Janus: Leveraging Incremental Computation for Efficient DNS Verification

Yao Wang[†], *Kexin Yu*[†], *Wenyun Xu*[†], *Ziyi Wang*[†], *Kaiqiang Hu*[⋆], *Lizhao You*[†], *Qiang Su*[†], *Dong Guo* [◇], *Haizhou Du*[⋆], *Wanjian Feng*[△], *Qingyu Song*[†], *Linghe Kong*[‡], *Qiao Xiang*[†], *Jiwu Shu*[▲†]

[†]*Xiamen University,* [⋆]*Shanghai University of Electric Power,* [△]*Yealink Technologies Co. Ltd., China,* [◇]*Fudan University,* [‡]*Shanghai Jiao Tong University,* ,[▲]*Minjiang University*

Abstract

Existing DNS configuration verification tools face significant issues (*e.g.*, inefficient and lacking support for incremental verification). Inspired by the advancements in recent work of distributed data plane verification and the resemblance between the data plane and DNS configuration, we tackle the challenge of DNS misconfiguration by introducing Janus, a DNS verification tool. Our key insight is that the process of a nameserver handling queries can be transformed into a matching process on a match-action table. With this insight, Janus consists of (1) an efficient data structure for partition query space based on the behaviors, (2) a symbolic execution algorithm that specifies how a single nameserver can efficiently cover all possible queries and ensure the accuracy of verification, (3) a mechanism to support incremental verification with less computational effort. Extensive experiments on real-world datasets (with over 6 million resource records) show that Janus achieves significant speedups, with peak improvements of up to 255.7× and a maximum 6046× reduction in the number of LECs.

1 Introduction

Over the past three decades since the initial publication of two RFCs [25, 26] 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 [3, 26, 27, 30, 33, 35]. Any failure in the DNS could lead to incalculable losses [7, 10, 11, 18, 36, 39].

Static methods support limited properties. To ensure uninterrupted DNS operations, operators employ various techniques, primarily centered on probing (*e.g.*, linting and testing) [2, 9, 16, 24, 29, 41] and the static analysis of individual files [8, 19, 40]. However, these tools have limited support for verifiable properties and lack precision. For example, the linting-based approach focuses on detecting basic syntax and formatting issues, and may not be able to detect complex logical errors or context-related problems.

Existing DNS configuration verification tools suffer from unsound definitions of Equivalence Class (EC), inefficient performance, and the lack of support for incremental update. Some studies [20,23,38] attempt to tackle the limitations of static analysis methods. GROOT [20] uses a centralized architecture to collect zonefiles are then passed to a centralized verifier for unified verification. Octopus [38] proposes a distributed DNS verification framework that allows each authoritative DNS server in the DNS system to focus on its own managed zonefiles and conduct behavioral analysis on each zonefile of the server. However, they still suffer from some issues.

First of all, we execute the source code for GROOT and observed certain scalability performance limitations. For 1,000,000 zones, it takes GROOT about 8455.13s to finish the verification. These findings indicate that GROOT demonstrates inefficiency even with moderate dataset sizes, which could impact its practicality in scenarios requiring faster verification speeds. Secondly, the EC definition of GROOT is unsound. As mentioned in the prior study [23] the same EC can exhibit different resolution behaviors in GROOT. This inconsistency arises when the same type of query leads to one being resolved with an immediate detection of a rewrite loop, while another requires additional queries to identify the same loop. Lastly, GROOT is intended to provide global verification of the entire DNS configuration. When the operator performs an incremental update of the DNS configuration, the logical structure of the algorithm may not support independent verification of only a portion of the configuration and can only update the entire domain name space, which does not support incremental update. Octopus [38] plans to support incremental verification, but it has not yet been implemented.

In this paper, we systematically tackle the important problem of DNS verification tools to be efficient, complete, and support incremental verification in the DNS ecosystem.

Proposal: Design an algorithm to compress the number of ECs, thereby reducing processing overhead. Inspired by the

studies of data plane verification on ECs [1,4,13,14,31,34,37], through a more in-depth analysis of the behavior of the nameserver, we compress ECs to reduce their number. A smaller number of ECs means that the volume of data to be processed can be reduced, thereby accelerating the speed of configuration verification. Additionally, we implement an efficient data structure to store and compute the EC and develop an incremental update algorithm based on the new EC definition.

Key insight: Nameserver can be modeled as a match-action table. The fundamental challenge in realizing DNS verification including: (1) how to construct and store each rule in the match-action table, (2) how to verify the configuration files through the query matching process, and (3) how to adapt to the frequent changes in configurations within the DNS system. To this end, we design Janus, a generic DNS configuration verification framework, with a key insight: by analyzing the query processing behaviors of nameservers to construct match-action tables and stitching the results in a symbolic way, we can substantially scale up the verification of DNS configuration. Janus has 3 key designs (D1-D3):

D1: A novel data structure to encode query space in each match-action table rule (§3). We use an equivalence class (EC) to aggregate DNS queries with the same behavior. In pursuit of the smallest possible number of ECs, we introduce a novel data structure and corresponding encode algorithm to calculate ECs. By aggregating all queries with the same behaviour, the ECs are disjointed from each other to obtain the minimum.

D2: A symbolic query algorithm to cover all possible query spaces (§4). 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 Janus without executing the actual query. We design a symbolic execution algorithm that takes a set of nameservers, a query, and a fuel parameter to simulate DNS behavior. By analyzing different resolution behaviors, the algorithm marks responses accordingly, allowing for the classification of various outcomes.

D3: A dynamic update with incremental verification algorithm (§5). DNS being an extremely large system, its configuration is updated extremely frequently and commonly. We regenerate execution traces affected by configuration updates, focusing on the affected query processing paths while maintaining the integrity of the verification.

Evaluation. We implement the prototype of Janus and evaluate it with a real-world open-source dataset [6] and a university dataset. We compare Janus with the first DNS verification tool: GROOT. For the open-source data set, we evaluate the end-to-end verification time that concludes the EC construction time, symbolic execution time, and incremental time. Additionally, the maximum number of GROOT end-to-end verifications can be as high as 255.7× that of Janus. In the incremental update scenario, we achieve a maximum speedup of 3,370×.

2 Overview

This section introduces some key concepts of Janus, and illustrates its workflow using an example.

2.1 Basic Concept

Simplification of the DNS System. Generally, the DNS system comprises three main roles [22]: clients, responsible for initiating and receiving query requests; resolvers, responsible for sending query requests to nameservers and receiving responses to serve the clients; and nameservers, responsible for storing the mappings between domain names and IP addresses to respond to resolver queries. In practice, these components can be further subdivided, such as into recursive resolvers and iterative resolvers.

To simplify verification challenges, we focus on the behavior of the nameservers that store configuration files, akin to the approach adopted in GROOT [20]. A nameserver typically consists of one or more configuration files, also referred to as zonefiles. Each zonefile contains a series of resource records (RRs) that describe the mappings between the domain names managed by the server and their corresponding IP addresses.

Each resource record (RR) comprises five components: domain name, TTL, class, type, and data. The domain name specifies the domain associated with the resource record (*e.g.*, www.google.com). The TTL (Time to Live) indicates the lifespan of the resource record. The class denotes the type of network the record belongs to (typically IN for Internet). The type specifies the data type of the resource record (*e.g.*, NS (designate the authoritative DNS server for a specific domain name), A (IPV4 address), AAAA (IPV6 address), CNAME (map one domain name to another domain name), etc.), and the data represents the content of the record (*e.g.*, an IP address or another domain name).

Since the class attribute is rarely used outside of IN, it will not be considered in subsequent discussions. Although TTL is a critical attribute of RRs, particularly for resolver caching, our focus on nameserver configuration files allows us to exclude TTL from further consideration. Errors related to TTL can be verified using existing tools such as named-checkzone [8]. Consequently, we simplify the RR to three components: $\langle rname, rtype, rdata \rangle$, where *rname* denotes the domain name, *rtype* represents the record type, and *rdata* denotes the record's data. And the

Furthermore, other DNS mechanisms, such as DNS caching and DNS Security Extensions (DNSSEC) [28], are excluded from consideration in this study to simplify the problem. In future work, we will progressively extend our model to accommodate additional DNS mechanisms.

DNS Model. To facilitate explanation, a domain nameserver can be modeled as a match-action table, where each row represents a rule. Each rule consists of two parts: matching conditions and actions. When a query request $\langle qname, qtype \rangle$

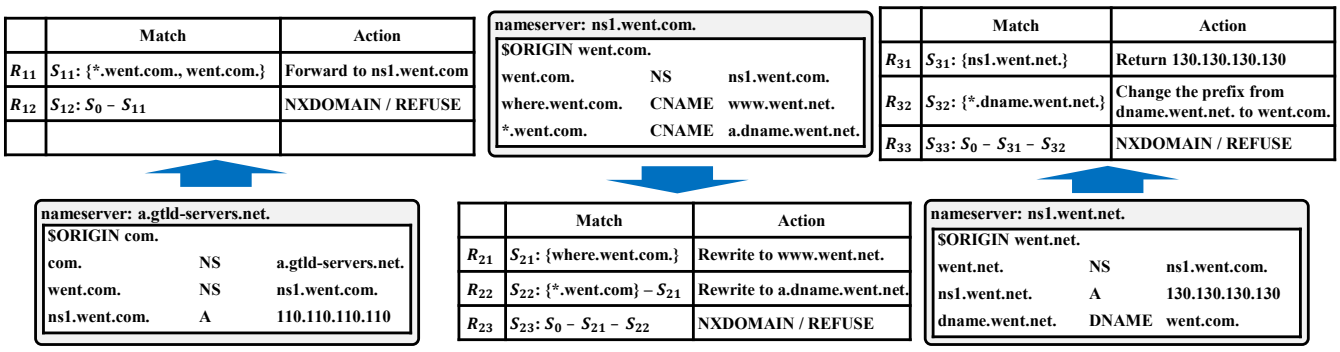


Figure 1: Example zonefiles for three nameservers and the result of query space partitioning for each nameserver. For simplicity, we merge the NXDOMAIN and REFUSE behaviors. S_0 denotes the entire query space.

arrives, it is matched against each rule in the table. Upon a successful match, the corresponding action is executed. The possible actions can be broadly categorized into two types: terminate and forward. A terminate action indicates that, upon a successful match, the query should terminate, returning the corresponding result (*e.g.*, the resolved data) or signaling a query failure. A forward action indicates that, upon a successful match, the query should be forwarded to the next domain nameservers for further resolution. The next servers could either be the current server itself or a different domain nameserver. Similar to the approach in GROOT, we assume that all configuration files required for verification are available and explicitly define which configuration files belong to each domain nameserver.

Query Log and Traces. We introduce *query trace* to record the processing workflow of a query, thereby defining the behavior of a query within the DNS system. When a query q passes through a DNS system, each successful match against a rule is defined as a query log. A log captures information about the initial value of the query, its final value (if it has been rewritten), and the location where the match occurred.

A query trace for q is then defined as an ordered sequence of logs, where the action in the final log must always be of the terminate type. However, for actions of the forward type, the number of subsequent domain nameservers cannot be deterministically identified. For example, it is well-known that root nameservers consist of 13 groups. A query q initiated by a user will be forwarded simultaneously to all 13 groups of root nameservers. As a result, a single query q may correspond to multiple query traces.

The concept of traces serves as the foundation for verification. All properties that require verification will be validated across all possible traces.

2.2 Workflow

We demonstrate Janus's workflow with the example in Figure 1 with the input query: $q_{origin} : \langle$*.com., A$\rangle$. The properties we verify here are that no query in the system, after being rewritten, fails to obtain a result (rewrite blackholing) or causes a loop (rewrite loop). We will also demonstrate how to perform incremental verification after fixing rewrite blackholing. In order to facilitate comprehension and streamline the demonstration process, this section assumes the presence of a single query of type A, thus omitting the type field from the query. Consequently, the input request will be denoted by q_{origin}:{ *.com.}.

2.2.1 Query Space Partition

Given a domain nameserver, Janus partitions the entire query space into several subspaces. After partitioning, the corresponding action for each subspace is determined. Each subspace can then be treated as a matching rule. Essentially, this process computes the match-action table. The core challenge lies in ensuring the completeness and mutual exclusivity of the partitioned subspaces. Specifically:

1. **Completeness** ensures that every possible query request can match exactly one subspace, guaranteeing the correctness of query processing.

2. **Mutual Exclusivity** ensures that no two subspaces overlap, thereby ensuring that a given query corresponds to a unique action.

To address this challenge, each subspace is treated as a set, and the problem is formulated as a set coverage problem, which can be solved using a greedy algorithm. The detailed algorithm will be elaborated in Section 3.

Figure 1 illustrates an example of computing a match-action table. In this example, match-action tables are computed for three domain nameservers. Since the computations for different domain nameservers are independent, they can be executed in parallel.

Let's take nameserver a.gtld-servers.net as an example and S_0 denotes the entire query space. For the domain nameserver a.gtld-server.net, the query space is partitioned into two subspaces:

S_{11}: {*.went.com., went.com.}, indicating that queries in this subspace are forwarded to the nameserver ns1.went.com.

S_{12}: S_0 - S_{11}, indicating that queries in this subspace could not get the correct response. In fact, we can further partition the space into {*.com., com.} - S_{11} and S_0 - {*.com., com.} according to whether the action is "domain does not exist" (NXDOMAIN) or "refused service" (REFUSE).

The subspaces partitioned for the remaining two domain nameservers are listed in Figure 1 and are not elaborated further here.

2.2.2 Trace Generation and Property Check

Once the match-action table is constructed, the process of generating query traces can begin. This process can be conceptualized as a depth-first search (DFS). Given a query q_{in} and a domain nameserver ns, We first match the query q with the match-action table of ns, which allows us to further refine the behavior of q. In other words, the match-action table partitions q into a set of subspaces $\{q_{in1}, q_{in2}, \dots\}$, and retrieves the corresponding action for each subspace q_{in}. If the action is of a termination type, we directly obtain a log. Otherwise, we execute the behavior specified by the action, obtain the query space q_{out} to be forwarded (which may be the same as q_{in} or different), record the information to generate a log, and forward q_{out} as a new request to the next nameserver. This process continues until the final action is of the terminate type, at which point a complete trace is generated. It is worth noting that due to the forwarding operation, q_{out} may be forwarded to multiple domain nameservers. As a result, the logs of different traces may overlap. However, this does not affect the definition or validity of individual traces.

Consider the scenario in Figure 2a, where a query q_{origin} is input to the domain nameserver a.gtld-server.net. We only consider the query matches rule R_{11}, generating a log L_1, where $q_{in1} = q_{out1} = S_{11}$. The query q_{out1} is then forwarded to ns1.went.com.

At ns1.went.com, we focus on the partial subspaces where the query matches successfully. The first match is with rule R_{21}, which modifies the input query q_{in2} = {where.went.com.} to a new output query q_{out2} = {www.went.net.} and forwards it to ns1.went.net. A log L_2 is generated for this match. Subsequently, at ns1.went.net, the query q_{out2} matches rule R_{33}, generating a log L_4. Here, $q_{in3} = q_{out3}$ = {www.went.net.} . Since the action corresponding to R_{33} is of the terminate type, the first trace is formed as: $T_1 = [L_1, L_2, L_4]$.

Next, we consider another match for q_{out1} at ns1.went.com, this time with rule R_{22}. The action for R_{22} modifies the query $q_{in3} = S_{22}$ to $q_{out3} = \{$a.dname.went.net.$\}$ and forwards it to ns1.went.net, generating a log L_3. At ns1.went.net, the query q_{out3} matches rule R_{33}, which modifies $q_{in5} = \{$a.dname.went.net.$\}$ to $q_{out5} = \{$a.went.com.$\}$, generating a log L_5. The query is then forwarded back to ns1.went.com, where it matches R_{23} again, generating a log L_6. Although further forwarding might occur, we stop here because of loop detection and define the second trace as: $T_2 = [L_1, L_4, L_5, L_6]$.

After obtaining these two traces, we can proceed to verify properties. For a given trace, Janus examines whether its behavior aligns with the semantics of specified properties.

For example, the rewrite blackholing property is defined as follows: if a rewrite operation occurs during the query process and the query ultimately fails to retrieve an answer result. Analyzing trace T_1, we find that a rewrite operation occurs in L_2, and the final action is NXDOMAIN. Therefore, T_1 satisfies the rewrite blackholing property.

Similarly, the rewriting loop property is defined as follows: if a rewrite operation occurs during the query process and the same domain nameserver receives the same query twice within the query path. Analyzing trace T_2, we observe that rewrite operations occur in both L_3 and L_5. Furthermore, the query is forwarded to ns1.went.com in both cases, and the intersection of q_{in3} and q_{in6} is $\{$a.went.com$\}$, indicating that ns1.went.com received the same query twice. Therefore, T_2 satisfies the rewriting loop property. Note that as mentioned in previous work [23], GROOT does not have a mechanism to make such a determination, so it is unable to detect the occurrence of loops in a timely manner.

2.2.3 Incremental Verification

When configuration files are updated, incremental verification allows us to quickly validate whether the new configuration satisfies the specified properties. This process primarily involves two steps: repartitioning the query space and incrementally generating and verifying traces.

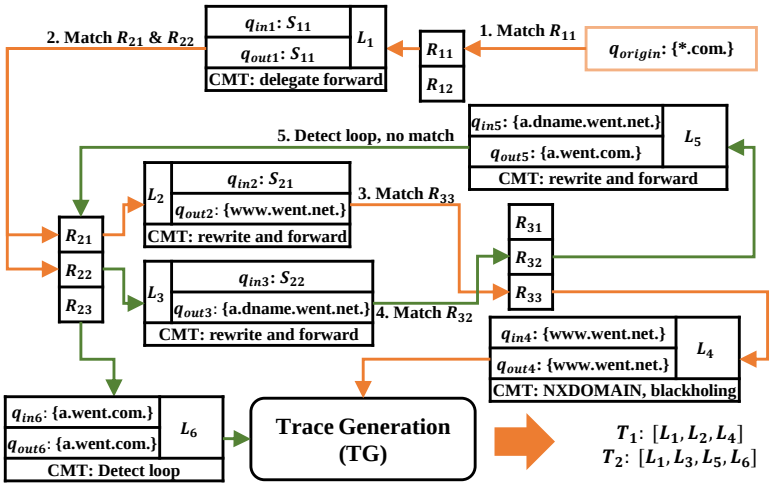
A straightforward and intuitive approach is to recompute the entire match-action table for domain nameservers with updated configurations. Then, all traces are regenerated from scratch, and properties are reverified. To reduce computation, previously cached traces can be leveraged.

Take the scenario in Figure 2b as an example. Suppose the record $\langle$where.went.com., CNAME, www.went.net.$\rangle$ is replaced with a new A type record $\langle$where.went.com., A, 100.100.100.100$\rangle$. In this case, we only need to recompute the match-action table for nameserver ns1.went.com. Rule R_{21} is replaced by R'_{21}, whose action directly returns the IPv4 100.100.100.100 and terminate.

Next, we regenerate all traces from scratch. For trace T_1, we observe that the rewrite operation in L_2 will no longer occur. Instead, q_{out1} obtains the query result after matching R'_{21}, generating a new log L'_2. Thus, the trace T_1 is replaced by $T'_1 : [L_1, L'_2]$, which means the trace no longer satisfies the rewrite blackholing property.

3 Query Space Partitioning

In this section, we will clarify how each nameserver constructs its own match-action table, along with some optimization techniques applied to it.



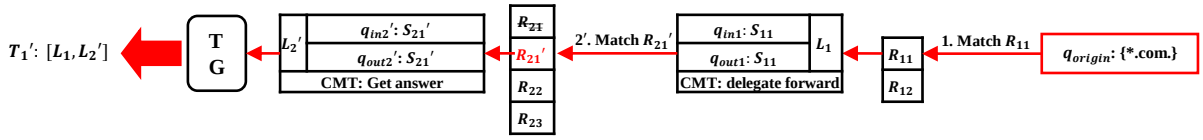
(a) The process of symbolic execution and generating two traces, T_1 and T_2. CMT is a comment on some key information in a log.

nameserver: ns1.went.com.

$ORIGIN went.com.		
went.com.	NS	ns1.went.com.
~~where.went.com.~~	~~CNAME~~	~~www.went.net.~~
where.went.com.	A	100.100.100.100
*.went.com.	CNAME	a.dname.went.net.

	Match	Action
~~R_{21}~~	~~S_{21}: {where.went.com.}~~	~~Rewrite to www.went.net.~~
R_{21}'	S_{21}': {where.went.com.}	Return 100.100.100.100
R_{22}	S_{22}: {*.went.com} – S_{21}'	Rewrite to a.dname.went.net.
R_{23}	S_{23}: S_0 - S_{21} - S_{22}	NXDOMAIN / REFUSE

(b) The result of the re-partitioned query space after changing a CNAME record (strikethrough) in ns1.went.com to an A record (red).



(c) The process of re-executing symbolic execution to update T_1 to T_1'.

Figure 2: An illustration example in Figure 1 to demonstrate the workflow of Janus.

3.1 Information Storage

For each nameserver object, the key data it should store is the LEC (Local Equivalence Class). Given a nameserver X, a LEC represents a set of queries whose actions are identical at X. X stores its LECs in a $(query\_space, aciton)$ mapping called the LEC table (match-action table). We choose to encode query sets as *predicates* using binary decision diagram (BDD) [5]. This is because the nameserver object performs query set operations (*e.g.*, $\cup$ and $\cap$), which can be realized efficiently using logical operations on BDD.

We cannot simply adopt previous work in data plane verification (DPV). In DPV, the length of packet headers is typically fixed (*e.g.*, IPv4 matching only needs 32 bits or variables in BDD). In DNS, domain names consist of multiple labels, separated by dots (.). Each label can be up to 63 characters long, and the overall domain name can be up to 255 characters in length. If we were to directly encode such domain names, we would require 255 * 8 = 2040 variables. This would render logical operations in BDDs exceedingly slow. To address this issue, we devise a compression Algorithm 1 that reduces each label in the domain name to an integer, decreasing the number of bits required.

Specifically, we utilize a hashmap as the data structure to store the encoding of all labels, which we denote as E. We attempt to traverse all possible resource records within the zonefiles under verification (line 7). Subsequently, for each resource record $\langle rname, rtype, rdata \rangle$, we analyze all the labels present in *rname*. For those labels that have not been encoded, we assign them a new value, which corresponds to the current length of E (lines 9, 1-5). Since we only add new label encodings to E and never delete any already encoded data, each label's encoding is guaranteed to be unique. Furthermore, we observe that for CNAME and DNAME records, the *rdata* field also represents a domain name. Therefore, we similarly encode the *rdata* field of such records (lines 10-11).

Algorithm 1: Encode labels as integer

```
1  Function UpdateEncodeMap(E, domain):
2      foreach label in domain do
3          if label not in E then
4              E[label] ← E.length;
5      return;
6  Function EncodeLabels(RRs):
7      Initialize E to an empty hash map;
8      foreach r in RRs do
9          UpdateEncodeMap(E, r.rname);
10         if r.rtype = CNAME or r.rtype = DNAME then
11             UpdateEncodeMap(E, r.rdata)
12     return E;
```

The number of variables required to construct a BDD has a logarithmic relationship with the number of distinct labels.

Assuming that all configuration files collectively contain m distinct labels, and the longest domain name consists of n labels, we only require $n * \lceil \log_2(m) \rceil + r$ variables to construct the BDD corresponding to all query sets. Here, r represents the number of variables needed to encode *rtype*. Since the encoding process for *rtype* is similar to that of labels, we will not elaborate further.

Details of storage LEC. We proceed to explain how a nameserver stores its own LECs. Initially, it is imperative to formally define an action to aggregate queries with identical actions into an LEC. Definition 1 stipulates that an action should consider the nameserver on which it resides, the zonefile, the *atype*, and the *adata*.

Definition 1 (Action) *We say* $Action_{s,z}(q) = \langle atype, adata \rangle$ *denotes the behavior generated by nameserver s and its zonefile z for q, where* $atype \in \{Answer, Delegate, RewriteC, RewriteD, Refuse, NonExist, ServiceFail, Loop\}$ *donates the type of action (*e.g.*, return the query answer Answer, delegate query to another nameserver or more nameservers Delegate, rewrite caused by CNAME record RewriteC, etc.) and adata stores the data needed for the action (*e.g.*, IP that should be returned, the value after CNAME rewrite, etc.).*

With the definition of ACTION, we design a hierarchical storage structure for LECs. Given a nameserver *ns*, we represent it as $\langle zones, refuse\_rule \rangle$, where *zones* encompass all the zonefile objects contained within *ns*, and *refuse_rule* denotes the refusal rule. For a specific zonefile object, its structure is $\langle rules, hit, bdd, nx\_rule \rangle$, with *rules* representing all the rules stored in the zonefile, *hit* indicating the query space that the zonefile can handle independently of other zonefiles, *bdd* representing the current query space that the zonefile can process, and *nx_rule* corresponding to the rule corresponds to rules for non-existent query results. A specific rule consists of three parts $\langle hit, bdd, action \rangle$, where the definitions of *hit* and *bdd* are similar to those in the zonefile, and *action* signifies the behavior corresponding to that rule.

This hierarchical architecture enables us to organize LECs efficiently. It also aligns with our definition of an action, which is tied to the nameservers and its zonefiles. This implies that LECs on different nameservers are distinct, and different zonefiles under the same nameserver also possess different LECs.

3.2 LEC Construction

To construct the LECs for each nameserver and each zonefile, it is imperative to delineate the logic by which nameservers process queries. While RFC1034 [25] provides a recommended template for a processing algorithm, the specific processing rules are ultimately determined by individual operating systems and software. This paper references the semantics provided by GROOT [20] to design a set of algorithms suitable for LEC construction, ensuring both completeness and mutual exclusivity. The specific algorithm is presented as Algorithm 4 and 2 (due to space constraints, we place Algorithm 3 and 4 in the appendix.).

Before delving into the specifics of the algorithm, we first clarify the $GetSpace(rname, types, flag)$ method mentioned therein. The primary objective of this method is to convert a given domain name and set of types into a BDD representation, effectively a query set. Here, *rname* is a sequence of labels stored as a string, and *types* represents a set of query types included in the query set, where **all_types** means all query types in the DNS system. A crucial parameter, $flag$, dictates whether and how subdomains are considered. Taking example.com as an instance, when $flag$=0, the domain name space represented is {example.com.} itself; when $flag$=1, the domain name space includes itself and its subdomains (i.e., {example.com., *.example.com.}); and when $flag$=2, the domain name space is solely its subdomains, excluding itself (i.e., {*.example.com.}).

Algorithm 2: Construct the LECs for a nameserver

```
1  Function ConstructLECs(zonefiles):
2      Sort zonefiles by number of labels in origin;
3      zones ← empty list;
4      remain_bdd ← true;
5      foreach zonefile in zonefiles do
6          zone_hit ← GETSPACE(zonefile.origin, all_types, 1, ;
7          , )zone_bdd ← zone_hit ∩ remain_bdd;
8          remain_bdd ← remain_bdd − zone_bdd;
9          rules, nx_rule ← zonefile, zone_bdd;
10         zones.push(⟨rules, zone_hit, zone_bdd, nx_rule⟩);
11     refuse_rule ← ⟨remain_bdd, remain_bdd, ⟨Refuse, ()⟩⟩;
12     return ⟨zones, refuse_rule⟩;
```

Next, in Algorithm 2, we partition the space for the different zonefile objects on the nameserver *ns*. As mentioned earlier, we need to ensure mutual exclusivity among LECs. Therefore, during the computation process, we maintain a variable *remain_bdd* to keep track of the remaining available domain space. The order of space allocation is determined by the length of origin (i.e., the domain covered by the zone), following the principle of longest prefix matching. Specifically, each zone first calculates the maximum space it can handle, denoted as *zone_hit*. Then, it subtracts its allocated space from *remain_bdd* and updates *remain_bdd* accordingly. This ensures that shorter zonefile objects can only acquire space from what remains, guaranteeing that the spaces of different zonefile objects are mutually exclusive. It is important to note that the space a zonefile can claim is unrelated to the domain name records it contains. In other words, when *ns* matches a query to a zonefile, it does not take specific resource records into account. After allocating query spaces to all zonefile objects, the behavior for the remaining space is set to $Refuse$.

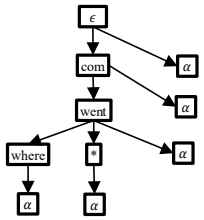
Then, in Algorithm 4, we consider how to compute each specific LEC. Given a zonefile and its allocated query space, its LECs are primarily divided based on the resource records

it contains. First, we aggregate records with the same *rname* and *rtype*, as these records evidently correspond to the result of the same query . Then, we calculate the priority (rank) for each aggregated record . The purpose of this step is similar to the earlier process: determining the order of query space division to ensure that LECs are mutually exclusive. Here, we need to handle two special cases: DNAME and CNAME. For DNAME records, for instance, $\langle$a.com., DNAME, a.net.$\rangle$, a query such as $\langle$r.a.com., A$\rangle$ will be rewritten to $\langle$r.a.net., A$\rangle$, while a query $\langle$a.com., DNAME$\rangle$ directly returns the result a.net. Thus, DNAME essentially involves two possible behaviors. Similarly, CNAME records exhibit comparable behavior. Therefore, we generate two rules for these types of records, with the two rules presented at different ranks. Finally, we compute each rule, following a process similar to that used for allocating space to zonefile objects. Notably, since the rank computation has already taken into account the assignment of different action types to separate ranges, the behavior corresponding to each rule can be quickly determined during rule computation. At this point, we complete the entire process of constructing LECs for a nameserver.

3.3 Further Optimization

A more comprehensive label encoding scheme. At times, due to the presence of DNAME records, the actual number of labels may exceed n. For instance, consider the record $\langle$a.com., DNAME, a.a.com.$\rangle$. This record can lead to unlimited rewriting, where the domain name *.a.com is rewritten as *.a.a.com, and then further rewritten as *.a.a.com, ad infinitum, until the maximum length is exceeded. It is evident that during this process, the number of labels in the domain name continuously increases. Although we can terminate the process upon detecting a cycle, it is challenging to directly identify cycles that span multiple zone files, meaning we cannot ascertain the maximum number of labels in the query domain name at the end of the first cycle. In such cases, we delegate the setting authority to the user, allowing them to specify the number of redundant labels *rl* (i.e., the maximum domain name length is $n+rl$ labels), thereby ensuring that the query process does not overflow due to insufficient label count reservation.

Additionally, assuming we have 16 distinct labels with a maximum of 4 labels per domain, the BDD variables are allocated such that $x0-x3$ encode the first-level label, $x4-x7$ encode the second-level label, and so on. Since the number of labels at different levels may not be the same, we can reduce the number of BDD variables by employing separate label encoding tables for different levels. For instance, if there are only 4 distinct labels at the first level and 8 at the second level, we can use $x0-x1$ to encode the first-level labels and $x2-x4$ for the second-level labels. However, we must consider the implications of DNAME records. Taking $\langle$a.com., DNAME, a.a.com.$\rangle$ as an example, b.a.com is rewritten as b.a.a.com. This means that although *b* appears at the third level in the record, it can actually appear at the fourth level during the process. Therefore, users need to configure the system to use independent hashmaps for the initial levels and a shared hashmap for subsequent levels. In this example, starting from the third level, all subsequent levels should share the same hashmap.



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

1	$\langle \epsilon, t \rangle$
2	$\langle \alpha.\epsilon, t \rangle$
3	$\langle$com.ϵ, $t\rangle$
4	$\langle \alpha$.com.ϵ, $t\rangle$
5	$\langle$went.com.ϵ, $t\rangle$
6	$\langle \alpha$.went.com.ϵ, $t\rangle$
7	$\langle$*. went.com.ϵ,$t\rangle$
8	$\langle$where. went.com.ϵ,$t\rangle$
9	$\langle \alpha$. where. went.com.ϵ,$t\rangle$
10	$\langle \alpha$.*. went.com.ϵ,$t\rangle$

(b) Local equivalence classes generated using the Label Graph, one for each query type.

Nameserver	Zone	Number of LECs	Number of LECs (LG)
a.gtld-servers.net.	com.	3	8 * T
ns1.went.com.	went.com.	3	10 * T
ns1.went.net.	went.net.	3	14 * T

(c) Similarly, we can derive local equivalence classes for the other zone files in Figure 1(a). Here we give a comparison of the number of equivalence classes generated by the two methods. 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 3: Comparison of the LEC/EC in the example of Figure 1 in detail.

Pursuing the ultimate number of LECs. Whether to reduce the storage overhead of LECs or to lower the time complexity of traversing all equivalence classes, we aim to minimize the number of equivalence classes as much as possible. To achieve this, we can perform further compression on the LECs generated in Section 3.2. Specifically, we can merge all query spaces with the same action into a single LEC, resulting in a minimal set of LECs. Thanks to the efficiency of BDD in performing logical operations, this merging process does not require significant computational effort. By comparing the EC generated by GROOT in Figure 3 with the LEC generated by Janus in Figure 2, it can be seen that GROOT generates LECs for the 3 zonefiles that at least 2x more than Janus generates. This shows that Janus has a significant effect on the compression of equivalence classes. At this point, the resulting LEC set for a nameserver *s*, denoted as $C = \{c_1, c_2, ..., c_n\}$, satisfies the following constraints:

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

2) $\forall z \in s : \forall c \in C : \forall q_1, q_2 \in c : q_1 \neq q_2 \implies Action_{s,z}(q_1) = Action_{s,z}(q_2)$.

3) $\{q \mid q \in c \cap c \in C\} = The\ complete\ query\ space$.

4) $\forall z \in s : \forall c_1, c_2 \in C : \forall q_1 \in c_1, q_2 \in c_2 : c_1 \neq c_2 \implies Action_{s,z}(q_1) \neq Action_{s,z}(q_2)$.

We call such a set the minimal LEC set.

Algorithm 3: Janus symbolic execution logic

```
1  Function Resolve(S, q, k):
2      foreach s zones, refuse_rule in S do
3          q' ← q − refuse_rule.bdd;
4          if q' ≠ ∅ then
5              Resolve(S, s, q', k);
6          q = q − q';
7      if q ≠ ∅ then
8          action ← ⟨ServiceFail, ()⟩;
9          Save relevant match information;
10 Function Resolve(S, s, q, k):
11     if s is not exist or k <= 0 then
12         action ← ⟨ServiceFail, ()⟩;
13         Save relevant match information;
14         return;
15     visit ← ⋃pq, where pq represents all queries that have
         previously accessed s;
16     if q ∩ visit ≠ ∅ then
17         action ← ⟨Loop, ()⟩;
18         Save relevant match information;
19         return;
20     foreach rule ⟨hit, bdd, action⟩ in s do
21         q' ← q ∩ bdd;
22         if q' ≠ ∅ then
23             Save relevant match information;
24             if action.atype = Delegate then
25                 Resolve(S, rule.action.adata, q', k-1);
26             if action.atype ∈ {RewriteC, RewriteD} then
27                 q' ← Calculate new query after rewrite;
28                 Resolve(S, q', k-1);
```

4 Symbolic Query Execution

Given a DNS subsystem comprising multiple nameservers, after constructing LECs for each nameserver, we should commence the search for errors. We introduce symbolic execution technology to cover all possible queries, achieving comprehensive coverage and ensuring the accuracy of verification.

4.1 A Symbolic Query Algorithm

We now formally present how Janus executes the symbolic query q and returns the final result. As illustrated in Algorithm 2, the first function accepts a nameserver set S, a query q, and a fuel parameter k, which serves as a mechanism to mimic DNS behavior for guaranteeing query resolution termination. This function operates by processing the query q across each nameserver $s \in S$ (lines 2-6). We will consider queries that cannot be resolved by the current DNS system and mark the behavior of these queries as *ServiceFail* (lines 7-9).

The second function delineates the process by which the nameserver s handles the query q. If the specified s does not exist within the DNS system, typically implying the delegation of the query to a null nameserver, the query is terminated (lines 11-14). Next, we consider the scenario where query loops occur. In Janus, each nameserver actively stores all

nameserver: ns.dns.com.

$ORIGIN dns.com.

dns.com.	**NS**	**ns.dns.net.**
1.dns.com.	**A**	**0.0.0.1**
2.dns.com.	**A**	**0.0.0.2**
…	**…**	**…**
9999.dns.com.	**A**	**0.0.39.15**

Rule No.	Match		Action	
	Qname	Qtype	Atype	Adata
r1	dns.com.	NS	Answer	ns1.dns.net.
r2	1.dns.com.	A	Answer	0.0.0.1
r3	2.dns.com.	A	Answer	0.0.0.2
…	…	A	Answer	…
r9999	9999.dns.com.	A	Answer	0.0.39.15
r0	Remaining query space		NonExist	-

Figure 4: An example with a large number of LEC.

the queries it has processed and checks for duplicates with each new query received to prevent loops (lines 15-19). Subsequently, for each rule held by s, we examine whether it matches q (lines 21-23). Upon successfully matching a non-terminal node, the query process continues (lines 24-28). For instance, if the query q necessitates delegating to a designated server, the next-hop nameserver is determined based on the *adata* stored in the action, and recursive resolution is continued.

4.2 Matching Optimization

Although BDD is highly efficient in logical operations, as the number of rules on a nameserver increases, matching a query against each rule remains a time-consuming task. Let us consider an extreme case, illustrated in Figure 4. The nameserver ns.extreme.com generates a total of 10,000 LEC rules (r0 - r9999). When it receives a query $q : \langle$2.example.com.,A$\rangle$, to process q, we need to perform 10,000 logical AND operations, each with a time complexity of $O(|G| \cdot |G|)$, where $|G|$ is the number of nodes in the BDD. Thus, the final time complexity would be $10000 \cdot O(|G| \cdot |G|)$. If the server needs to handle multiple queries, this could become a performance bottleneck for the system.

To address this, we need to perform targeted filtering before formal matching to identify LECs that may potentially affect q. Both Veriflow [21] and Flash [12] utilize multi-dimension prefix Trie to efficiently discover or filter rules. Borrowing from this concept, we observe that LECs generated by Section 3.2 can correspond to a specific record r, allowing us to filter LECs related to the query through *r.rname*. Specifically, we introduce a $prefix$ field to the query q, which denotes the common prefix of the query space and is initially empty. This allows us to determine whether to perform the logical AND operation by assessing if the $prefix$ matches *r.rname* (i.e., the $prefix$ is a subdomain of *r.rname* or *r.rname* is a subdomain of the $prefix$) before proceeding with the matching process. Returning to the example in Figure 4, We assume that the query $q : \langle$1.extreme.com.,A,1.extreme.com.$\rangle$ is derived from the query $\langle$*.extreme.net.,$A\rangle$ through rewriting

Error	Description
Delegation Inconsistency	For a trace that the parent node with the answer type is Ref and the child with the answer type is ANS but do not have the same set of NS and A records for delegation.
Lame Delegation	For a trace that the answer type returns REFUSED.
Missing Glue Records	For a trace that the answer type in any log is NS but not followed by the A (glue) records.
Non-Existent Domain	For a trace that the answer returns NXDOMAIN.
Cyclic Zone Dependency	For a trace that there exists two logs that are the same.
Rewriting Loop	For a trace that there exists a rewrite that result in a loop.
Query Exceeds Maximum Length	For a trace that the length of the input query or the number of the labels exceeds the maximum number.
Rewrite Blackholing	For a trace after the rewrite log the next log returns NXDOMAIN.

Table 1: Types of DNS misconfiguration that our framework support. We conclude from GROOT [20], the formal method [23] and the RFC [15].

due to the record $\langle$ *.extreme.net.,CNAME,1.extreme.com.$\rangle$, meaning it matches a *RewriteC* action. Here, we define that when a query matches a *Delegate* type action, the prefix is set to *r.rname*; when it matches a *RewriteC* or *RewriteD* type action, the prefix is set to *r.radata*; other types of actions do not alter the prefix. When q reaches ns.extreme.com again, we perform 10,000 comparison operations, ultimately only executing the logical AND operation with r2. This reduces our time complexity to $O(10000 \cdot n + |G| \cdot |G|)$, where n is the length of *rname*.

4.3 Property Check

Upon completing symbolic execution, we obtain all execution traces, each representing the behavior of an EC q. These traces enable verification by analyzing whether q's behavior adheres to predefined assertions. Specifically, each trace captures the sequence of query resolution steps, including delegation, rewriting, and response generation, which can be systematically compared against expected outcomes. Finally, we list in Table 1 a few typical error representations and how they are verified in our model.

5 Incremental Verification

We propose an efficient algorithm for fast incremental verification when configuration files are updated.

5.1 LECs Update

Currently, our primary focus is on updates involving the addition or deletion of resource records within a zonefile. This section does not address the wholesale deletion or addition of zonefiles within a nameserver.

Basic Idea: Priority-based preemptive update. In Algorithm 2, we can deduce two properties for each rule r where $atype$ is not *NonExist*: 1) The *r.hit* of each rule is mutually independent; 2) Each rule implicitly contains a field *rank*, which influences the calculation of *r.bdd*, meaning that rules with higher ranks are computed first. We denote rules with $atype$ as *NonExist* as nr. We denote the rule with type *NonExist* as nr, and assign its priority as $nr.rank = -1$.

Step 1. Compute the query space for the update rule. When we insert a resource record $\langle rname1, rtype1, rdata1 \rangle$ into zonefile z, if z already contains another record $\langle rname2, rtype2, rdata2 \rangle$ that satisfies $rname1 = rname2 \wedge rtype1 = rtype2$; or when deleting $\langle rname1, rtype1, rdata1 \rangle$, if it satisfies $rname1 = rname2 \wedge rtype1 = rtype2 \wedge adata1 \neq adata2$, the LECs remain unchanged, and only the $adata$ field of the corresponding rule is updated. Otherwise, we add or delete a rule r from z, and Equation (1) provides the method for calculating *r.bdd*. When deleting a rule, since the rule has been precomputed and exists within z, we can directly retrieve *r.bdd*. For the case of adding a rule, we will preempt the query space from nr and other lower-rank rules r'.

$$r.bdd = \begin{cases} r.bdd, & \text{delete rule } r \\ r.hit \wedge \left(\bigvee_{r.rank > r'.rank} r'.bdd \right), & \text{add rule } r \end{cases} \tag{1}$$

Step 2. Update of affected rules. Since we performed space preemption on the rules with lower ranks while calculating *r.bdd*, we now need to update the BDDs of those rules that were subject to space preemption. Equation (2) provides the algorithm for this operation. After the deletion of rule r, the space held by r will be released, and all rules with lower ranks (r') will sequentially preempt the released query space. In other words, r' cannot preempt the space $r''.hit$ that is intended to be preempted by a higher-priority rule r''. For the case of adding a rule, r' simply removes the query space previously taken by r.

$$r'.bdd = \begin{cases} r'.bdd \vee \Big(r'.hit \wedge r.bdd \wedge \\ \neg \Big(\bigvee_{r''.rank > r'.rank} r''.hit \Big) \Big), & \text{delete rule } r \\ r'.hit \wedge \neg r.bdd, & \text{add rule } r \end{cases} \tag{2}$$

Optimizing the computation process. As mentioned in Section 4.2, logical operations still incur a significant time overhead. Therefore, we will now introduce some tricky methods

to reduce the amount of logical operations. We begin by analyzing the process of computing the disjunction of all $r'.bdd$ terms as shown in Equation (1). We can transform the entire process into a form similar to "$r.hit \wedge r'_1.bdd \wedge r'_2.bdd$ ...". By doing so, after each conjunction operation, we can check if $r.bdd$ is equal to $r.hit$ in order to decide whether to terminate early, thus reducing the computational complexity. For Equation (2), we can adopt a similar approach by tracking the space released or preempted by r (initially $r.bdd$), which we denote as s. After each update to $r'.bdd$, we update s accordingly. When s becomes empty, we can terminate the traversal of r'. As for the disjunction of r'', we can apply dynamic programming techniques to avoid recalculating the entire disjunction each time r' is updated. Additionally, we can accelerate the computation by comparing the *rname* fields between rules to filter out unrelated or unaffected rules.

Addition and deletion at the zonefile level. To extend the aforementioned methodology to the addition or deletion of zonefiles on a nameserver *ns*, we discuss zonefile-level operations. In Algorithm 2, zonefiles on *ns* are sorted by the hierarchical depth of their origins within the DNS system, where the origin's hierarchy determines its rank. For instance, a top-level domain (*e.g.*, com) is assigned a rank=1, while a second-level domain (*e.g.*, went.com) is assigned rank=2. Each zonefile is endowed with three attributes: *hit*, *bdd*, and *rank*, enabling direct application of Equations (1) and (2) for incremental updates of LECs. For a zonefile z whose *bdd* is updated, this can be interpreted as either implicitly adding a rule with $rank = 512$ (where the rule's *bdd* corresponds to the preempted query space from z) or implicitly deleting a rule with $rank = 0$ (where the rule's *bdd* corresponds to the newly acquired query space for z). Next, just update all the rules in z in the same way you would add or delete a rule.

5.2 Incremental Execution

To enhance verification efficiency while circumventing the computational burden of full symbolic execution, we propose a delta-update mechanism that selectively regenerates execution traces impacted by modifications to LECs.

Phase 1: Affected trace identification. When a zone file z is modified, the system analyzes dependencies via log relationships. It retrieves historical logs L linked to z and extracts the maximal preceding sub-trace T_{prev}, capturing nameserver states and query history. Input queries $Q = \{q_1, q_2, ..., q_n\}$ are filtered using BDD-based checks to isolate relevant queries.

Phase 2: Partial symbolic re-execution. Recomputation begins at the nameserver hosting z, with execution parameters restored from T_{prev}. Input queries Q are replayed through the updated LEC configuration, pruning unchanged BDD subspaces. The system generates version-tagged logs T_{next}.

Phase 3: Trace integration and verification. The final synthesis aligns T_{prev} and T_{next} to reconstruct event timelines. Verification follows the approach outlined in section 4.3.

6 Evaluation

We implement a prototype of Janus using Rust based on the OxiDD [17] framework and conduct extensive evaluations on Janus. For all the zones in the dataset, the initial input query is $q : \langle *, * \rangle$, which means that for all possible domains and all the types of query, we want to know their response. Specifically, we study the following questions: (1) What is the capability of Janus in verifying DNS configuration files? (§6.1) (2) What is the performance of Janus in a real-world DNS dataset? (§6.2)

6.1 Capability Evaluation

To maintain the uniformity of the experimental environment, the verification experiments all run on the platform configured with Ubuntu 20.04.3 LTS (kernel version 5.13.0-41-generic), equipped with two Intel Xeon Gold 6126 CPU (2.60GHz) and 128GB of DDR4 memory.

Error	GROOT	Janus
Delegation Inconsistency	×⋆	✓⋆
Lame Delegation	✓⋆	✓⋆
Rewriting Loop	✓⋆	✓⋆
Rewrite Blackholing	✓⋆	✓⋆
Number of rewrites >=2	✓†	✓†
Number of hops >=2	✓†	✓†

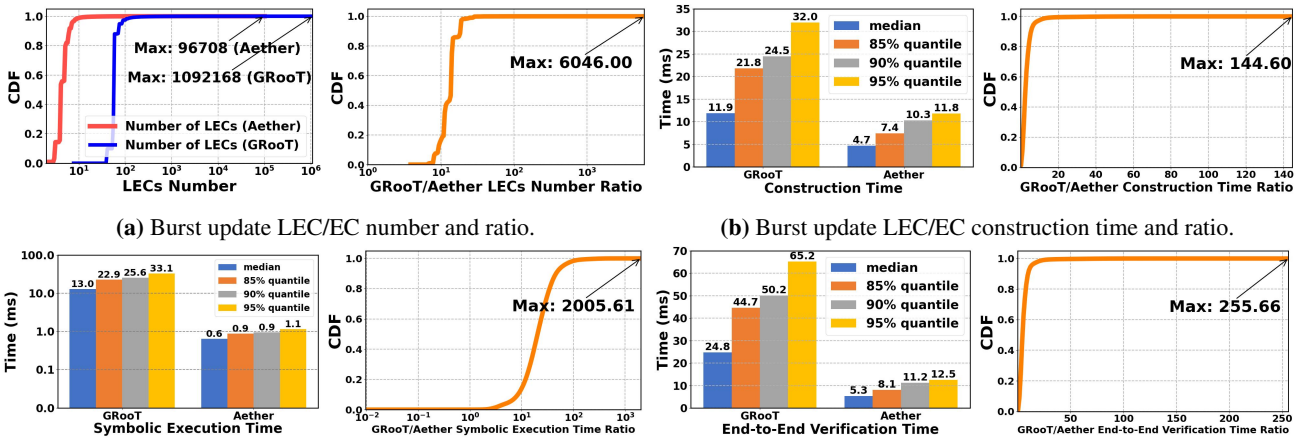
Table 2: Errors checked on the campus dataset and the number of cases Janus reported. Cases in red with marker ⋆ are errors while orange with marker † are warnings.

To evaluate the capability of Janus, we set up the experiment on the university dataset and evaluate the property that Janus verified.

Dataset. We collect the DNS information from a university with more zonefiles from multiple nameservers and any wildcard records to show the capability of Janus. The university dataset obtained from the laboratory features multiple levels of subdomains (more than 2). It includes 9,700 zonefiles with 134,349 resource records in total.

Metric. We record the DNS configuration errors that Janus and GROOT verified in this dataset.

Experiment result. The results in Table 2 demonstrate that Janus successfully verified 4 types of DNS configuration errors and 2 types of warnings, while GROOT verified 3 types of DNS configuration errors and 2 warnings. Through analyzing Groot's code and design, we discovered discrepancies between the implementation and the documentation. This led to an incomplete detection of the Delegation Inconsistency error. The system only checks whether the parent and child domains are about the same domain name by verifying the consistency of the first record's name. Consequently, if the child domain's zone file is located in a subsequent position, it



(a) Burst update LEC/EC number and ratio.

(b) Burst update LEC/EC construction time and ratio.

(c) Burst update symbolic execution time and ratio.

(d) Burst update end-to-end verification time and ratio.

Figure 5: Burst update experiment result using Census dataset.

may be missed during verification.

Additional Experiment. As mentioned in the previous section, the definition of GROOT is unsound. To demonstrate this, we have set up an additional experiment. The example illustrated in our workflow (Figure 1) serves as a representative case. Through this experimental result, we intend to illustrate not only the unsoundness of GROOT but also the necessity of introducing the concept of local equivalence classes.

We transform the examples in the workflow into a dataset containing three zonefiles.we run the source code of GROOT and Janus to get the verification result.We check the default properties of GROOT, as it claims that the default properties can detect all loops. However, during the execution, we find that GROOT did not verify the existence of a rewriting loop. Then we check for the "Number of Hop" and the result shows GROOT found 3 number of hop errors. In Janus, we identify 6 errors, one of which is rewrite blackholing, another is a rewrite loop, and the remaining 4 are the number of hop errors. This indicates that, under the local equivalence classes we defined, Janusis capable of verifying these errors.

6.2 Performance Evaluation

To evaluate the performance of Janus, we set up the experiment on the open-sourced dataset and evaluate the end-to-end verification time in the scenario of burst update and incremental update.

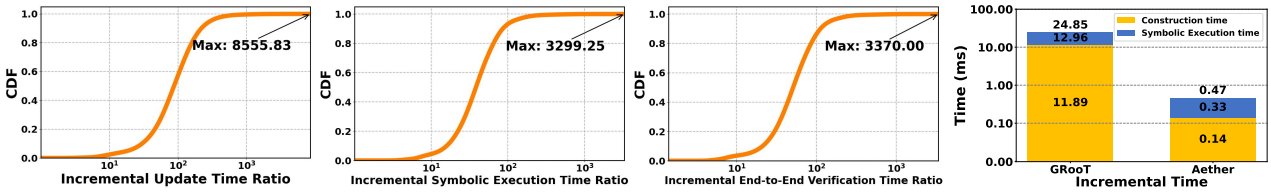
Dataset. We get the dataset from the real world named Census [6]. The census consists of 1,368,523 zones and over 65 million resource records. However, according to our test results with Bind9 [32], some files in this dataset violate the named-checkzone directive (*e.g.*, quotes are not enclosed). These zones are defined as invalid zones, and we filter them out in our experiments (remaining 1,186,517 zones) because they are syntactically incorrect or incomplete. Thees files cannot be parsed successfully.

Metric. We evaluate the LEC construction time, symbolic execution time, and end-to-end verification time (LEC construction time + symbolic execution time) in burst and incremental update scenarios to represent the overall performance. We also compare the number of LEC of Janus with the number of EC of GROOT to show our contribution to the definition of LEC.

Experiment 1: Burst update. We first evaluate Janus in the scenario of burst update (make the complete censu dataset available to Janus at once for full-volume evaluation). We record the number of LECs generated by Janus on each zonefile and compare it with GROOT. The experimental results are shown in Figure 5a indicate that for all zones, the number of ECs generated by GROOT is greater than that produced by Janus, with a maximum increase of 6046×. For Janus, the maximum LEC number is 96708, whereas for GROOT is 1092168. This demonstrates that Janus is highly effective in compressing equivalence classes. For instance, aggregating the behavior of rules containing NS records and A records means that identical resolutions for these two types of requests can be treated as a single equivalence class. In contrast, GROOT's compression requires enumerating these two request types separately, resulting in an exponential increase in the number of equivalence classes to compress.

We also compare the construction time of LECs and ECs, and the results are shown in Figure 5b. The median construction time for GROOT is 11.89 ms, while Janus achieves a median of 4.65 ms. Additionally, at the 85%, 90%, and 95% percentiles, GROOT completes construction within 32.03 ms, compared to Janus's 11.83 ms. Furthermore, from the ratio figure the maximum speedup reaches 144.6×. This is because we support parallel computation during the generation of local equivalence classes, whereas GROOT's architecture requires all zonfile equivalence classes to be computed sequentially.

In symbolic execution (Figure 5c), the median execution



(a) Incremental update construction time ratio, symbolic execution ratio and end-to-end verification time ratio. All the ratios are calculated as: GROOT/Janus.

(b) The mean time of end-to-end verification time.

Figure 6: Incremental update experiment result using Census dataset.

time for GROOT is 12.96 ms, while Janus achieves a median of just 0.64 ms. At the three quantiles, the maximum time for Janus is 1.15 ms, compared to a minimum of 22.88 ms for GROOT. The ratio figure also indicates Janus can achieve a maximum speedup of 2005.61×. Because Janus generates fewer equivalence classes, it reduces the need to match repetitive behaviors during symbolic execution, thereby improving the efficiency of symbolic execution. Additionally, for end-to-end verification (Figure 5d), Janus achieves a median runtime of 5.29 ms, compared to 44.67 ms for GROOT. Quantile analysis further highlights this improvement: the maximum runtime for GROOT is 65.15 ms, whereas Janus reduces it to just 12.54 ms. There is a maximum speedup of 255.66× of end-to-end verification, as shown in the ratio figure. And as we have mentioned befored, for 1,000,000 zones Janus can finish verification in 140.56s that acheieves 80.87x speed up.

Experiment 2: Incremental update. We randomly delete and update records in census and then evaluate the performance of Janus and GRoot. For each zone, we randomly select a zonefile to modify according to the following rules: first, we randomly choose a record type from the rtype. We suppose the number of records in the zonefile as *num* and calculate $Bound = \max(\min(num \times 0.01, 10), 1)$. Then, within the range of $[1, Bound]$, we generate a random number to create the corresponding number of records. Finally, we generate another random number within the range of $[1, Bound]$ to delete the specified number of records. Throughout the running period (Figure 6a), Janus consistently outperforms GROOT, achieving a peak ratio of up to 8555.83×. In symbolic execution, Janus demonstrates speedups across all zones, with a maximum speedup of 3299.25×. Similarly, in end-to-end verification, Janus achieves speedups in all zones, reaching a maximum of 3370×. Figure 6b shows the mean of incremental end-to-end verification time component of Janus and GROOT. Notably, Janus demonstrates a speedup of 82.57×, while symbolic execution achieves a speedup of 39.27×. Because Janus only needs to update the query space affected by the modified rule, while GROOT must re-collect the zone file, generate equivalence classes, and perform symbolic execution to support rule changes.

7 Conclusion

We present Janus, a novel DNS verification framework that achieves efficient and incremental verification through a new data structure, LEC. Experimental results demonstrate its effectiveness. In future work, we plan to investigate methods for diagnosing and repairing erroneous DNS configurations.

This work does not raise any ethical issues.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. *Acm sigplan notices*, 49(1):113–126, 2014.

[2] Oracle and/or its affiliates. dig - dns lookup utility. https://docs.oracle.com/cd/E88353_01/html/E37839/dig-1.html, 2020. Accessed: January 16, 2025.

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

[4] Conor Black and Sandra Scott-Hayward. A survey on the verification of adversarial data planes in software-defined networks. In *Proceedings of the 2021 ACM International Workshop on Software Defined Networks & Network Function Virtualization Security*, SDN-NFV Sec'21, page 3–10, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[6] DNS Census2013. DNS Census2013, 2013. Accessed: January 16, 2025.

[7] Cloudfare. 1.1.1.1 lookup failures on october 4, 2023. https://blog.cloudflare.com/1-1-1-1-lookup-failures-on-october-4th-2023, 2023. Accessed: January 16, 2025.

[8] Internet Systems Consortium. Linux man page: named-checkzone. https://linux.die.net/man/8/named-checkconf, 2009. Accessed: January 16, 2025.

[9] Carlos Perez etc. dnsrecon. https://github.com/darkoperator/dnsrecon, 2014. Accessed: January 16, 2025.

[10] Incident Report for npm. Dns misconfiguration cached in isp dns caches. https://status.npmjs.org/incidents/v22ffls5cd6h, 2018. Accessed: January 16, 2025.

[11] James Fryman. Dns outage post mortem. https://github.blog/2014-01-18-dns-outage-post-mortem, 2014. Accessed: January 16, 2025.

[12] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y. Richard Yang. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 314–335, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. Sdx: a software defined internet exchange. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 551–562, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Sylvain Hallé. Test suite generation for boolean conditions with equivalence class partitioning. In *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*, FormaliSE '22, page 23–33, New York, NY, USA, 2022. Association for Computing Machinery.

[15] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). RFC 8484, 2018.

[16] Check Host. Check host. http://check-host.net/check-dns, 2020. Accessed: January 16, 2025.

[17] Nils Husung, Clemens Dubslaff, Holger Hermanns, and Maximilian A. Köhl. OxiDD: A safe, concurrent, modular, and performant decision diagram framework in Rust. In *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24)*, 2024.

[18] InfinityFree. Dns outage at ifastnet: Softaculous down. https://forum.infinityfree.com/t/dns-outage-at-ifastnet-softaculous-down/19374, 2019. Accessed: January 16, 2025.

[19] Internet Systems Consortium, Inc. BIND - DNS Software. https://www.isc.org/bind/, 2022. Accessed: January 16, 2025.

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

[21] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. 42(4):467–472, September 2012.

[22] J. Klensin. Rfc3467: Role of the domain name system (dns), 2003.

[23] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. A formal framework for end-to-end dns resolution. In *SIGCOMM'23*, pages ACM, 932–949, 2023.

[24] Microsoft. Description of the dnslint utility, 2020. Accessed: January 16, 2025.

[25] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, 1987.

[26] P. Mockapetris. Domain names - implementation and specification. RFC 1035, 1987.

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

[28] Moritz Müller, Matthew Thomas, Duane Wessels, Wes Hardaker, Taejoong Chung, Willem Toorop, and Roland van Rijswijk-Deij. Roll, roll, roll your root: A comprehensive analysis of the first ever dnssec root ksk rollover. In *Proceedings of the Internet Measurement Conference*, IMC '19, page 1–14, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Inc MXToolBox. Mxtoolbox. https://mxtoolbox.com, 2004. Accessed: January 16, 2025.

[30] Jeffrey Pang, Aditya Akella, Anees Shaikh, Balachander Krishnamurthy, and Srinivasan Seshan. On the responsiveness of dns-based network control. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04. Association for Computing Machinery, 2004.

[31] Federico Parola, Roberto Procopio, Roberto Querio, and Fulvio Risso. Comparing user space and in-kernel packet processing for edge data centers. *SIGCOMM Comput. Commun. Rev.*, 53(1):14–29, April 2023.

[32] Keerthi PS. Bind9 check zone file - how we do it, Mar 2020. Accessed: January 16, 2025.

[33] Kyle Schomp, Tom Callahan, Michael Rabinovich, and Mark Allman. On measuring the client-side dns infrastructure. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13. Association for Computing Machinery, 2013.

[34] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting blackbox sdn control software with minimal causal sequences. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 395–406, New York, NY, USA, 2014. Association for Computing Machinery.

[35] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of dns-based server selection. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, 2001.

[36] 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. Accessed: January 16, 2025.

[37] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. Cope: traffic engineering in dynamic networks. *SIGCOMM Comput. Commun. Rev.*, 36(4):99–110, August 2006.

[38] Yao Wang, Kexin Yu, Ziyi Wang, Kaiqiang Hu, Haizhou Du, Qiao Xiang, Xing Fang, Geng Li, Ruiting Zhou, Linghe Kong, and Jiwu Shu. Rethinking dns configuration verification with a distributed architecture. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, page 23–30, New York, NY, USA, 2024. Association for Computing Machinery.

[39] Zack Whittaker. 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/, 2021. Accessed: January 16, 2025.

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

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

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

A Pseudocode for constructing LEC for a zonefile.

Here is the pseudocode th demonstrates how Janus constructs the LEC for a single zonefile in detail.

Algorithm 4: Construct the LECs for a zonefile

```
1  Function Rank(r, origin):
2      if r.rtype = NS and r.rname ≠ origin then
3          return 512 − r.rname.num_labels ∗ 2;
4      else if r.rtype = DNAME then
5          return 512 − r.rname.num_labels ∗ 2 − 1;
6      else
7          is_cname ← r.rtype = CNAME;
8          is_not_wildcard ← ∗ ∉ r.rname;
9          return is_not_wildcard ∗ 2 + is_cname;
10 Function ZoneLECs(zonefile, remain_bdd):
11     Aggregate records with the same rname and rtype;
12     records ← empty list;
13     foreach record r ⟨rname, rtype, rdata⟩ in zonefile do
14         rank ← Rank(r, zonefile.origin);
15         records.push(⟨rname, rtype, rdata, rank⟩);
16         if r.rtype = DNAME then
17             records.push(⟨rname, rtype, rdata, 2⟩);
18         else if r.rtype = CNAME then
19             records.push(⟨rname, rtype, rdata, rank − 1⟩);
20     Sort records by decreasing rank;
21     rules ← empty list;
22     foreach ⟨rname, rtype, rdata, rank⟩ in records do
23         if rank > 3 and rank % 2 = 0 then
24             rule_hit ← GetSpace(rname, all_types, 1);
25             action ← ⟨Delegate, adata⟩;
26         else if rank > 3 and rank % 2 = 1 then
27             rule_hit ← GetSpace(rname, all_types, 2);
28             action ← ⟨RewriteD, adata⟩;
29         else if rank = 1 or rank = 3 then
30             rule_hit ← GetSpace(rname, all_types - CNAME,
                 0);
31             action ← ⟨RewriteC, adata⟩;
32         else
33             rule_hit ← GetSpace(rname, rtype, 0);
34             action ← ⟨Answer, adata⟩;
35         rule_bdd ← rule_hit ∩ remain_bdd;
36         remain_bdd ← remain_bdd − rule_bdd;
37         rules.push(⟨rule_hit, rule_bdd, action⟩);
38     return (rules, ⟨remain_bdd, remain_bdd, ⟨NonExist, ()⟩⟩);
```