equivalence in the future work Section 7.1.1, as the complete proof is still a work in progress.

**Applications.** The applications that we have presented are somewhat contrived. Can we build a real application that takes advantage of the complexity of DNS? We can take

inspiration from existing applications that use the DNS, ranging from service discovery [CK13] to load balancing [Bri95, Inc19] to spam filtering [DNS22a, KCH11, Kit14, DNS22b].

**New Record Types.** Contributors frequently add new drafts and RFCs to the DNS specification, with new record types intended to enable new use cases. For example, the recent NAPTR record type [Mea02] supports prioritized regular expressions that provide lookup for dynamic resources. How do these newly proposed types affect DNS complexity?

**Security Implications.** Does the complexity of DNS have security implications? This is a natural direction to explore. However, we note that, unlike for conventional DNS attacks, the attacker must control the target's zone files in order to leverage the complexity of DNS.

## 6.4 Summary

In this chapter, I investigated the computational complexity of DNS and shown its ability to simulate a finite state machine and a pushdown system. While this is in the spirit of earlier investigations into the complexity of protocols like BGP, I note, unlike BGP, DNS features are available to applications. Thus besides the verification implications of computational power, DNS computational power is a two-edged sword. It can allow useful systems (such as organizational access control) and is potentially an enabler of new applications. On the other hand, it can also provide potentially increased power to an adversary.

# CHAPTER 7

# Conclusion

The Domain Name System or DNS is one of the pillars of the Internet. Today, every user device requires a correct working DNS to translate a domain name to an IP address in order to access various resources and services online. Despite its importance as the "phonebook" of the Internet, DNS is fraught with configuration errors and implementation errors that have far-reaching disruptive consequences, impacting millions of users. Even though the DNS has been around for more than three decades, the impact of such errors was not felt until a decade ago, when cloud networking grew exponentially. As more and more services moved online, even a tiny error in DNS had a huge blast radius, as is evident from past DNS issues in the decade that have rendered popular services such as Slack [Raf21], Salesforce [Spe21], GitHub [Fry14], HBO [Yor15], LinkedIn [Zel13], and Azure [Tun19] inaccessible for extended periods.

A simple DNS misconfiguration was the reason behind one of Microsoft's worst outages, which affected many of the Azure services [Tun19]. This outage was what first piqued my interest in DNS, and was the motivation behind my thesis. After this incident, I started investigating state-of-the-art DNS management techniques during an internship at Microsoft research. I realized DNS was managed the same way the routing layer was managed a decade ago, using live testing and monitoring approaches. I decided to apply formal methods to DNS. I was inspired by the fact that formal methods have been widely successful in handling the complexity and the scale of the *routing layer* of modern networks, while also providing strong

proactive guarantees. This dissertation presents my contribution in making DNS robust with the help of formal methods.

Unfortunately, treating DNS as a black box and applying classical formal techniques does not scale well. Instead, I needed to develop new techniques that combined the traditional formal methods ideas with DNS-specific insights, such as the hierarchical structure of domain names. To develop such techniques, I had to understand DNS deeply, poring over DNS RFCs, only to realize what a daunting task it was. While the RFCs are in an easily readable English format, they were, at the same time, ambiguous and confusing. I also noticed that DNS semantics are relatively poorly understood compared to other routing layer semantics. I often had to test my understanding of the semantics by interpreting the responses from BIND and POWERDNS for small hand-crafted test zone files and queries.

This led me to build a formal model of DNS authoritative and recursive semantics using a mathematical formalization. My goal in doing so was to remove the ambiguity of the specifications, and to serve as the foundation for developing tools using formal methods. Given that DNS has many RFCs, I prioritized developing the model for the record types most frequently used. The developed formal model is in Chapter 3. I believe my formal model of DNS can serve as the basis of future work in this area. Researchers can add other record types and RFCs incrementally to my formal model.

In Chapter 4, I presented SCALE and FERRET, the first techniques for automatically finding RFC compliance errors in DNS nameserver implementations. A unique feature of the SCALE approach is the joint generation of zone files and queries to produce high-coverage behavioral tests. The key insight underlying SCALE is to create an executable version of the formal model developed earlier and then symbolically execute it to generate wide variety of tests that can be used to test any DNS implementation. Intuitively, tests that cover a wide variety of behaviors in the executable model will also cover a wide variety of behaviors in DNS nameservers since they have the same goal, namely to implement the RFCs. I solved many

technical challenges to make SCALE practicable for DNS as FERRET, which are detailed in the chapter.

Using FERRET I tested 8 popular open-source DNS implementations and Amazon Route 53 DNS implementation. FERRET uses a novel hybrid fingerprinting approach for bug deduplication that takes advantage of my formal model to help triage bugs. With the FERRET tests, I discovered 30 new unique bugs and no false positives, including 3 previously unknown critical security vulnerabilities. One of these was a new vulnerability in BIND that attackers could remotely exploit to crash DNS resolvers and nameservers. BIND released a patch and a high-severity CVE-2021-25215 as a result.

Chapter 5 described GROOT, the first verification tool for DNS configurations. Using the formal model of the DNS, I described and proved the correctness of a fast algorithm that generates equivalence classes of DNS queries. These equivalence classes enable GROOT to efficiently and exhaustively check the correctness of DNS zone files. The key insight in GROOT is to generate an exhaustive set of equivalence classes of DNS queries by leveraging the zone files and the hierarchical tree structure of domain names. I applied GROOT to the configuration files obtained from a large campus network which has over a hundred thousand records, and it revealed 109 new bugs and completed in under 10 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, which initiated a cleanup of the zone files.

As the final contribution of this thesis, I presented a theoretical analysis of the complexity of the DNS in Chapter 6. After FERRET and GROOT, I realized DNS was no longer the simple protocol that we are taught in introductory network classes, but instead is rich and complex. I wanted to know what classes of computational problems DNS can solve to help understand the theoretical limits of DNS configuration verification time complexity. I showed that DNS has the power to express regular languages and pushdown systems. Consequently, the verification time of DNS configuration is likely to be cubic in the number of records in

the worst case. I strongly believe DNS is not Turing-complete; I am currently working on a proof to show that DNS and PDS are, in fact, equivalent and hence have the same power (Section 7.1.1).

## 7.1 Future Work and Open Problems

The work in this dissertation made progress towards making DNS robust by providing techniques to address nameserver implementation errors and DNS configuration errors. However, many challenges remain in DNS and beyond DNS in other parts of the cloud networking stack.

### 7.1.1 Equivalence of DNS and PDS

In Section 6.2, we have seen how to encode a pushdown system in the DNS with an example. In this ongoing work, I hope to how to reduce DNS to PDS and hence, formally prove their equivalence. In other words, for every step in DNS, there is an equivalent step in PDS and vice-versa. The intuition behind the equivalence is shown in Figure 7.1. Both PDS and DNS are term rewriting systems [DJ90] where the only possible rewrite is a prefix rewrite. The prefix rewriting occurs through rules in a PDS where only the top element of the stack can be modified and using the `DNAME` record type in the DNS. The nondeterminism in PDS *i.e.*, presence of more than one rule for the same configuration, is simulated by nameserver delegation in DNS.

The equivalence of DNS and PDS has interesting consequences for zone file verification. If we can reduce a DNS configuration to a PDS, then we can leverage existing work on model checking of pushdown systems for verifying DNS configurations (zone files) in an acceptable time even in the presence of arbitrary `DNAME` records. We have seen how the zone file verification time of GRoot can blow up quickly in the presence of multiple interacting `DNAME` records in Section 6.3. A possible disadvantage of using the PDS approach could be

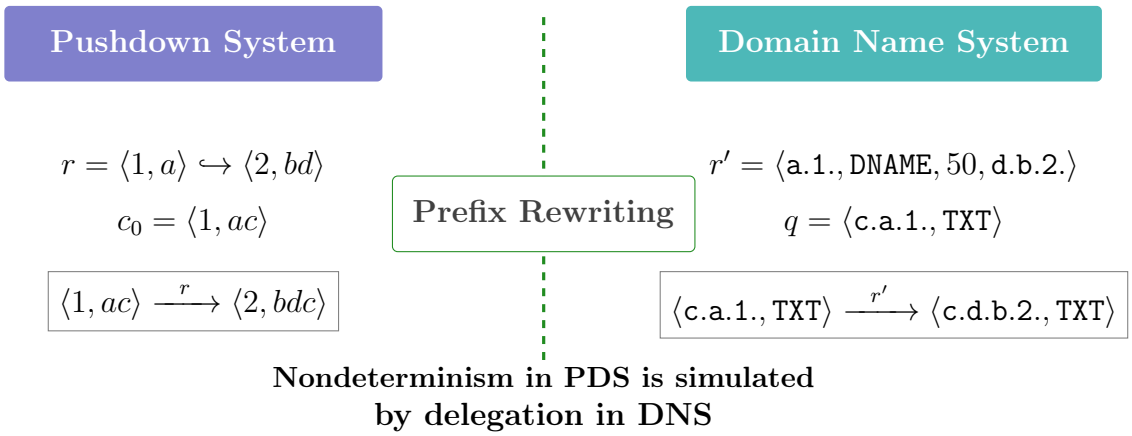



**Figure 7.1:** Equivalence of PDS and DNS.

that one may not be able to check for all the properties that GRoot can check. It might be the case that we can use GRoot in the absence of `DNAME` loops and use the PDS approach when there are loops, so as to have the best of both worlds.

### 7.1.2 Resolver Testing

The DNS can be considered as consisting of three components - zone files, nameservers and resolvers. This thesis focused on the correctness of the first two components. Compared to the authoritative side of DNS, resolvers have relatively simple control logic. The errors in the resolver side of DNS are mostly not functional (RFC compliance) in nature but rather involve performance issues. These include for example, how the implementation handles the memory stack, how it protects from DDoS attacks, how it avoids leaking resources, and how it performs under network constraints [ABS20]. These are harder to quantify and normalize across implementations to decide correct behavior. The test setup for a resolver would also be more complex than nameservers as a test may need to initialize the cache to a required state before sending the test query. It may be interesting in the future to explore how to

extend FERRET to generate such tests for resolvers as well as for nameservers as was done in this thesis.

### 7.1.3 Extending SCALE for Testing Correctness of Other Protocols

SCALE is a general technique that can broadly be applied beyond DNS to other networking protocols, especially to protocols that employ configurations such as BGP. One of the advantages of SCALE is that it is extensible - one can start with a small base model and incrementally add new logic or branches to expand the RFCs covered. The following are the two main challenges in expanding SCALE to other protocols:

- DNS nameservers are stateless, but DNS resolvers, ICMP [Pos81], and BGP [RHL06] protocols are stateful, requiring us to handle more than configurations. For example, in BGP, the response to a route announcement depends not only on the configuration file but also on what previous route announcements were received.
- SCALE is a powerful approach but requires manual effort to build the formal model, however small that is.

I believe the small-scope property of networking protocols will help deal with the statefulness challenge. We will need to model caches and routing tables and generate the sequence of inputs required to pre-configure them before sending the test query or packet. However, the sequence of inputs will likely be small enough for SMT solvers to generate tests in a reasonable time while covering a wide variety of behaviors. For testing BGP implementations, we are using the insight that a BGP implementation compares the input announcement to only the best announcement it has. Therefore only a sequence of two route announcements suffices to test the route priority code of the implementations. We could characterize the protocols by the maximum number of messages needed to prepare any state. For example, for DNS nameservers, this metric is 0 (since nameservers are stateless); on the other hand, for BGP route priority, this metric is 1 (modulo some corner cases); for TCP, this metric is

likely to be larger. In general, doing a complete characterization is hard. However, one could potentially do some analysis on a protocol model to automatically discover "shortcuts" to drive protocols to desired states to test various combinations of inputs and states.

Finally, recent papers automatically learn protocol models from implementations [FBD21] or RFCs using NLP techniques [YLY21]. In particular, SAGE [YLY21] introduced semi-automated protocol processing using domain-specific extensions to semantic parsing to convert ICMP RFC 792 [Pos81] into executable code prototypes. Arguably, the Zen model we have used for DNS is also an executable prototype. Thus in principle we should be able to leverage the SAGE work for SCALE by changing the backend from C++ to Zen. This is a promising step, but it still has a long way for other complex protocols. Nevertheless, it can help. One could potentially adopt these techniques (like leveraging the intermediate logical form of SAGE) in the future to reduce the burden of producing the formal model for SCALE.

# APPENDIX A

# Appendix

Here I give full proofs of soundness, completeness and efficiency of GROOT's equivalence class generation algorithm. First I introduce several helpful definitions. The first definition lets us more easily extract information from DNS answers:

**Definition A.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 A.2** (Real record extraction)**.** Given a set of resource records $R$, we extract those that are not synthesized with real($R$) $= \{r \in R \mid \text{synth}(r) = F\}$. 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 A.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($A_2$).

## A.1 Equivalence Class Generation Algorithm Correctness

**Theorem A.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 $s$:

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

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 $\rho$ 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 $\alpha$.

**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 $\rho$, 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 $q_i = \underbrace{\ldots \circ l_{k+1}}_{\alpha} \circ l_k \circ l_{k-1} \circ \ldots \circ \epsilon$ 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 $\alpha$ 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 $\alpha$, 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 $\textsc{Match}(r, q_1) = \textsc{Match}(r, q_2)$ 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 $\alpha$ 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** ($\mathrm{dn}(q_1) \in_* \mathrm{dn}(R)$). In this case, the matching record(s) are wildcard records. From before, we know that $\mathrm{dn}(q_2) \in_* \mathrm{dn}(R)$. We therefore get the following:

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

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:

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

Expanding the definition of Syn:

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

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

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

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:

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

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

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

**Case** ($\mathrm{dn}(R) < \mathrm{dn}(q_1),\ \texttt{DNAME} \notin T, \texttt{NS} \in T, \texttt{SOA} \notin T$). As in the previous case, we know that $\mathrm{dn}(R) < \mathrm{dn}(q_2)$. 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** ($q_1' \in \mathrm{query}(a_1), q_2' \in \mathrm{query}(a_2) \implies q_1' \sim_C q_2'$): 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 $\mathrm{query}(a_i) = \emptyset$.

**Case** ($\mathrm{dn}(R) = \mathrm{dn}(q_1), \textsc{Authoritative}(T), \mathrm{ty}(q_1) \notin T, \texttt{CNAME} \in T, R = \{r\}$). 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:

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

For this `DNAME` case, we know that $\text{dn}(q_i)$ (represented by path $\rho$) 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 $\rho'$ 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 $\rho'$ 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 $q'_1$ 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 $\rho$ ended in $\alpha$ in the label graph, and since by construction this $\alpha$ 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 $\alpha$. Since they match the same path, we conclude that $q'_1 \sim_C q'_2$. □

## A.2 Soundness

**Theorem A.2** (Soundness)**.** *For all $C$, $q_1$, $q_2$, and $k$, if $q_1 \sim_C q_2$, then* $\text{RESOLVE}(q_1, C, k) \approx \text{RESOLVE}(q_2, C, k)$.

*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

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

**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:

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

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

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

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:

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

□

## A.3 Completeness

**Theorem A.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. □

## A.4 Efficiency

**Theorem A.4** (Linear time)**.** *In the absence of* `DNAME` *records, for a collection of zone files with $n$ resource records, my 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. □

[1920] ARTICLE 19. "Coronavirus: Access to the internet can be a matter of life and death during a pandemic." https://www.article19.org/resources/access-to-the-internet-can-be-a-matter-of-life-and-death-during-the-coronavirus-pandemic/, 2020.

[ABS20] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. "NXNSAttack: Recursive DNS Inefficiencies and Vulnerabilities." In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 631–648, Virtual, August 2020. USENIX Association.

[Aka20] Akamai. "Fast DNS." https://www.akamai.com/us/en/products/security/fast-dns.jsp, 2020.

[Ama10] Amazon. "Amazon Route 53." https://aws.amazon.com/route53/, 2010.

[AS19] Hamad Al Salem and Jia Song. "A review on grammar-based fuzzing techniques." *International Journal of Computer Science & Security (IJCSS)*, **13**(3):114–123, 2019.

[Aut22] IANA (Internet Assigned Numbers Authority). "Resource Record (RR) TYPEs." https://www.iana.org/assignments/dns-parameters/, 2022. [Online; accessed May-2022].

[Bar96] David Barr. "Common DNS Operational and Configuration Errors." RFC 1912, February 1996.

[Bar12] John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, London, GBR, 2012.

[BBC19] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. "Reachability Analysis for AWS-Based Networks." In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pp. 231–241, Cham, 2019. Springer International Publishing.

[BD87] Stanislaw Budkowski and Piotr Dembinski. "An introduction to Estelle: A specification language for distributed systems." *Computer Networks and ISDN systems*, **14**(1):3–23, 1987.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. "Reachability analysis of pushdown automata: Application to model-checking." In Antoni Mazurkiewicz

and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory*, pp. 135–150, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Bem05] David Abrahams Beman Dawes. "Boost C++ Libraries." https://www.boost.org/, 2005.

[Ben01] Bob Bentley. "Validating the Intel Pentium 4 Microprocessor." In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, p. 244–248, New York, NY, USA, 2001. Association for Computing Machinery.

[BFN05] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. "Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets." In Roch Guérin, Ramesh Govindan, and Greg Minshall, editors, *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, pp. 265–276. ACM, 2005.

[BG17] Petar D. Bojovic and Slavko Gajin. "An approach to evaluation of common DNS misconfigurations." *CoRR*, **abs/1711.05696**, 2017.

[BG22] Ryan Beckett and Aarti Gupta. "Katra: Realtime Verification for Multilayer Networks." In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 617–634, Renton, WA, April 2022. USENIX Association.

[BGM17] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. "A General Approach to Network Configuration Verification." In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pp. 155–168, New York, NY, USA, 2017. ACM.

[BH14] Bernhard Beckert and Reiner Hahnle. "Reasoning and verification: State of the art and current trends." *IEEE Intelligent Systems*, **29**(1):20–29, 2014.

[BH16] Stéphane Bortzmeyer and Shumon Huque. "NXDOMAIN: There Really Is Nothing Underneath." RFC 8020, November 2016.

[Bin22] "Bind GitLab Issues." https://gitlab.isc.org/isc-projects/bind9/-/issues. [Online; accessed May-2022], 2022.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. "Korat: Automated Testing Based on Java Predicates." In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, p. 123–133, New York, NY, USA, 2002. Association for Computing Machinery.

[BM20] Ryan Beckett and Ratul Mahajan. "A General Framework for Compositional Network Modeling." In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, p. 8–15, New York, NY, USA, 2020. Association for Computing Machinery.

[BMM18] Josip Bozic, Lina Marsso, Radu Mateescu, and Franz Wotawa. "A Formal TLS Handshake Model in LNT." In John P. Gallagher, Rob van Glabbeek, and Wendelin Serwe, editors, *Proceedings Third Workshop on Models for Formal Analysis of Real Systems and Sixth International Workshop on Verification and Program Transformation, Thessaloniki, Greece, 20th April 2018*, volume 268 of *Electronic Proceedings in Theoretical Computer Science*, pp. 1–40, Greece, 2018. Open Publishing Association.

[BOG02] Karthikeyan Bhargavan, Davor Obradovic, and Carl A Gunter. "Formal verification of standards for distance vector routing protocols." *Journal of the ACM (JACM)*, **49**(4):538–576, 2002.

[Bor16] Stéphane Bortzmeyer. "DNS Query Name Minimisation to Improve Privacy." RFC 7816, March 2016.

[BPR16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain." In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, p. 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[BR18] Jonas Bushart and Christian Rossow. "DNS Unchained: Amplified Application-Layer DoS Attacks Against DNS Authoritatives." In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pp. 139–160, Cham, 2018. Springer International Publishing.

[Bri95] Thomas P. Brisco. "DNS Support for Load Balancing." RFC 1794, April 1995.

[BS95] Olaf Burkart and Bernhard Steffen. "Composition, decomposition and model checking of pushdown processes." *Nordic Journal of Computing*, **2**(2):89–125, 1995.

[But01] R W Butler. "WHAT IS FORMAL METHODS?", 2001.

[BV22] PowerDNS.com BV. "PDNSUtil." https://doc.powerdns.com/authoritative/manpages/pdnsutil.1.html, 2022. [Online; accessed May-2022].

[Cam19] Frederic Cambus. "Fuzzing DNS zone parsers." https://www.cambus.net/fuzzing-dns-zone-parsers/, 2019.

[CC18] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search." In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, 2018.

[CCD13] Marco Chiesa, Luca Cittadini, Giuseppe Di Battista, Laurent Vanbever, and Stefano Vissicchio. "Using routers to build logic circuits: How powerful is BGP?" In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pp. 1–10. IEEE, 2013.

[CCH17] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. "What is Decidable about String Constraints with the ReplaceAll Function." *Proc. ACM Program. Lang.*, **2**(POPL), December 2017.

[CCP17] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. "Optimal Dyck reachability for data-dependence and alias analysis." *Proceedings of the ACM on Programming Languages*, **2**(POPL):1–30, 2017.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, p. 209–224, USA, 2008. USENIX Association.

[CF12] M. Carlisle and B. Fagin. "IRONSIDES: DNS with no single-packet denial of service or remote code execution vulnerabilities." In *2012 IEEE Global Communications Conference (GLOBECOM)*, pp. 839–844, Anaheim, CA, USA, 2012. IEE.

[Cha08] Swarat Chaudhuri. "Subcubic Algorithms for Recursive State Machines." In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, p. 159–169, New York, NY, USA, 2008. Association for Computing Machinery.

[CJV11] Marco Canini, Vojin Jovanović, Daniele Venzano, Dejan Novaković, and Dejan Kostić. "Online Testing of Federated and Heterogeneous Distributed Systems." In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, p. 434–435, New York, NY, USA, 2011. Association for Computing Machinery.

[CK13] Stuart Cheshire and Marc Krochmal. "DNS-Based Service Discovery." RFC 6763, February 2013.

[Cla21] Mitchell Clark. "Slack is down for some people, and of course, the problem is DNS." https://www.theverge.com/2021/9/30/22702876/slack-is-down-outage-morning-disruption-work-chat, 2021.

[CO17] Krishnendu Chatterjee and Georg Osang. "Pushdown reachability with constant treewidth." *Information Processing Letters*, **122**:25–29, 2017.

[Col98] Michael Collins. "Formal Methods.", 1998.

[Con86] Internet Systems Consortium. "BIND 9." https://www.isc.org/bind/, 1986. Code commit used: https://gitlab.isc.org/isc-projects/bind9/-/tree/dbcf683c1a57f49876e329fca183cb39d20ca3a4. [Online; accessed May-2022].

[Con02] Internet Systems Consortium. "named-checkzone(8)." https://linux.die.net/man/8/named-checkzone, 2002. [Online; accessed May-2022].

[CZ11] CZ.NIC. "Knot." https://www.knot-dns.cz/, 2011. Code commit used: https://gitlab.nic.cz/knot/knot-dns/-/tree/563fcdd886b5d5c52bceeb8fda3c4bda59ece73e. [Online; accessed May-2022].

[Dat21] National Vulnerability Database. "CVE-2021-25215 Detail." https://nvd.nist.gov/vuln/detail/CVE-2021-25215, 2021.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, p. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[Dij75] Edsger W Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs." *Communications of the ACM*, **18**(8):453–457, 1975.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. "Rewrite Systems.", 1990.

[DNS13] "DNS Census." https://dnscensus2013.neocities.org/, 2013.

[DNS22a] "DNS Response Policy Zones." https://dnsrpz.info/, 2022. [Online; accessed May-2022].

[DNS22b] "DNSBL information - spam database and blacklist check." https://www.dnsbl.info/, 2022. [Online; accessed May-2022].

[EB97] Robert Elz and Randy Bush. "Clarifications to the DNS Specification." RFC 2181, July 1997.

[Edd04] Michael Eddington. "Peach fuzzer platform." https://peachtech.gitlab.io/peach-fuzzer-community/, 2004.

[EHR00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. "Efficient algorithms for model checking pushdown systems." In *International Conference on Computer Aided Verification*, pp. 232–247. Springer, 2000.

[EUR12] EURid.eu. "YADIFA." https://www.yadifa.eu/, 2012. Code commit used: https://github.com/yadifa/yadifa/tree/dc5bed2fb8ec204af9b65eeb91934c2c85098cbb. [Online; accessed May-2022].

[FBD21] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. "Prognosis: Closed-Box Analysis of Network Protocol Implementations." In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, p. 762–774, New York, NY, USA, 2021. Association for Computing Machinery.

[FC15] Benjamin Fry and TrustDNS Community. "TRust-DNS." http://trust-dns.org/, 2015. Code commit used: https://github.com/bluejekyll/trust-dns/tree/7d9b186121fb5cb331cf2ec6baa47846b83de8fc. [Online; accessed May-2022].

[FFP15] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. "A General Approach to Network Configuration Analysis." In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 469–483, Oakland, CA, 2015. USENIX Association.

[Flo67] Robert W Floyd. "Assigning meanings to programs." *Mathematical aspects of computer science*, **19**(19-32):1, 1967.

[FM18] Limor Fix and Ken McMillan. "Formal Property Verification." In *EDA for IC System Design, Verification, and Testing*, pp. 20–1. CRC Press, 2018.

[Foo15] Jonathan Foote. "How to fuzz a server with American Fuzzy Lop." https://www.fastly.com/blog/how-fuzz-server-american-fuzzy-lop, 2015.

[For22] Internet Engineering Task Force. "The RFC Archive." https://www.rfc-archive.org/errata, 2022.

[Fri08] Steve Friedl. "An Illustrated Guide to the Kaminsky DNS Vulnerability." http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html, 2008.

[Fry14] James Fryman. "DNS Outage Post Mortem." https://github.blog/2014-01-18-dns-outage-post-mortem/, 2014.

[FSF16] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. "Efficient Network Reachability Analysis Using a Succinct Control Plane Representation." In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 217–232, Savannah, GA, 2016. USENIX Association.

[FYT16] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. "BUZZ: Testing Context-Dependent Policies in Stateful Networks." In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 275–289, Santa Clara, CA, March 2016. USENIX Association.

[Gc16] Miek Gieben and CoreDNS community. "CoreDNS." https://coredns.io/, 2016. Code commit used: https://github.com/coredns/coredns/tree/6edc8fe7f6c2f57844c8ee7f7f5deef71085ebe8. [Online; accessed May-2022].

[GCB16] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. "QuickFuzz: An Automatic Random Fuzzer for Common File Formats." In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, p. 13–20, New York, NY, USA, 2016. Association for Computing Machinery.

[GKD21] Suzanne Goldlust, Michał Kępień, Peter Davies, and Everett Fulton. "CVE-2021-25215: An assertion check can fail while answering queries for DNAME records that require the DNAME to be processed to resolve itself." https://kb.isc.org/v1/docs/cve-2021-25215, 2021.

[GKL08] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. "Grammar-Based Whitebox Fuzzing." In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, p. 206–215, New York, NY, USA, 2008. Association for Computing Machinery.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, p. 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[Gle18] Thomas Gleason. "DNS Based Lever - An Untapped DevOps tool." https://community.akamai.com/customers/s/article/DNS-Based-Lever-An-Untapped-DevOps-tool?language=en_US, 2018.

[GMS12] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. "Word Equations with Length Constraints: What's Decidable?" In *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing*, HVC'12, p. 209–226, Berlin, Heidelberg, 2012. Springer-Verlag.

[God97] Patrice Godefroid. "Model checking for programming languages using VeriSoft." In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 174–186, 1997.

[God20] Patrice Godefroid. "Fuzzing: Hack, art, and science." *Communications of the ACM*, **63**(2):70–76, 2020.

[Goo22] Google. "Cloud DNS." https://cloud.google.com/dns/, 2022. [Online; accessed May-2022].

[GSW02] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. "The stable paths problem and interdomain routing." *IEEE/ACM Transactions On Networking*, **10**(2):232–243, 2002.

[GVA16] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. "Fast Control Plane Analysis Using an Abstract Representation." In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pp. 300–313, New York, NY, USA, 2016. ACM.

[Hat12] Chris Hathhorn. "Engineering a Compiler, Second Edition by Keith D. Cooper and Linda Torczon." *SIGSOFT Softw. Eng. Notes*, **37**(1):36–37, January 2012.

[HC02] Bert Hubert and PowerDNS Community. "PowerDNS." https://www.powerdns.com/, 2002. Code commit used: https://github.com/PowerDNS/pdns/tree/a03aaad7554483ee6efe72a81eda00a9d1a94fe5. [Online; accessed May-2022].

[HC05] Bob Halley and Dnspython Community. "Dnspython." https://www.dnspython.org/, 2005.

[HM97] Nevin Heintze and David McAllester. "On the cubic bottleneck in subtyping and flow analysis." In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*, pp. 342–351. IEEE, 1997.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[Hoa69] Charles Antony Richard Hoare. "An axiomatic basis for computer programming." *Communications of the ACM*, **12**(10):576–580, 1969.

[Hoc07] Sam Hocevar. "zzuf: multi-purpose fuzzer." http://caca.zoy.org/wiki/zzuf/, 2007.

[Hos20] Check Host. "Check Host." http://check-host.net/check-dns, 2020.

[HSF15] Paul E. Hoffman, Andrew Sullivan, and Kazunori Fujiwara. "DNS Terminology." RFC 7719, December 2015.

[HSF19] Paul E. Hoffman, Andrew Sullivan, and Kazunori Fujiwara. "DNS Terminology." RFC 8499, January 2019.

[Hus20] Geoff Huston. "Scaling the root of the DNS." https://blog.apnic.net/2020/09/28/scaling-the-root-of-the-dns/, 2020.

[Inc19] Internet Initiative Japan Inc. "IP Location Load Balancing Resource Record." Internet-Draft draft-sonoda-dnsop-lb-01, Internet Engineering Task Force, March 2019. Work in Progress.

[Inc22] Dyn Inc. "Dynamic DNS." https://account.dyn.com/, 2022. [Online; accessed May-2022].

[Inf19] InfinityFree. "DNS Outage at iFastNet: Softaculous down." https://forum.infinityfree.net/t/dns-outage-at-ifastnet-softaculous-down/19374, 2019.

[Inf22] Infoblox. "Simplify DNS, DHCP and IPAM (DDI) everywhere." https://www.infoblox.com/products/ddi/, 2022. [Online; accessed May-2022].

[Jac02] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation." *ACM Trans. Softw. Eng. Methodol.*, **11**(2):256–290, April 2002.

[JBP19] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. "Validating Datacenters at Scale." In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, p. 200–213, New York, NY, USA, 2019. Association for Computing Machinery.

[KA21] Siva Kesava R Kakarla and Mark Andrews. "Glue records can be returned when the name server's name is same as the zone origin." https://gitlab.isc.org/isc-projects/bind9/-/issues/2385, 2021.

[KAK20] Siva Kesava R Kakarla, Mark Andrews, and Michał Kępień. "DNAME: synthetized CNAME might be perfect answer to CNAME query." https://gitlab.isc.org/isc-projects/bind9/-/issues/2284, 2020.

[KAK21a] Siva Kakarla, Mark Andrews, Michał Kępień, Peter Davies, and Michal Nowak. "[CVE-2021-25215] An assertion check can fail while answering queries for DNAME records that require the DNAME to be processed to resolve itself." https://gitlab.isc.org/isc-projects/bind9/-/issues/2540, 2021.

[KAK21b] Siva Kesava R Kakarla, Mark Andrews, and Michał Kępień. "Sibling (In-bailiwick rule of RFC 8499) domain IP records not returned." https://gitlab.isc.org/isc-projects/bind9/-/issues/2384, 2021.

[KBA20a] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. "GRoot: Proactive Verification of DNS Configurations." In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data*

*Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, p. 310–328, New York, NY, USA, 2020. Association for Computing Machinery.

[KBA20b] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. "Software Artifact for the SIGCOMM'20 Paper Titled "GRoot: Proactive Verification of DNS Configurations"." https://doi.org/10.5281/zenodo.3905968, June 2020.

[KBM21] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. "How Complex is DNS?" In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, p. 116–122, New York, NY, USA, 2021. Association for Computing Machinery.

[KBM22] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. "SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers." In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 307–323, Renton, WA, April 2022. USENIX Association.

[KCH11] Murray Kucherawy, Dave Crocker, and Tony Hansen. "DomainKeys Identified Mail (DKIM) Signatures." RFC 6376, September 2011.

[KD20a] Siva Kesava R Kakarla and Peter van Dijk. "CNAME need not be followed after a synthesized CNAME for a CNAME query." https://github.com/PowerDNS/pdns/issues/9886, 2020.

[KD20b] Siva Kesava R Kakarla and Peter van Dijk. "pdnsutil DNAME checks have issues." https://github.com/PowerDNS/pdns/issues/9734, 2020.

[KEH09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "SeL4: Formal Verification of an OS Kernel." In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, p. 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[Kep20] Nick Kephart. "Best Practices for Monitoring DNS." https://www.thousandeyes.com/resources/dns-webinar, 2020.

[KF20a] Siva Kesava R Kakarla and Benjamin Fry. "CNAME loops throws off the server." https://github.com/bluejekyll/trust-dns/issues/1283, 2020.

[KF20b] Siva Kesava R Kakarla and Benjamin Fry. "Zone cut NS RRs returned as authoritative records." https://github.com/bluejekyll/trust-dns/issues/1273, 2020.

[KF21] Siva Kesava R Kakarla and Benjamin Fry. "Wildcards match only one label." https://github.com/bluejekyll/trust-dns/issues/1342, 2021.

[KFB20] Siva Kesava R Kakarla, Benjamin Fry, and Jonas Bushart. "Glue records returned as authoritative records by the server ." https://github.com/bluejekyll/trust-dns/issues/1272, 2020.

[KG99] Christoph Kern and Mark R Greenstreet. "Formal verification in hardware design: a survey." *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, **4**(2):123–193, 1999.

[KG21a] Siva Kesava R Kakarla and Miek Gieben. "Handling wildcard CNAME loops." https://github.com/coredns/coredns/issues/4378, 2021.

[KG21b] Siva Kesava R Kakarla and Miek Gieben. "NXDOMAIN returned when the domain exists." https://github.com/coredns/coredns/issues/4374, 2021.

[KG21c] Siva Kesava R Kakarla and Miek Gieben. "Sibling (In-bailiwick rule of RFC 8499) domain IP records can also be returned along with NS records." https://github.com/coredns/coredns/issues/4377, 2021.

[KHB15] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. "Going Wild: Large-Scale Classification of Open DNS Resolvers." In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, p. 355–368, New York, NY, USA, 2015. Association for Computing Machinery.

[KHT03] Vladimir Ksinant, Christian Huitema, Dr. Susan Thomson, and Mohsen Souissi. "DNS Extensions to Support IP Version 6." RFC 3596, October 2003.

[Kin76] James C. King. "Symbolic Execution and Program Testing." *Commun. ACM*, **19**(7):385–394, July 1976.

[Kit14] Scott Kitterman. "Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1." RFC 7208, April 2014.

[KKM99] Eddie Kohler, M Frans Kaashoek, and David R Montgomery. "A readable TCP in the Prolac protocol language." In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 3–13, 1999.

[KO20] Siva Kesava R Kakarla and Chris O'Haver. "Non-existent CNAME target in the same zone should be returned with NXDOMAIN instead of NOERROR rcode." https://github.com/coredns/coredns/issues/4288, 2020.

[Kov18] Eduard Kovacs. "DNS Servers Crash Due to BIND Security Flaw." https://www.securityweek.com/dns-servers-crash-due-bind-security-flaw, 2018.

[KOY20] Siva Kesava R Kakarla, Chris O'Haver, and Kohei Yoshida. "Return code for synthesized CNAME records (from wildcards and DNAMEs)." https://github.com/coredns/coredns/issues/4341, 2020.

[KOY21] Siva Kesava R Kakarla, Chris O'Haver, and Kohei Yoshida. "CNAME need not be followed after a synthesized CNAME for a CNAME query." https://github.com/coredns/coredns/issues/4398, 2021.

[KP20] Ella Koeze and Nathaniel Popper. "The Virus Changed the Way We Internet." https://www.nytimes.com/interactive/2020/04/07/technology/coronavirus-internet-use.html, 2020.

[KPS20] Siva Kesava R Kakarla, Libor Peltan, Daniel Salzman, and mscbg. "DNAME-DNAME loop test case is not a loop." https://gitlab.nic.cz/knot/knot-dns/-/issues/703, 2020.

[KPS21a] Siva Kesava R Kakarla, Libor Peltan, and Daniel Salzman. "Record incorrectly synthesized from wildcard record." https://gitlab.nic.cz/knot/knot-dns/-/issues/715, 2021.

[KPS21b] Siva Kesava R Kakarla, Libor Peltan, and Daniel Salzman. "Records below delegation are not ignored (kzonecheck also does not raise any issue)." https://gitlab.nic.cz/knot/knot-dns/-/issues/713, 2021.

[KPS21c] Siva Kesava R Kakarla, Libor Peltan, Daniel Salzman, and Vladimír Čunát. "DNAME not applied more than once to resolve the query." https://gitlab.nic.cz/knot/knot-dns/-/issues/714, 2021.

[KVM12] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks." In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pp. 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[KW20] Siva Kesava R Kakarla and Wouter Wijngaards. "DNAME: synthesized CNAME might be perfect answer to CNAME query." https://github.com/NLnetLabs/nsd/issues/140, 2020.

[KW21a] Siva Kesava R Kakarla and Wouter Wijngaards. "'*' in Rdata causes the return code to be NOERROR instead of NX." https://github.com/NLnetLabs/nsd/issues/152, 2021.

[KW21b] Siva Kesava R Kakarla and Wouter Wijngaards. "DNAME not applied more than once to resolve the query." https://github.com/NLnetLabs/nsd/issues/151, 2021.

[KW21c] Siva Kesava R Kakarla and Wouter Wijngaards. "NS Records below delegation are not ignored (nsd-checkzone also does not raise any issue)." https://github.com/NLnetLabs/nsd/issues/174, 2021.

[Ky21] Siva Kesava R Kakarla and yadifa. "Records below delegation are not ignored." https://github.com/yadifa/yadifa/issues/12, 2021.

[Kye20a] Siva Kesava R Kakarla, yadifa, and edfeu. "Non-existent CNAME target in the same zone should be returned with NXDOMAIN instead of NOERROR." https://github.com/yadifa/yadifa/issues/11, 2020.

[Kye20b] Siva Kesava R Kakarla, yadifa, and edfeu. "Why are CNAME chains not followed?" https://github.com/yadifa/yadifa/issues/10, 2020.

[KZC12] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. "Veriflow: Verifying Network-wide Invariants in Real Time." *SIGCOMM Comput. Commun. Rev.*, **42**(4):467–472, September 2012.

[Lab02a] NLnet Labs. "NSD." https://nlnetlabs.nl/projects/nsd/about/, 2002. Code commit used: https://github.com/NLnetLabs/nsd/tree/4043a5ab7be7abaec969011e48e4d0d60a0056a6 [Online; accessed May-2022].

[Lab02b] NLnet Labs. "nsd-checkzone - NSD zone file syntax checker." https://nsd.docs.nlnetlabs.nl/en/latest/manpages/nsd-checkzone.html, 2002. [Online; accessed May-2022].

[Lab22] CZ.NIC Labs. "kzonecheck – Knot DNS zone file checking tool." https://www.knot-dns.cz/docs/2.5/html/man_kzonecheck.html, 2022. [Online; accessed May-2022].

[LBG15] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. "Checking Beliefs in Dynamic Networks." In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 499–512, Oakland, CA, 2015. USENIX Association.

[Let04] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris Sud, 2004.

[Lew06] Edward P. Lewis. "The Role of Wildcards in the Domain Name System." RFC 4592, July 2006.

[LS18] Caroline Lemieux and Koushik Sen. *FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage*, p. 475–485. Association for Computing Machinery, New York, NY, USA, 2018.

[LSF15] Hyojeong Lee, Jeff Seibert, Dylan Fistrovic, Charles Killian, and Cristina Nita-Rotaru. "Gatling: Automatic Performance Attack Discovery in Large-Scale Distributed Systems." *ACM Trans. Inf. Syst. Secur.*, **17**(4), April 2015.

[MBJ20] Stephen McQuistin, Vivian Band, Dejice Jacob, and Colin Perkins. "Parsing Protocol Standards to Parse Standard Protocols." In *Proceedings of the Applied Networking Research Workshop*, ANRW '20, p. 25–31, New York, NY, USA, 2020. Association for Computing Machinery.

[MCH21] Giovane C. M. Moura, Sebastian Castro, John Heidemann, and Wes Hardaker. "TsuNAME: Exploiting Misconfiguration and Vulnerability to DDoS DNS." In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, p. 398–418, New York, NY, USA, 2021. Association for Computing Machinery.

[Mea02] Michael H. Mealling. "Dynamic Delegation Discovery System (DDDS) Part Three: The Domain Name System (DNS) Database." RFC 3403, October 2002.

[Mer14] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." *Linux J.*, **2014**(239):2, March 2014.

[Mic18] Microsoft. "Description of the DNSLint utility." https://support.microsoft.com/en-us/help/321045/description-of-the-dnslint-utility, 2018.

[Mic20] Microsoft. "Azure DNS." https://azure.microsoft.com/en-us/services/dns/, 2020.

[Mic22] Microsoft. "Microsoft DNS." https://docs.microsoft.com/en-us/windows-server/networking/dns/dns-top, 2022. [Online; accessed May-2022].

[MKA11] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. "Debugging the Data Plane with Anteater." In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pp. 290–301, New York, NY, USA, 2011. ACM.

[Moc87a] P. Mockapetris. "Domain names - concepts and facilities." RFC 1034, November 1987.

[Moc87b] Paul Mockapetris. "Domain names - implementation and specification." RFC 1035, November 1987.

[MZ19a] Kenneth L. McMillan and Lenore D. Zuck. "Compositional Testing of Internet Protocols." In *2019 IEEE Cybersecurity Development (SecDev)*, pp. 161–174, 2019.

[MZ19b] Kenneth L. McMillan and Lenore D. Zuck. "Formal Specification and Testing of QUIC." In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, p. 227–240, New York, NY, USA, 2019. Association for Computing Machinery.

[New10] SecurityWeek News. "Reports of Massive DNS Outages in Germany." https://www.securityweek.com/content/reports-massive-dns-outages-germany, 2010.

[npm18] Incident Report for npm. "DNS misconfiguration cached in ISP DNS caches." https://status.npmjs.org/incidents/v22ffls5cd6h, 2018.

[NR95] B. Neelakantan and S. V. Raghavan. *Protocol Conformance Testing — A Survey*, pp. 175–191. Springer US, Boston, MA, 1995.

[NSD22] "NSD GitHub Issues." https://github.com/NLnetLabs/nsd/issues, 2022. [Online; accessed May-2022].

[PA09] Javier Paris and Thomas Arts. "Automatic Testing of TCP/IP Implementations Using QuickCheck." In *Proceedings of the 8th ACM SIGPLAN Workshop on ERLANG*, ERLANG '09, p. 83–92, New York, NY, USA, 2009. Association for Computing Machinery.

[Pat22] Michael Pattrick. "DNS-fuzz." https://nmap.org/nsedoc/scripts/dns-fuzz.html, 2022. [Online; accessed May-2022].

[Pel20] Libor Peltans. "Nsd and Knot discussion." https://github.com/NLnetLabs/nsd/issues/142#issuecomment-732753256, 2020.

[PFM04] Vasileios Pappas, Patrik Fältström, Daniel Massey, and Lixia Zhang. "Distributed DNS troubleshooting." In *Proceedings of the ACM SIGCOMM workshop on Network troubleshooting: research, theory and operations practice meet malfunctioning reality*, pp. 265–270, 2004.

[Pom17] Raymond Pompon. "DNS Is Still the Achilles' Heel of the Internet." https://www.f5.com/labs/articles/threat-intelligence/dns-is-still-the-achilles-heel-of-the-internet-25613, 2017.

[Pos81] John Postel. "Internet Control Message Protocol." RFC 792, September 1981.

[Pow22] "PowerDNS GitHub Issues." https://github.com/PowerDNS/pdns/issues?q=is%3Aissue+is%3Aopen+label%3Aauth, 2022. [Online; accessed May-2022].

[PS20] Libor Peltan and Daniel Salzman. "DNAME: synthesized CNAME might be perfect answer to CNAME query." https://gitlab.nic.cz/knot/knot-dns/-/merge_requests/1217, 2020.

[PWH08] Tom Preston-Werner, Chris Wanstrath, P. J. Hyett, and Scott Chacon. "GitHub, Inc." https://github.com/, 2008.

[Raf21] Laura Nolan Rafael Elvira. "The Case of the Recursive Resolvers." https://slack.engineering/what-happened-during-slacks-dnssec-rollout/, 2021.

[Ras16] Fahmida Y. Rashid. "ISC updates critical DoS bug in BIND DNS software." https://www.infoworld.com/article/3126472/isc-updates-critical-dos-bug-in-bind-dns-software.html, 2016.

[RBC16] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. "Model Finding for Recursive Functions in SMT." In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning*, pp. 133–151, Cham, 2016. Springer International Publishing.

[RE15] David A. Ramos and Dawson Engler. "Under-Constrained Symbolic Execution: Correctness Checking for Real Code." In *24th USENIX Security Symposium (USENIX Security 15)*, pp. 49–64, Washington, D.C., August 2015. USENIX Association.

[RHL06] Yakov Rekhter, Susan Hares, and Tony Li. "A Border Gateway Protocol 4 (BGP-4)." RFC 4271, January 2006.

[RLK07] Thomas Reps, Akash Lal, and Nick Kidd. "Program Analysis Using Weighted Pushdown Systems." In V. Arvind and Sanjiva Prasad, editors, *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, pp. 23–51, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[RS59] M. O. Rabin and D. Scott. "Finite Automata and Their Decision Problems." *IBM Journal of Research and Development*, **3**(2):114–125, 1959.

[RSJ05] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. "Weighted pushdown systems and their application to interprocedural dataflow analysis." *Science of Computer Programming*, **58**(1-2):206–263, 2005.

[RW12a] Scott Rose and Wouter Wijngaards. "DNAME Redirection in the DNS." RFC 6672, June 2012.

[RW12b] Scott Rose and Wouter Wijngaards. "DNAME Redirection in the DNS." RFC 6672, June 2012.

[SBK20] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K. Sitaraman. "Akamai DNS: Providing Authoritative Answers to the World's Queries." In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, p. 465–478, New York, NY, USA, 2020. Association for Computing Machinery.

[Sch01] Johann M Schumann. "Formal Methods in Software Engineering." In *Automated Theorem Proving in Software Engineering*, pp. 11–22. Springer, 2001.

[Sch02] Stefan Schwoon. *Model-checking pushdown systems.* PhD thesis, Technische Universität München, 2002.

[SCK06] Ao-Jan Su, David R Choffnes, Aleksandar Kuzmanovic, and Fabián E Bustamante. "Drafting behind Akamai (travelocity-based detouring)." *ACM SIGCOMM Computer Communication Review*, **36**(4):435–446, 2006.

[SCP14] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. "SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications." *IEEE Trans. Softw. Eng.*, **40**(7):695–709, July 2014.

[SJR20] Raffaele Sommese, Mattijs Jonker, Roland van Rijswijk-Deij, Alberto Dainotti, K.C. Claffy, and Anna Sperotto. "The Forgotten Side of DNS: Orphan and Abandoned Records." In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pp. 538–543, 2020.

[SMJ20] Raffaele Sommese, Giovane C. M. Moura, Mattijs Jonker, Roland van Rijswijk-Deij, Alberto Dainotti, K. C. Claffy, and Anna Sperotto. "When Parents and Children Disagree: Diving into DNS Delegation Inconsistency." In Anna Sperotto, Alberto Dainotti, and Burkhard Stiller, editors, *Passive and Active Measurement*, pp. 175–189, Cham, 2020. Springer International Publishing.

[Spe21] Richard Speed. "That Salesforce outage: Global DNS downfall started by one engineer trying a quick fix." https://www.theregister.com/2021/05/19/salesforce_root_cause/, 2021.

[SPN16] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. "SymNet: Scalable Symbolic Execution for Modern Networks." In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, p. 314–327, New York, NY, USA, 2016. Association for Computing Machinery.

[St10] Robert Swiecki and *et al.* "Honggfuzz - Security oriented software fuzzer." https://github.com/google/honggfuzz/tree/master/examples/bind, 2010. [Online; accessed May-2022].

[Sta21] The Stack. "Slack outage: "It's always DNS" (and, well...)." https://thestack.technology/slack-outage-dns, 2021.

[sta22] statdns. "DNS and Domain Name statistics and tools." https://www.statdns.com/rfc/, 2022. [Online; accessed May-2022].

[SWD20] Harsha Sharma, Wenfei Wu, and Bangwen Deng. "Symbolic Execution for Network Functions with Time-Driven Logic." In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, 2020.

[SZ14] Sytse "Sid" Sijbrandij and Dmytro Zaporozhets. "GitLab, Inc." https://gitlab.com/, 2014.

[Tho68] Ken Thompson. "Programming Techniques: Regular Expression Search Algorithm." *Commun. ACM*, **11**(6):419–422, June 1968.

[Tre01] Sam Trenholme. "MaraDNS." https://maradns.samiam.org/, 2001. Code commit used: https://github.com/samboy/MaraDNS/tree/3ec477f227b2bf6947be8fbe8fd0ab73130227d0. [Online; accessed May-2022].

[Tun19] Liam Tung. "Azure global outage: Our DNS update mangled domain records, says Microsoft." https://www.zdnet.com/article/azure-global-outage-our-dns-update-mangled-domain-records-says-microsoft/, 2019.

[VCG08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, pp. 39–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[WH97] Russ Wright and Martin Hamilton. "Use of DNS Aliases for Network Services." RFC 2219, October 1997.

[YL16] Hongkun Yang and Simon S. Lam. "Real-time Verification of Network Properties Using Atomic Predicates." *IEEE/ACM Trans. Netw.*, **24**(2):887–900, April 2016.

[YL17] Hongkun Yang and Simon S Lam. "Scalable verification of networks with packet transformers using atomic predicates." *IEEE/ACM Transactions on Networking*, **25**(5):2900–2915, 2017.

[YLY21] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. "Semi-Automated Protocol Disambiguation and Code Generation." In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, p. 272–286, New York, NY, USA, 2021. Association for Computing Machinery.

[Yor15] Dan York. "HBO NOW DNSSEC Misconfiguration Makes Site Unavailable From Comcast Networks (Fixed Now)." https://www.internetsociety.org/blog/2015/03/hbo-now-dnssec-misconfiguration-makes-site-unavailable-from-comcast-networks-fixed-now/, 2015.

[Zal13] Michał Zalewski. "American Fuzzing Lop (AFL)." https://lcamtuf.coredump.cx/afl/, 2013.

[ZBW07] Bojan Zdrnja, Nevil Brownlee, and Duane Wessels. "Passive monitoring of DNS anomalies." In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 129–139. Springer, 2007.

[Zel13] Help Net Security Zeljka Zorz, Managing Editor. "LinkedIn outage was due to DNS records misconfiguration." https://www.helpnetsecurity.com/2013/06/21/linkedin-outage-was-due-to-dns-records-misconfiguration/, 2013.

[ZLY20] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. "APKeep: Realtime Verification for Real Networks." In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 241–255, Santa Clara, CA, February 2020. USENIX Association.