Rethinking DNS Configuration Verification with a Distributed Architecture

Yao Wang[†], Kexin Yu[†], Ziyi Wang[†], Kaiqiang Hu[★], Haizhou Du[★], Qiao Xiang[†], Xing Fang[†], Geng Li[△], Ruiting Zhou[◇], Linghe Kong[‡], Jiwu Shu[▲†]

[†]Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen University, [▲]Minjiang University, [★]Shanghai University of Electric Power, [△]Huawei, [◇]Southeast University, [‡]Shanghai Jiao Tong University

ABSTRACT

DNS misconfiguration can result in severe social and financial consequences. Existing DNS configuration verification tools employ a centralized architecture, where all zone files are collected for verification. This architecture faces significant scalability issues (*e.g.*, the verifier becoming the performance bottleneck and not supporting incremental verification). Inspired by the recent proposal of distributed data plane verification and the resemblance between the network data plane and DNS configuration, we propose to rearchitect DNS configuration verification with a distributed design. Our key insight is that by analyzing the query processing behavior of each DNS zone file in parallel and stitching the results in a symbolic way, we can substantially scale up the verification of DNS configuration. Evaluation shows that an up to 9.51× speed up on a dataset with over 410,000 resource records while having small overhead.

CCS CONCEPTS

• **Networks** → **Network reliability**;

KEYWORDS

DNS configuration verification, Distributed verification

ACM Reference Format:

Yao Wang, Kexin Yu, Ziyi Wang, Kaiqiang Hu, Haizhou Du, Qiao Xiang, Xing Fang, Geng Li, Ruiting Zhou, Linghe Kong, and Jiwu Shu. 2024. Rethinking DNS Configuration Verification with a Distributed Architecture. In *The 8th Asia-Pacific Workshop on Networking (APNet 2024), August 3–4, 2024, Sydney, Australia.* ACM, Sydney, Australia, 7 pages. https://doi.org/10.1145/3663408.3663412

1 INTRODUCTION

Over the past three decades since the initial publication of two RFCs [19, 20] defining the basic Domain Name System (DNS), the DNS system has evolved into a crucial component of the internet. As one of the largest distributed hierarchical database systems, its immense scale and complexity pose significant challenges for configuration [21]. Any failure in the DNS could lead to incalculable losses [3, 6, 7, 13, 26, 27].

The research related to DNS verification has been persistently underway for a significant duration of time. Early efforts mainly focused on probing [10, 31] and static analysis of single files [4, 14, 29]. In recent years, with the application of formal methods in network verification [1, 17, 18, 24, 28, 30], tools like GRoot [15] began to focus on formally defining the DNS system by validating a set of zone files. It scrutinizes DNS zone files against specified properties, verifying whether these properties hold for all potential DNS queries or providing a counterexample if they do not.

Despite the considerable progress GRoot made in DNS verification, it still suffers from two major issues — scalability and data privacy security. First, it shows that when the number of resource records reaches over ten million, the entire verification process takes about 10,000 seconds, according to the GRoot [15]. Second, GRoot needs to collect the zone files from various DNS servers to a trusted centralized server. This is hard for a decentralized system such as DNS.

In this paper, we systematically tackle the important problem of scaling DNS verification to be applicable in a large DNS ecosystem. Not only can a scalable DNS verification tool quickly find faults in a large-scale DNS ecosystem, but it can also shed light on the way forward for new research topics, such as the diagnosis and repair of DNS configurations.

Proposal: distributed analysis of the behavior of individual zone files and stitching the result in a symbolic way. We propose to rethink DNS configuration verification with a distributed architecture. To be specific, we let each DNS authoritative server focus on its own managed zone files. Furthermore, we individually conduct behavioral analysis on each zone file of a server to derive equivalence classes (ECs). We call these ECs as the local equivalence class (LEC). At the same time, we adopt the symbolic execution technique, which covers all possible query scenarios from top to bottom. With such a distributed architecture, we offload the verification from the centralized servers in GRoot to individual DNS servers, substantially improving scalability. Additionally, it allows DNS servers from different operators to collaborate and verify their configurations without exposing their own configurations.

Evaluation. We implement the prototype of Octopus and evaluate it with two datasets. One of the datasets is a real world open-sourced [22] dataset, and another is collected from a university laboratory. We compared Octopus with the centralized architecture GRoot. Meanwhile, we set up a testbed experiment with 7 servers. We choose 6 servers as the distributed DNS configuration verification platform and 1 server for centralized. For the university data, We evaluate the time of DNS configuration verification. The result

Haizhou Du and Qiao Xiang are co-corresponding authors.

APNet 2024, August 3–4, 2024, Sydney, Australia

ACM ISBN 979-8-4007-1758-1/24/08…$15.00
https://doi.org/10.1145/3663408.3663412

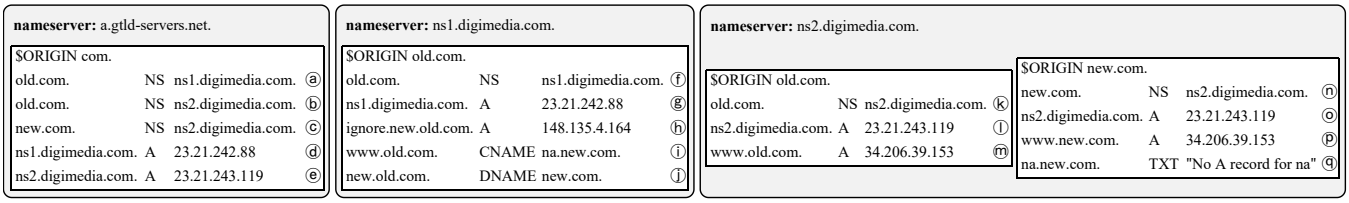


Figure 1: Example zone files for three nameservers: a.gtld-servers.net, ns1.digimedia.com, and ns2.digimedia.com. Suppose the company is converting its domain name from old.com to new.com.

shows Octopus can speed up 9.51× than the baseline in verification time. For the open-sourced dataset, we evaluate the relationship between the LECs number and its generating time to explain the reason for the smaller verification time. Octopus can generate less LECs than the baseline at the same time, which traverses smaller data structures to achieve the DNS configuration verification. The result shows the number of GRoot LECs can be up to 21.67× than Octopus with 99% data.

2 BACKGROUND

Preliminaries. Domain Name System (DNS) is a distributed naming system used on the Internet to translate domain names into corresponding IP addresses. It acts as the "phone book" of the Internet, converting user-friendly domain names into IP addresses used for network communication. Users send queries to DNS servers in the format $q : \langle d, t \rangle$, where d represents the fully qualified domain name (FQDN, e.g., *www.google.com*) and t indicates the record type format, to query the desired data. DNS servers store data in the form of resource records, and these files are known as zone files. Each resource record can be seen as a quintuple $\langle d, t, c, \tau, v \rangle$, where d and t have the same meanings as in the query, c represents the network type, τ is the time-to-live (TTL) of the record, and v contains specific data that is of interest to users [19].

As a globally used "phone book", the importance of DNS is undeniable. Therefore, ensuring the correctness of DNS configuration and maintenance poses significant challenges. Figure 1 illustrates an example where a company is transitioning its domain name server from *old.com* to *new.com*. However, due to a lack of synchronization in the configuration of just two authoritative servers, a query for *www.old.com* results in two different outcomes: record ⓜ and a query failure caused by forwarding to record ⓘ.

Equivalence Class. To verify DNS configurations, an intuitive approach is to iterate through all possible queries and judge based on the results obtained from executing the queries. However, the domain name can be divided into multiple labels separated by dots. For example, *www.google.com* consists of three labels: "*www*", "*google*", and "*com*". The length of each label can range from 0 to 63 octets. This means that a single label alone can have 2^{504} possible combinations, which is nearly impossible to enumerate. Therefore, the concepts of equivalence classes and symbolic execution [1] have been proposed and utilized to reduce the number of necessary queries while ensuring coverage. Essentially, an equivalence class is a set of queries that exhibit consistent behavior.

Nevertheless, the equivalence classes used in the current works are mainly computed based on the label graph proposed by GRoot. The main problem with this approach is that the number of equivalence classes computed is still too large. Figure 2(c) shows that at least 18 equivalence classes are generated for just 5 pieces of data, but only 7 different behaviors (our method) are possible in total. For example, the behavior of equivalence classes 3 and 9 in the figure is consistent, both returning NXDOMAIN to indicate that the query does not exist. The main reason for this problem is that GRoot computes the equivalence classes by considering the DNS system as a whole, which makes it difficult to directly obtain the behavior corresponding to a query, and the equivalence classes can only be generated by enumerating the paths on the label graph. That is, the set of equivalence classes given by this method is not the smallest set of equivalence classes. Therefore, the scalability problem becomes very challenging as the amount of data increases.

In order to solve the above problem, we would like to analyze the DNS individually by splitting it into zone files. Find all possible behaviors and construct equivalence classes to reduce the number of LECs.

3 DESIGN

In this section, we will introduce two key designs of Octopus: the action trie and property verification. The action trie reduces the amount of data the server needs to process, while property verification gives different verification schemes based on two different types of misconfiguration.

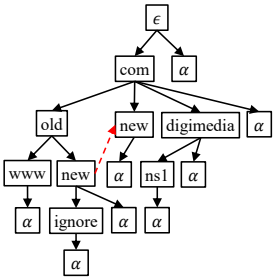
3.1 Local Equivalence Classes

Inspired by previous work [15, 28, 32], we first define local equivalence classes (LECs) for DNS query. It makes it easier to analyze all possible behaviors of a single server by focusing on its single zone file.

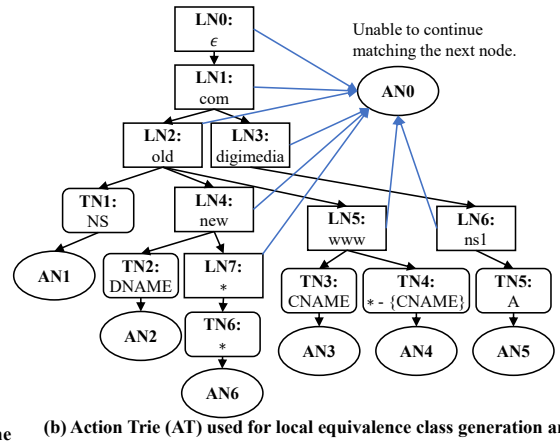
Definition 1 (Action). *We say* $Action_z(q) = \langle x, \{\langle v \rangle \,|\langle d, t, c, \tau, v \rangle \in RelR(q)\}\rangle$ *denotes the behavior generated by zone z for q, where* $RelR(q)$ *is the records associated with q and x is the type of DNS answer.*

Similarly to GRoot, we represent the type of DNS answer using $x \in \{Ans, AnsQ, Ref, NX, Refused, ServFail\}$ in definition 1. This definition states that an action is a combination of the type of answer returned by a DNS query and the value of the query result. Further, *Action* can reflect whether that DNS authoritative server chooses to return results, forward to a subdomain server, or some other behavior after receiving a query.

Then, for a domain z, there will necessarily be some queries for which the corresponding $Action_z$ is the same. We group the queries



(a) Label Graph (LG) used for local equivalence class generation for the zone file in nameserver ns1.digimedia.com. from Figure 1.



(b) Action Trie (AT) used for local equivalence class generation and symbolic Query for the zone file in nameserver ns1.digimedia.com. from Figure 1.

Action Node	LEC	Description
AN1	⟨old.com., NS⟩	Return value ns1.digimedia.com.
AN2	⟨new.old.com., DNAME⟩	Return value new.com.
AN3	⟨www.old.com., CNAME⟩	Return value na.new.com.
AN4	⟨www.old.com., * - {CNAME}⟩	Change the domain name of the query to na.new.com and forward it.
AN5	⟨ns1.digimedia.com., A⟩	Return value 23.21.242.88.
AN6	⟨*.new.old.com., *⟩	Replace new.old.com. with new.com. in the query domain and forward it.
AN0	other queries	Return NXDOMAIN.

(d) Local equivalence classes and behavioral descriptions corresponding to each Action Node in Figure 2(a).

1	⟨ε, t⟩
2	⟨α, t⟩
3	⟨com., t⟩
4	⟨α.com., t⟩
5	⟨old.com., t⟩
6	⟨α.old.com., t⟩
7	⟨new.com., t⟩
8	⟨α.new.com., t⟩
9	⟨digimedia.com., t⟩
10	⟨α.digimedia.com., t⟩
11	⟨www.old.com., t⟩
12	⟨α.www.old.com., t⟩
13	⟨new.old.com., t⟩
14	⟨α.new.old.com., t⟩
15	⟨ns1.digimedia.com., t⟩
16	⟨α.ns1.digimedia.com., t⟩
17	⟨ignore.new.old.com., t⟩
18	⟨α.ignore.new.old.com., t⟩

(c) Local equivalence classes generated using the Label Graph in Figure 2(a), one for each query type.

Nameserver	Zone	Number of LECs (AT)	Number of LECs (LG)
a.gtld-servers.net.	com.	7	14 * **T**
ns1.digimedia.com.	old.com.	7	18 * **T**
ns2.digimedia.com.	old.com.	4	12 * **T**
ns2.digimedia.com.	new.com.	5	14 * **T**

(e) Similarly, we can derive local equivalence classes for the other zone files in Figure 1. Here we give a comparison of the number of equivalence classes generated by the two methods AT and LG. T represents the number of query types. Note that even using the notion of strong equivalence classes in GRoot, T is still greater than or equal to 1.

Figure 2: Comparison of two data structures, Action Trie (AT) and Label Graph (LG). Note that the action trie is used not only for generating local equivalence classes but more importantly for processing queries during symbolic execution. In addition, $*$ denotes the set of all label sequences or the set of all query types, and α denotes an arbitrary sequence of labels.

with the same $Action_z$ into one class i.e., the equivalence class (EC). We denote EC as $c = \{q1, q2, ..., q_n\}$

Definition 2 (LECs). *We say $C = \{c_1, c_2, ..., c_n\}$ is a set of local equivalence classes (LECs) concerning a specified zone z if:*

1) $\forall i, j : i \neq j \implies c_i \cap c_j = \emptyset$.

2) $\forall c \in LECs : \forall q_1, q_2 \in c : q_1 \neq q_2 \implies Action_z(q_1) = Action_z(q_2)$.

3) $\{q \mid q \in c \wedge c \in LECs\} = \langle *, * \rangle$.

Additionally, if it also meets:

4) $\forall c_1, c_2 \in LECs, q_1 \in c_1, q_2 \in c_2 : c_1 \neq c_2 \implies Action_z(q_1) \neq Action_z(q_2)$.

We say C is the minimum set of LECs.

The above definition shows that for *LECs* of a domain z, 1) a query can only belong to one local equivalence class, and 2) the behavior of all queries in the same local equivalence class must be consistent. Additionally, 3) *LECs* must contain all possible queries. The first $*$ in $\langle *, * \rangle$ denotes the set consisting of all domain names and the second one denotes all types of queries. Minimization also requires that the behavior of two local equivalence classes must not be consistent.

Action Trie: An action trie provides a representation of local equivalence classes. We write $q.d$ and $q.t$ for the values of the domain name and type of a query q.

Definition 3 (Action Trie). *An action trie (AT) is a rooted tree that consists of three types of nodes:* **LN** (*label node*), **TN** (*type node*), *and* **AN** (*action node*).

1) *A* **LN** *contains a label set used to match the domain name field of a query q. If q.d is not fully matched, it will attempt to continue matching with the next* **LN**. *Otherwise, it will try to match the* **TN** *using the q.t. If neither matches, the query is dropped, or in other words, considered a "non-existent domain".*

2) *A* **TN** *contains a type set used to match a q.t. It has one and only one* **AN** *as its child node. When q is matched to a* **TN**, *it automatically enters the corresponding* **AN**.

3) *An* **AN** *directs what to do with a matched query q. It may give the value corresponding to q, or forward the query q with modifications.*

Action trie construction. As a first step in building an action trie for a zone, we read the zone file and insert each record as a path into the action trie. Consider again the zone file at a.gtld-servers.net. from Figure 1. The corresponding action trie is shown in Figure 2(b).

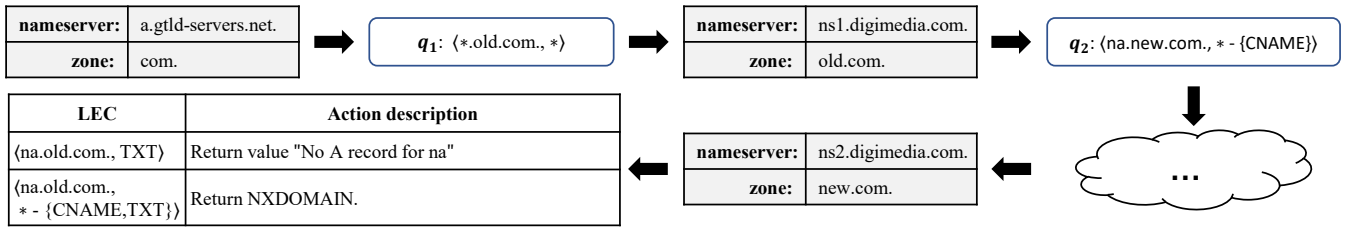


Figure 3: A query path with server a.gtld-servers.net. in Figure 1 as the starting point. The result of this path contains a Non-Existent Domain fault.

You can always find a path in action trie that contains any of the records in the zone file. For example, record ⓖ corresponds to the path **LN0** → **LN1** → **LN3** → **LN6** → **TN5** → **AN5**. In the second part, there are some special circumstances to consider. For instance, for a DNAME record ⓙ, in addition to the original path from **LN0** to **AN2**, an additional path from **LN0** to **AN6** needs to be added. This is necessary in case the query ⟨*.new.old.com., *⟩ is modified to ⟨*.new.com., *⟩ and forwarded. Similarly, CNAME, NS, and other records have similar situations, but they will not be expanded here due to space constraints. In the last step, we need to connect each **LN** to **AN0** to indicate a failed match. Also, merge the LNs with the same children.

As you may have noticed, the path corresponding to record ⓗ is also **LN0** to **AN6**. This means that the query ⟨ignore.new .old.com., *⟩ is rewritten and forwarded instead of returning the value 148.135.4.164. This is because all records in the subdomain corresponding to the DNAME record are ignored, or the DNAME record has a higher priority [5]. Obviously, the Label Graph proposed by GRoot does not take this into account when generating equivalence classes, as shown in Figure 2(a).

Additionally, since the label graph does not consider operations such as aggregation of queries, it does not generate minimal LECs. From Figure 2(e), it can be seen that compared to the label graph, action trie has a significant advantage in generating the number of local equivalence classes.

Now, we'll provide a straightforward explanation for why our local equivalent class count can be smaller than GRoot's. Let's acknowledge that action trie is correct; indeed, its processing logic aligns with the RFC definition, considering all records. We know that the AN nodes in action trie are pairwise distinct, and the generation of equivalence classes relies on a reverse traversal of AN. Consequently, the number of equivalence classes equals the AN node count, which is fewer than the LN node count, as defined. Conversely, GRoot's enumeration method links the count of equivalence classes to the node count, which is no less than the LN node count defined in AT, all related to the number of labels. This leads to a better number of LECs generated by our method. Furthermore, it is precisely because AN nodes are different from each other that we can say that the local equivalence class generated by AT is the minimum.

3.2 Symbolic Execution and Property Verification

To achieve comprehensive coverage of all query possibilities, we utilize symbolic execution, a technique that enables us to explore various query paths and behaviors in the action trie without executing the actual query. Considering a forwarding scenario caused by a CNAME record (as shown in Figure 3), let's check whether there are any Non-Existent Domain faults within the zone old.com.

Before formal verification, it is necessary to deploy our framework to each DNS authoritative server and read the resource records it manages to build the action trie.

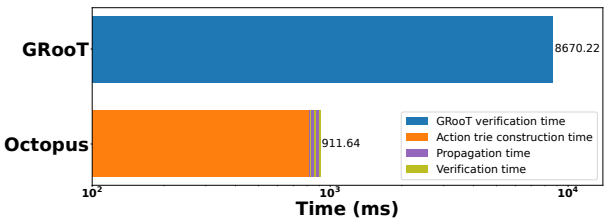
First, we traverse the action trie corresponding to the zone com and generate the seven local equivalence classes. One of the local equivalence classes corresponding to the action generates a new query q_1 and forwards it to the nameservers ns1.digimedia.com and ns2.digimedia.com. Here we only consider the former. Next, due to record ⓘ, a new query q_2 is generated, which is eventually received by ns2.digimedia.com and generates two local equivalence classes. The action of one of them gives NXDOMAIN, indicating that there is a Non-Existent Domain fault.

In this query path, we notice that not all LECs in the zone new.com are generated. Only one LEC related to query q_2 is matched and identified. This is one of the benefits of local equivalence classes, as they allow for more efficient matching by focusing on the relevant LEC for a specific query. In contrast, the centralized approach requires simulating and validating all server return values in addition to considering more equivalence classes.

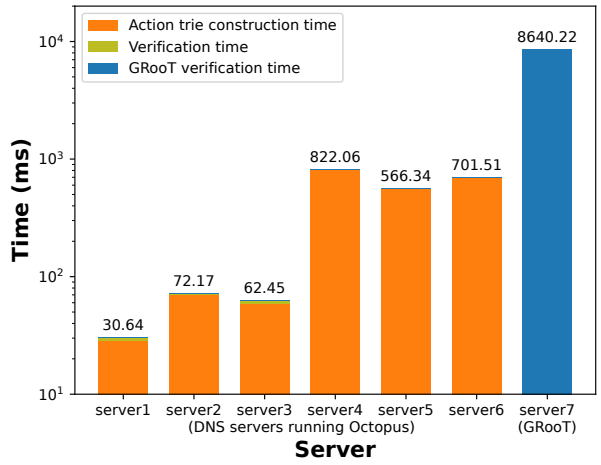
Then we discuss property classification for verification.

Chained properties. A chaining property refers to a fault caused by one or more configuration errors along a query path. A typical example is the Non-Existent Domain, which is caused by incorrect forwarding of an upstream configuration file or insufficient redundancy in the downstream configuration file. In this paper, only the information necessary to verify this type of property is appended to the query packet and forwarded. This makes it possible to log the entire process information for direct validation at each server. The advantage is that we not only distribute the simulation across all servers but also directly participate in the verification process. Please note that simulation and verification occur in parallel.

Tree Properties. In contrast to chained properties, tree properties cannot be determined by a single path alone, but are faults that result from multiple paths. An example of this is Answer Inconsistency, which occurs when different forwarding paths generate inconsistent results for the same query. To verify this property, we

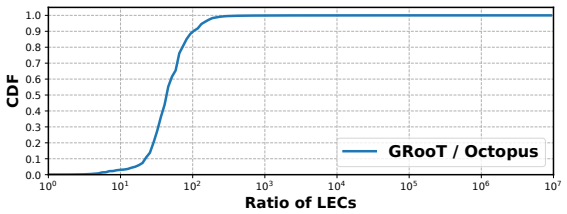


(a) The end-to-end verification time comparison between Octopus and GRooT with propagation latencies.

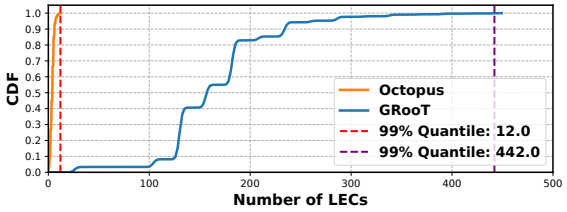


(b) Time breakdown on each server.

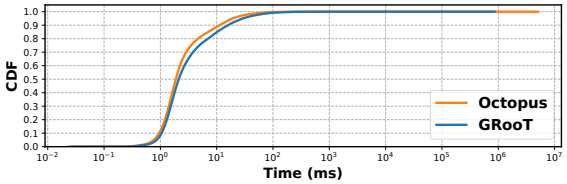
Figure 4: The verification time comparison between Octopus and GRooT.



(a) Ratio of the number of LECs/ECs generated by Octopus/GRooT on over six million zone files.



(b) The number of LECs/ECs generated by Octopus/GRooT on over six million zone files.



(c) LECs/ECs generation time of Octopus/GRooT on over six million zone files.

Figure 5: Local equivalence class and equivalence class generation comparison between Octopus and GRooT.

must record the necessary information during the query forwarding process on each server. This information is then accessed and verified by a designated critical server.

4 EVALUATION

To evaluate the performance and efficiency of Octopus, we demonstrate the testbed (§4.1) and LEC construction efficiency experiment (§4.2). The control group for all the experiments is the centralized architecture GRooT. Octopus consists of about 1500 lines of Java code. We use the open-sourced C++ code to implement the testbed experiment for GRooT. In the LEC construction efficiency experiment, we reproduce the LECs generation of GRooT using Java and experiment with reproduced code.

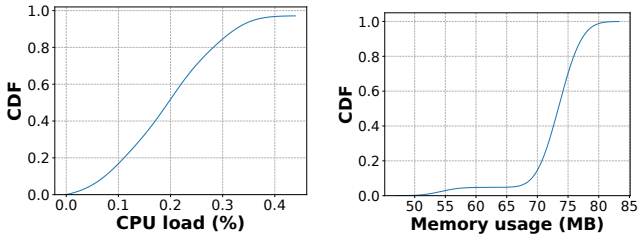
4.1 Testbed Experiments

The experiment employs 7 servers to simulate DNS servers based on Intel Xeon Silver 4210R @ 2.40GHz. We use 6 servers to build the distributed DNS configuration verification experiment testbed (Server1-Server6) and the remaining one as the centralized DNS misconfiguration experiment testbed (Server7). We also add the propagation latencies for the dataset based on its root DNS server location.

Dataset. A collection of DNS zone file datasets are procured from a campus laboratory includes 41 zones, each of which has about 10,000 records, for a total of 410,000 resource records.

Metric. We compare the performance of the two frameworks by comparing the end-to-end verification time to verify DNS misconfiguration. The DNS misconfiguration property that has been verified in the campus dataset is the loop (the same query that was received ≥ 2 for the same server) and the query exceeds the maximum length (after the DNAME rewrites the query size > 253 characters [15, 23]). We calculate their end-to-end verification time for these DNS misconfigurations. Here the end-to-end verification time contains AT construction time, and verification time. Since GRooT uses a multi-thread structure while verifying we do not extract them separately. Considering that the distributed DNS misconfiguration verification framework suffers from latencies, we provide experimental data containing propagation latencies. The propagation latencies is the constant of 30 ms, calculated as $distance$ / c (light speed). We select the distance between the IPv6 root servers in China and Japan as the distance for calculating the propagation latencies.

Our framework natively supports incremental deployment and can run in the background to realize periodic verification. This means that different servers can independently carry out update



(a) CPU load (b) Memory usage

Figure 6: CPU load and Memory usage of Octopus

operations to compute their respective AT. To stress this, we record the CPU load and memory usage of our system.

Result. Figure 4b shows GRoot finish checking in 8640.22 ms, while the most time-consuming DNS server in Octopus spends 822.06 ms. Because the first 6 server processes the on-device data in parallel and then performs the misconfiguration checking. This shows that Octopus has a better performance than GRoot on the single device. Then we pick the $Action\ trie\ construction\ time_{\max}$ (822.06ms) and $verification\ time_{\text{sum}}$ (29.58ms) to represent the total verification time with propagation latencies in Figure 4a. It shows that Octopus completes the verification in 911.64 ms, achieving an up to 9.51× speed up compared to the GRoot that is 8670.22ms. Even though Octopus requires to calculate the latency twice, the verification time is still lower than GRoot. The author of GRoot also shows in [16] that if the number of zone files is too large, it will blow up its verification time. From Figure 6, we can see that the incremental verification is running in the background all the time and the CPU load $\leq$ 0.4% and the memory usage $\leq$ 0.82 MB.

4.2 LEC Construction Efficiency

We conduct a comparative analysis between GRoot and Octopus by running an open-sourced real world dataset. This experiment aims to compare the efficiency of the two frameworks by comparing the number of LECs and the generation time.

Environment. In order to maintain the uniformity of the experimental environment, the LEC construction efficiency experiments all run on Intel(R) Xeon(R) Gold 6126 CPU 2.60GHz.

Dataset. We select the same dataset named Census [2] as GRoot. This dataset was collected from real-time DNS queries during 2012-2013, filtered by timestamps, and processed by DNS namespace hierarchy. The dataset consists of 1,368,523 domains and over 65 million resource records.

Metric. We focus on the LECs generation time and the number of LECs generated. The LECs generation time is the time consumed by the two frameworks in generating the equivalence classes, including I/O time and the LECs AT construction time.

Result. In Figure 5b, 99% number of LECs for Octopus $\leq$ 12 compares to GRoot $\leq$ 442. GRoot is about 36.8× larger than Octopus in the number of LECs. According to the ratio in Figure 5a, GRoot can be up to 316 times to Octopus in the number of LECs for all the data. For the LECs generation time in Figure 5c, Octopus and GRoot are at the same level and both of the generation time $\leq$ 100 ms. This shows that for the same number of zone files, Octopus can achieve the generation of fewer ECs, thus reducing the complexity of the data structures that the system has to traverse during the verification process. This reduces the amount of computation, verification time, and improves the verification efficiency.

5 DISCUSSION

Supported properties. This work can verify the same properties as GRoot. As demonstrated in the experiments, its effectiveness in verifying multiple files shows significant improvement compared to GRoot. However, for certain single zone file configuration issues, such as missing glue records, the improvement within the distributed framework could be limited. We plan to further investigate how to support the verification of other properties in the DNS iteration, such as cache verification.

Deployment. Our work can be deployed in various ways. For instance, the system can naturally be deployed in a distributed manner on platforms such as Akamai, Microsoft, Google, and others, allowing them to manage their own customers independently. Secondly, for DNS managed by organizations such as root servers, the system can be deployed as an agent on different servers, utilizing encryption techniques for message exchange to ensure security and reliability. As more organizations deploy the system, more people will be incentivized to use it once they see the benefits enjoyed by others.

Equivalence class representation. Currently, we design a tree-based approach to represent DNS record equivalence classes. However, the architecture of Octopus supports the usage of alternative representation methods, such as EPVerifier [33] and TOBDD [8].

Supporting incremental verification. Different from centralized verifiers such as GRoot, the design of Octopus naturally facilitates the incremental verification of DNS configuration updates. Consequently, when updates occur, different servers can independently perform incremental updates to calculate their respective AT, resulting in improved efficiency. We will continue to investigate incremental verification in the future.

Security and privacy risks. Plaintext-based DNS resolution risks privacy, so encryption is needed for communication between DNS resolvers and authoritative servers. Despite encryption standards like Q-min, DoT [11], DoH [9], and DoQ [12], the current probing policy still uses plaintext communication. However, a new approach [25] for discovering authoritative DNS servers ensures privacy and compatibility. Therefore, we don't need to consider privacy protection across multiple domains.

6 CONCLUSION

To solve the DNS misconfiguration issue, we propose to rearchitect DNS configuration verification with a distributed design. The evaluation results indicate that our design has better scalability and low overhead.

Acknowledgments. We are extremely grateful to the anonymous APNet'24 reviewers for their feedback. We also thank Siva Kesava Reddy Kakarla, the author of GRoot [15], for sharing the evaluation dataset. This work is supported in part by the National Key R&D Program of China 2022YFB2901502, NSFC Award #62172345, and NSF-Fujian-China 2022J01004.

REFERENCES

[1] Robert S Boyer, Bernard Elspas, and Karl N Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* 10, 6, 234–245.

[2] DNS Census2013. 2013. . https://dnscensus2013.neocities.org/ Accessed: March 16, 2024.

[3] Cloudfare. 2023. 1.1.1.1 lookup failures on October 4, 2023. https://blog.cloudflare.com/1-1-1-1-lookup-failures-on-october-4th-2023. Accessed: March 16, 2024.

[4] Internet Systems Consortium. 2009. Linux man page: named-checkzone. https://linux.die.net/man/8/named-checkconf. Accessed: March 16, 2024.

[5] Dr. Matt Crawford. 1999. Non-Terminal DNS Name Redirection. RFC 2672. https://www.rfc-editor.org/rfc/rfc2672.html

[6] Incident Report for npm. 2018. DNS misconfiguration cached in ISP DNS caches. https://status.npmjs.org/incidents/v22ffls5cd6h. Accessed: March 16, 2024.

[7] James Fryman. 2014. DNS Outage Post Mortem. https://github.blog/2014-01-18-dns-outage-post-mortem. Accessed: March 16, 2024.

[8] Dong Guo, Jian Luo, Kai Gao, and Y Richard Yang. 2023. Poster: Scaling Data Plane Verification with Throughput-Optimized Atomic Predicates. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1141–1143.

[9] Paul E. Hoffman and Patrick McManus. 2018. DNS Queries over HTTPS (DoH). RFC 8484. https://www.rfc-editor.org/rfc/rfc8484.html

[10] Check Host. 2020. Check Host. http://check-host.net/check-dns. Accessed: March 16, 2024.

[11] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. 2016. Specification for DNS over Transport Layer Security (TLS). RFC 7858. https://www.rfc-editor.org/rfc/rfc7858.html

[12] Christian Huitema, Sara Dickinson, and Allison Mankin. 2022. DNS over Dedicated QUIC Connections. RFC 9250. https://www.rfc-editor.org/rfc/rfc9250.html

[13] InfinityFree. 2019. DNS Outage at iFastNet: Softaculous down. https://forum.infinityfree.com/t/dns-outage-at-ifastnet-softaculous-down/19374. Accessed: March 16, 2024.

[14] Internet Systems Consortium, Inc. 2022. BIND - DNS Software. https://www.isc.org/bind/. Accessed: March 16, 2024.

[15] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRooT: Proactive verification of dns configurations. In *SIGCOMM'20*. ACM, 310–328.

[16] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. 2021. How Complex is DNS?. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*. 116–122.

[17] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation*. 99–111.

[18] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. 2023. A Formal Framework for End-to-End DNS Resolution. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 932–949.

[19] P. Mockapetris. 1987. Domain names - concepts and facilities. RFC 1034. https://www.rfc-editor.org/rfc/rfc1034.html

[20] P. Mockapetris. 1987. Domain names - implementation and specification. RFC 1035. https://www.rfc-editor.org/rfc/rfc1035.html

[21] Paul Mockapetris and Kevin J Dunlap. 1988. Development of the domain name system. In *Symposium proceedings on Communications architectures and protocols*. 123–133.

[22] SecurityWeek News. 2010. Reports of Massive DNS Outages in Germany. https://www.securityweek.com/content/reports-massivedns-outages-germany. Accessed: March 16, 2024.

[23] Wayne Schlitt and Meng Weng Wong. 2006. Sender Policy Framework (SPF) for Authorizing Use of Domains in E-Mail, Version 1. RFC 4408. https://www.rfc-editor.org/rfc/rfc4408.html

[24] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 314–327.

[25] Satoru Sunahara, Yong Jin, and Katsuyoshi Iida. 2023. Authoritative DNS Server Discovery Method to Enhance DNS Privacy Preservation. In *Proceedings of the on CoNEXT Student Workshop 2023*. 31–32.

[26] Liam Tung. 2019. 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/. Accessed: March 16, 2024.

[27] Zack Whittaker. 2021. A DNS outage just took down a large chunk of the internet. https://techcrunch.com/2021/07/22/a-dns-outage-just-took-down-a-good-chunk-of-the-internet/. Accessed: March 16, 2024.

[28] Qiao Xiang, Chenyang Huang, Ridi Wen, Yuxin Wang, Xiwen Fan, Zaoxing Liu, Linghe Kong, Dennis Duan, Franck Le, and Wei Sun. 2023. Beyond a Centralized Verifier: Scaling Data Plane Checking via Distributed, On-Device Verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 152–166.

[29] Tianyin Xu and Yuanyuan Zhou. 2015. Systems approaches to tackling configuration errors: A survey. *Comput. Surveys* 47, 4 (2015), 1–41.

[30] Hongkun Yang and Simon S Lam. 2015. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking* 24, 2 (2015), 887–900.

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

[32] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation*. 241–255.

[33] Chenyang Zhao, Yuebin Guo, Jingyu Wang, Qi Qi, Zirui Zhuang, Haifeng Sun, Lingqi Guo, Yuming Xie, and Jianxin Liao. 2024. EPVerifier: Accelerating Update Storms Verification with Edge-Predicate. In *21st USENIX Symposium on Networked Systems Design and Implementation*. 979–992.