# ResolFuzz: Differential Fuzzing of DNS Resolvers

Jonas Bushart[1] and Christian Rossow[1]

CISPA Helmholtz Center for Information Security, Germany
{jonas.bushart,rossow}@cispa.de

**Abstract.** This paper identifies and analyzes vulnerabilities in the DNS infrastructure, with particular focus on recursive DNS resolvers. We aim to identify semantic bugs that could lead to incorrect resolver responses, introducing risks to the internet's critical infrastructure. To achieve this, we introduce ResolFuzz, a mutation-based fuzzer to search for semantic differences across DNS resolver implementations. ResolFuzz combines differential analysis with a rule-based mechanism to distinguish between benign differences and potential threats. We evaluate our prototype on seven resolvers and uncover multiple security vulnerabilities, including inaccuracies in resolver responses and possible amplification issues in PowerDNS Recursor's handling of `DNAME` Resource Records (RRs). Moreover, we demonstrate the potential for self-sustaining DoS attacks in resolved and trust-dns, further underlining the necessity of comprehensive DNS security. Through these contributions, our research underscores the potential of differential fuzzing in uncovering DNS vulnerabilities.

## 1 Introduction

The Domain Name System (DNS) is often explained as the internet's phone book since it turns human readable names like esorics2023.org into an IP address. This analogy greatly simplifies the central role DNS plays on the internet besides just delivering IP addresses. In fact, DNS defines the singular namespace of the internet, provides cryptographic material for secure communications, and acts as a backbone for other services such as anti-spam measures or secure routing.

This makes DNS part of the critical infrastructure. Thus, risks and vulnerabilities in DNS are of the highest concern. Recursive resolvers are the centerpiece of DNS, as they resolve domains for clients, iteratively getting answers from authoritative name servers. By design, resolvers are public or at least semi-public, exposing them to various threats from malicious clients. Likewise, they interact with potentially malicious authoritative name servers. This complex setting also complicates the task of testing DNS resolvers, as it requires modeling both, clients and authoritative name servers. Combined with their central role, this means DNS servers are a highly prized target for malicious actors. Infiltration and manipulations of DNS allow far-reaching exploits like preparing for Denial-of-Service attacks, intercepting communication, or forging TLS certificates. Further security protocols like DNSSEC or full TLS encryption of services like email are not widespread enough to catch DNS manipulations.

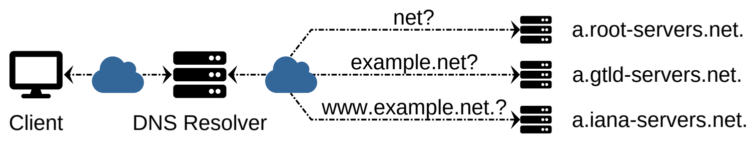

**Fig. 1.** Basic DNS resolution process. The client sends a query to the resolver, which then recursively resolves the query by interacting with multiple AuthNSes.

In this paper, our primary goal is to identify and analyze semantic bugs and gaps in the DNS resolver implementations. Such bugs could allow an attacker to get a resolver to return wrong answers. With this work, we want to support developers by highlighting problems earlier in the development process, as they hint at bugs or problems with the specification. To this end, we developed ResolFuzz, a fuzzer specifically designed for DNS and implementing differential testing mechanisms, allowing us to test thousands of scenarios. We build a rule-based mechanism to identify common and benign differences, such as random values, underspecified behavior, or feature differences.

In the course of our research, we have uncovered a multitude of critical issues. These range from cases where the DNS resolver returned incorrect values, to more complex problems like traffic looping bugs and potential amplification issues. These findings underscore the importance of our differential fuzzing approach in identifying and addressing vulnerabilities in the DNS infrastructure.

In summary, our paper presents the following contributions:

1. We create a fuzzer for recursive DNS resolvers to uncover vulnerabilities.
2. We build a differential analysis framework for investigating DNS outputs. This includes rules to separate common benign differences from other sources.
3. We have discovered multiple new bugs in popular open-source resolvers.

## 2 Methodology

In this section, we describe our methodology for finding semantic differences between DNS resolvers. Before we can dig into the technical details, we first need to provide a bit of background on the DNS protocol and the functionality of a DNS resolver. With that in mind, we can lay out our goals and the challenges we face. Lastly, we provide the technical details of ResolFuzz, the infrastructure choices we made, and describe the input generation and output analysis.

### 2.1 Threat Model

Recursive resolvers sit at a very precarious place on the internet, as shown in Fig. 1. They receive queries from client and answer them from their cache. If the information is missing, they traverse the DNS hierarchy to find the answer from an AuthNS (right). Many resolvers are exposed to the whole internet and have to talk to many untrusted AuthNSes, indicated by the blue clouds. This opens them up to attacks from clients, AuthNSes, or combined attacks.

For testing the behavior of resolvers, we need to assume that all network communication is potentially malicious. We assume an attacker can send arbitrary queries to the resolver and has control over one or more AuthNSes. While this assumption is trivially fulfilled for public resolvers, it even holds for private resolvers. Server-side requests in HTTP servers, like fetching link previews, or checking email authentication information in SMTP servers allow users to send queries to specific domains, although with less control over the query.

Creating different resolver states is trivial if the attacker is allowed to send different data to the resolvers. For a fair analysis, we must restrict the scenario such that all resolvers receive the same logical data. Some variation must be allowed, as DNS messages are not fully deterministic since they contain random values and have no canonical encoding.

## 2.2 Goals

We aim to develop a semi-automated approach for finding and evaluating semantic differences between DNS resolvers. We envision two use cases: i) Finding semantic bugs in DNS resolvers, which lead to differences, such that two DNS clients no longer agree on the same answer. In the worst case, this is a cache poisoning vulnerability, which can, e.g., lead to a Certificate Authority that wrongly issues a domain-validated certificate. ii) Our system can be used during the development of DNS resolvers, either to ensure a new resolver is compatible with the existing ecosystem or to check how a new RFC is implemented.

The system should be scalable to many resolvers and be easy to use, thus it should work with minimal domain-specific knowledge from the human user. This allows it to be run by developers and protocol designers to check their implementations. Integrating a resolver should be free of code modifications or complex adoption necessary by some fuzzing systems.

We do not aim for bugs in the network packet parsing code. Fuzzing on that level will not yield many interesting results in the DNS behavior, since most packets will be dropped early. Bugs in the parsers can be found better with fuzz harnesses for the parsers. Therefore, we only create syntactically valid DNS packets, i.e., following the size and allowed values of the fields.

## 2.3 Challenges

DNS is a complex protocol. Over 300 IETF RFCs [3,21] specify different aspects of it, and over 100 of them are relevant to resolver implementations [4,15]. This provides a lot of potential for implementation differences and bugs. Automatically exploring such semantic differences is challenging for several reasons.

**State:** A DNS resolver is inherently stateful. Resolvers cache answers, metadata about answers, and information about AuthNSes. Over time the same query can result in different answers. Even at the same time, the same query from different clients can result in different answers, since the client IP address can be part of the cache key with Extended DNS (EDNS) Client Subnet (RFC 7871 [5]).

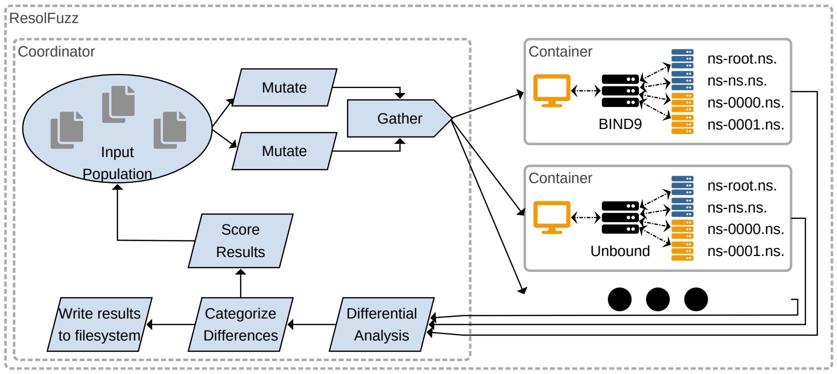

**Fig. 2.** Overview of our ResolFuzz fuzzer. The boxes indicate different components, where the main component is the Coordinator. The Helper (blue/yellow) runs inside each container and acts as client and servers. The arrows indicate the dataflow. From a pool of fuzzing inputs, some are selected and mutated. They are gathered into a larger set and sent to the resolvers. The yellow parts are controlled by the fuzzing input. The results are collected and compared. Inputs resulting in differences are written to the filesystem for later analysis. All new inputs are scored and added to the pool.

**Multiple Clients and Servers:** The DNS resolvers sit in the middle between many clients and AuthNSes. Answering a single query can involve multiple servers, pointing the resolver to the next place to ask. The resolver only handles untrusted data. Clients can send queries for any domain name, even for attacker-controlled ones. Our system must be able to represent enough of this complex interaction to provide a large enough coverage but also to find bugs.

**Feedback:** A DNS resolver is for the most part a black box. It receives a query, does some processing, potentially triggering network requests, and returns an answer. On the protocol level, we only have input-output-based interaction, with multiple inputs and outputs. But this lacks information about what is happening inside the resolver, including its state (e.g., cache).

**Output Similarity:** The human language specification used in RFCs is often vague and leaves room for interpretation. Later RFCs might clarify or change the meaning of previous ones. This makes it hard to determine if a difference is a bug or a valid interpretation of the specification. For example, the wording around glue records in RFC 1034 [18] has been interpreted differently by different implementations, requiring further clarification [1].

## 2.4 Addressing the Challenges

We use differential fuzzing to tackle the aforementioned challenges. In differential fuzzing, the same fuzzing input is given to multiple implementations, here resolvers. The fuzzer then compares outputs to report on any differences, which may indicate bugs. This approach still leaves us with the challenge of classifying the differences into critical or benign ones, due to the *output similarity* problem.

Unfortunately, there is no reference implementation of a perfect DNS resolver that we could use as an oracle. Thus, to reason about the identified differences, we use a combination of heuristics and manual inspection. First, we define rules that describe classes of benign differences, e.g., deviations in the DNS ID of the header. We group the remaining differences based on fingerprints. They consist of the DNS headers of both outputs and a list of all fields that differ between the two. This cuts down the number of cases that need to be inspected manually.

We restrict the *state* by only using one client-side query during fuzzing, with arbitrarily many AuthNS side responses. This brings the state down to a manageable level but can miss some bugs. For example, the first query caches the wrong AuthNS for a domain and the answer of the second query is based on that value. We do inspect the cache of the resolver, after the fuzzing. For this, we send cache probing queries, i.e., queries with the recursion desired (RD) bit set to 0. This forces the resolver to answer only from the cache. This cache snooping also allows us to obtain *feedback* for fuzzing. But only relying on cache state and DNS messages as output would be too coarse-grained for a fuzzer, which performs best if learning about incremental progress. We thus extend the definition of output and include edge coverage between basic blocks [31].

The fuzzer runs in a fully separated DNS environment with separate DNS root servers, as shown in the containers in Fig. 2. For the *multiple clients and servers* challenge we simulate all AuthNSes. The DNS environment is separated using a custom Root Hint [12] configuration for all resolvers and only pointing to AuthNSes on localhost, which is simulated by ResolFuzz.

## 2.5 Fuzzing Infrastructure

This section describes the architecture of ResolFuzz. A flowchart of our dataflow is shown in Fig. 2. ResolFuzz consists of multiple components. This is necessary to achieve scalability in the number of tested resolvers and to keep modifications on the resolvers minimal. The main component *Coordinator* is responsible for input generation, mutation, and output evaluation. A helper component *Helper* runs alongside each resolver and provides a simulated DNS ecosystem for the resolver and communicates with the Coordinator to get fuzzing inputs and deliver the outputs back. We describe how the Helper interacts with the resolver and afterward explain the interaction between the Coordinator and multiple Helpers.

**Resolver Isolation:** One of our goals is that adding a new resolver should require minimal adaptions. One way we achieve this is by running the resolver unmodified, with a normal network stack. However, this requires that we separate different resolvers from each other since all resolvers listen on the same port. For this, we use Linux namespaces in the form of containers. The Helper simulates a custom DNS ecosystem with separate roots and runs in each container.

The Helper also runs the fuzzing inputs against the resolver. Multiple fuzzing queries can be run sequentially, with separate AuthNSes on different IPs per query. This helps with the separation of state between multiple fuzzing inputs.

Lastly, the Helper gathers coverage information from the resolvers and communicates it to the Coordinator. Each resolver is instrumented with LLVMs

Sanitizer Coverage [31] pass to gather edge coverage information and only include the edges between sending a query and receiving a response. All edges involved in startup and background tasks are excluded.

**Container Startup:** Spawning a resolver including the Helper in such a fashion is relatively expensive compared to individual DNS queries. For fast fuzzing iterations it is therefore necessary to limit the amount of time we wait for the resolvers to start. Generic network fuzzers like AFLNet [19] come with a forkserver to speed up the spawning of new processes, but that requires modifications to the program and is incompatible with threads. We pre-spawn the containers such that they are ready to receive fuzzing inputs and execute multiple fuzzing inputs sequentially, thus sharing the startup cost. Unfortunately, we cannot run them in parallel, as this would interfere with the coverage information gathering since coverage information is a global property of the resolver, and concurrent queries would interfere with each other.

Second, most of the resolver startup is independent of the concrete fuzzing input. We warm the resolver cache with unrelated DNS queries, such that all name servers will be cached before fuzzing. Testing the fuzzing inputs now only requires the Helper to read the inputs, configure the dynamic DNS server, and start sending the client queries.

**Coordinator and Helper Interaction:** The Coordinator is responsible for generating and mutating the fuzzing inputs and output evaluation, as well as, managing the containers with the Helper. We explain the steps of input generation and evaluations in more detail in Sections 2.6 and 2.7, respectively. For now, it is sufficient to know, that input generation generates a batch of fuzzing inputs, each consisting of a DNS client query and a set of DNS responses.

After the batch is completed for all resolvers, the output results are collected. The output contains i) edge coverage information between basic blocks in the form of a hitmap, ii) all DNS queries sent to AuthNSes by the resolver, iii) the DNS responses provided to these queries, iv) the DNS response sent to the client, and v) information about the resolver cache state. From these outputs, we determine which inputs are “interesting”, i.e., cover new code paths or uncover behavioral differences. More details are in Sections 2.7 and 2.8.

### 2.6 Input Generation and Mutation

ResolFuzz uses mutation for input generation. By picking specific mutations, we can ensure that the mutated inputs stay syntactically valid DNS messages. Having a valid DNS message is important, such that we are fuzzing semantic bugs and not bugs in the DNS message parsers.

We have a population of inputs that we mutate and the ability to generate new inputs. In the beginning, our population is empty, and we start with only newly generated inputs. They are a single DNS query and response pair with randomized query names, labels, types, classes, and header flags. We always include some new random entries in a batch to ensure diversity in the population.

The top $n$ inputs in our population with the highest score are mutated by adding, removing, or modifying the different fields and values of the DNS messages. The score is assigned before adding the input to the population and is decremented each time the input is used as a base for mutations. It covers information about the coverage increase, how many known differences, and unknown differences were found when the input was last used.

Mutation consists of modifying existing values and where possible adding or removing values. The typed in-memory representations of DNS messages allow us to walk over the structure and pick a mutation for each field. The DNS header has a fixed length, so the only mutations are changing the existing values to new ones. The different sections can have resource records added or removed. Each resource record has a domain, type, class, Time-to-Live (TTL), and data. These can be modified, but we always ensure that the data is still valid for the type.

Modifying domain names requires more care than random modifications to ensure enough "collisions" are created to trigger proper behavior in the resolvers. For example, the query name used in the DNS client query should also appear in one of the DNS responses, as otherwise the resolver likely ignores responses and we fail to test the core logic of a resolver. We use a small set of labels from which a domain name can be generated. The inputs use a fixed domain `test.fuzz.` during mutation, but the Helper will later replace the labels with unique ones to separate the inputs. We have two C-string specific mutations, by adding a zero-byte at the end of the last label, i.e., `test.fuzz\0.`, and adding the zero-byte but also duplicating the domain name, i.e., `test.fuzz\0.test.fuzz.`. Two labels can be merged into a new dot-containing label, i.e., `foo` and `bar` can be merged into `foo\.bar`. These mutations verify that the resolver treats domain names as a sequence of labels and not as a string.

Only a small set of RR type and Record Data (RDATA) is generated. While the RR type is a 16-bit value, most of the values are not assigned, and hence the creation of valid RDATA for them is impossible. Most RR types are for data storage, without interacting with the resolver, and only a few should be interpreted by the resolver. These include `A`/`AAAA` for the IP addresses of AuthNSes and `NS` for delegations between AuthNSes, `SOA` for caching, `CNAME`/`DNAME` for aliases/canonical names, and DNSSEC records when supported (`RRSIG`, `DNSKEY`, `DS`, `NSEC`, `NSEC3`, `NSEC3PARAM`). We only generate `A`, `AAAA`, `TXT`, `CNAME`, `NS`, `SRV`, `SOA` and the special query type (QTYPE) `*` often called `ANY`.

## 2.7 Fuzzing Output and Processing

We now explain the collected data for each input and how we use it to determine if an input is interesting. The coverage information we gather is edge coverage given from an instrumented resolver. We compile the resolver using LLVMs Sanitizer Coverage Instrumentation [31] and count how often each edge gets executed during the resolution for a single DNS client query.

For each input we capture the outgoing queries from the resolver, the responses sent to the DNS client, the cache state afterward, how the dynamic DNS servers answered, and the coverage information. This gives us all the information about what the resolver is doing.

We use all available data to determine if an input is interesting, i.e., produces new or different behavior, but the most important information is the client's response and the coverage information. The client responses reveal whether two resolvers behave significantly differently. If different responses are returned two clients might behave differently. The coverage information is important to allow for partial progress during fuzzing, by giving a means of identifying new behavior in a resolver, even if the client response remains unchanged. The cache state is important too, but harder to interpret, since a difference here does not necessarily mean a semantic difference. Lastly, the outgoing queries to the dynamic DNS server have a low value, since comparing them is hard and has many downfalls. For example, there are many ways in which a resolver can implement Query Name (QNAME) minimization, such as which query type is used and which labels are removed. For a conforming DNS server, this does not matter and in the end, the resolver will come up with the same answer in all cases. But these differences are a problem for automatic analysis since we have many semantically equivalent queries with different representations.

The collected data is cleaned and checked for known differences, like random DNS ID values, to only leave the unknown differences. DNS records in messages are unordered, but we normalize the representations by sorting them. We also have further known differences, for example, some resolvers already support extended DNS errors which will be added to the client response in some cases. We create rules for identifying these known differences and track them separately, see Section 3.2. The remaining differences are unknown and of the highest interest.

Unfortunately, the fuzzing is not fully deterministic, so we require a validation run for any found new differences. Randomness and unwanted interactions can come from many sources, such as choices the resolver makes, how we batch multiple inputs together, or the kernel via the network and scheduler. We detect non-deterministic differences by fuzzing the same input multiple times and only accepting those results that show the same class of differences each time.

Many differences have an identical root cause and we group them using a fingerprint to better model that. Our fingerprint consists of all the DNS header fields in the DNS client response and all the field names that differ between the two resolvers. For example, we describe the difference in the answer section of the client response using identifiers like `.fuzz_result.fuzzee_response.answers.0` which has the subfields `.name_labels`, `.dns_class`, `.rr_type`, `.ttl`, and `.rdata`. The `.0` refers to the first resource record in the answer section. We do not use the values here, only the field names, since the values often contain randomized data and thus would always lead to different fingerprints. The same problem does not exist for the header values since these are mostly booleans.

### 2.8 Finding Bugs

Our main idea for finding semantic bugs is using differential fuzzing between many resolvers. DNS is a complex protocol with a lot of variability and edge cases. Defining valid behavior in the DNS is difficult as it requires a deep understanding of the standards and a lot of domain-specific knowledge. Instead,

**Table 1.** Software versions used for the evaluation.

| Software | Version | Language |
|---|---|---|
| BIND9 | v9.18.0, v9.11.0 | C |
| Deadwood | 3.5.0032 | C |
| Knot Resolver | 5.5.3 | C, Lua |
| PowerDNS Recursor | 4.7.3 | C++ |
| resolved | `463644c` | Rust |
| trust-dns | `0b6fefe` | Rust |
| Unbound | 1.15.0 | C |

using differential testing, we can automatically leverage this knowledge as the resolvers are implemented by independent expert groups. Instead of deciding if every response we see is valid, we now can focus on a much smaller set of cases where multiple resolvers disagree. Each difference is a case for further manual analysis as they can indicate implementation or semantic bugs.

During the manual analysis, we also built further rules to describe known differences. For example, BIND9 has a strict DNS response validation and discards the whole message if a single value is invalid, while Unbound will only discard the single invalid resource record. This causes BIND9 to produce a ServFail answer while Unbound responds with NoError or NoData depending on the situation. We go into more detail about this in Table 2.

The effort for manual analysis and writing rules ranges from a couple of minutes to a few hours. Simple cases, like DNS ID randomization or optional features like extended DNS error, are easy to spot and describe and have no follow-on impacts. Other cases, like the BIND9 strict validation, are more complex and require more time to understand and describe, as the root cause can lead to non-obvious and large differences. Concretely, BIND9 might send more upstream queries than other resolvers.

ResolFuzz is guided by the score each input receives. The score is composed of the coverage results per resolver and the differences for each resolver pair, after validation. New differences score the highest, followed by differences we deem "interesting"; coverage is only a small contributing factor. New mutations partially inherit the score of their parent inputs. Finally, if the input population already contains many samples producing the same fingerprint, we apply a penalty to all samples in the population. This is to discourage further mutations based on those samples. This ensures that the input population is diverse and ResolFuzz will not get "stuck" mutating a group of similar inputs with high scores and thus drowning out other interesting inputs.

## 3 Evaluation

We evaluate our fuzzer on seven resolvers, as listed in Table 1. Where possible we disable features during compile time, to reduce the size of the binary and simplify fuzzing. These features do not affect our evaluation, as these relate to system integration, such as systemd, enhanced security with chroot and capabilities, extra logging like dnstap, or HTTPS support. Most of these features have no impact at all on the DNS protocol and HTTPS for DoH is not used by ResolFuzz as UDP is more efficient for us. Each resolver is compiled from scratch using LLVM with coverage instrumentation [31], which is available in clang and rustc, as both are LLVM-based compilers. The containers are a Fedora 37 image.

## 3.1 Case Studies

In this section, we highlight our findings in three case studies.

**resolved fails with query and missing `CNAME`:** resolved misbehaves for `CNAME` queries if no `CNAME` record exists. Instead of returning the expected NoError response code, resolved returns ServFail. The `CNAME` record type redirects queries for a domain to another canonical domain. This means a resolver must follow the redirection, except here, because the original query is for the `CNAME` type. This means the first answer is the final one.

**PowerDNS Recursor self-loop:** We discovered a bug in handling `DNAME` RRs, similar to a known issue for `CNAME`s [6]. The problem occurs if the result after `DNAME` expansion matches the same `DNAME` again. `DNAME` and `CNAME` records both provide a way to specify a new canonical place where information is stored. In case of a `CNAME` RR like `this.old.domain CNAME new.canonical.name`, any query for `this.old.domain` should be redirected to `new.canonical.name` and the data from the new place should be used. A `DNAME` redirection is similar to `CNAME`. A `CNAME` only applies for a single specific domain, but a `DNAME` applies to a whole subtree. For example the `DNAME` RR `old.domain DNAME new.name` means that any query for *`this.`* `old.domain` redirects to *`this.`* `new.name`. The *italic* subdomain part is arbitrary and is preserved in the rewrite. The `DNAME` can point to itself, by having the re-written part match the pattern, for example, `old.domain DNAME` *`extra.`* `old.domain`, which will put *`extra`* many times in the domain. When PowerDNS Recursor encounters such a `DNAME` record, it applies the rewriting repeatedly. Consequently, in our setting, the resulting DNS answer will contain the same `DNAME` record 16 times. Further, each time the `DNAME` rewriting rule is applied, a synthetic `CNAME` record is created. Ultimately, the resolver gives up resolving this infinite loop and returns a ServFail with 32 records. The effect is that PowerDNS Recursor spends time following the loop and creating ever-increasing answers, thus wasting resources. The answer becomes large, making it a target for reflective DDoS attacks [25].

**Self-Sustained DoS in resolved and trust-dns:** We found a vulnerability in resolved and trust-dns that enables a self-sustaining DoS attack. The vulnerability is caused by these resolvers answering DNS responses (QR bit=1). When receiving responses on their listening port (UDP/53), both resolved and trust-dns return a FormErr error to indicate that the request was malformed.

Blindly responding to responses, even with error messages, can be abused for creating a traffic loop. If both resolvers are vulnerable, attackers can inject IP-spoofed responses to provoke the two resolvers to send responses to each other in an endless loop, causing a self-sustaining DoS attack. The traffic loop will not

**Table 2.** Higher level categories of the common differences we found between resolver outputs. Each category lists the number of rules that fall into it and gives an example.

| **Category** | **Size** | **Description** | **Example** |
|---|---|---|---|
| Configuration | 6 | Configurable behavior | Limiting the maximal TTL values, EDNS buffer sizes |
| Error Handling | 9 | Behavioral differences in error messages (ServFail, NotImp, FormErr, Refused) | On receiving a query with TC bit set, ServFail and FormErr responses exist. |
| Incomparable | 3 | Values that always differ | Random DNS ID. |
| Metadata | 3 | Metadata about the captured results | Resolver Name, redundant Length Values |
| Missing Features | 3 | Optional missing features | EDNS Error Codes |
| Resolver Specific | 10 | Other uncategorized, but resolver specific behavior | EDNS buffer size (512B in PowerDNS vs. 1232B in others). |
| Upstream Queries | 4 | Behavior of the upstream queries | QNAME Minimization, re-transmissions |

stop except when packet loss in the network causes the traffic to stop. Enough looping traffic will overwhelm the resolver or network and cause a DoS.

## 3.2 Result Statistics

The described case studies were found over multiple runs. We now describe how a *single* run of the fuzzer behaves and the kind of differences it finds.

The results of this section refer to a six-hour run of the fuzzer using all eight configurations listed in Table 1. During that time, 7140 inputs were generated or roughly 0.33/s. In total, the fuzzer made 198 240 comparisons, as each input takes part in 28 comparisons. This resulted in 1903 distinct fingerprints. The fingerprint count is larger than the number of root causes. For example, to trigger the resolved and trust-dns bug the only prerequisite is a query with QR=1 bit set, but a fingerprint includes all header fields.

We created 38 rules to describe benign differences, as shown in Table 2. The rules are grouped into seven categories. The nine *Error Handling* rules capture the variability of error messages (e.g., differences in error types). *Resolver-specific* behavior is similarly sized with ten rules, but the behavior covered here is more diverse. Two categories of differences will be found for any comparison of two outputs, *Metadata* and *Incomparable*. There are some fixed values about the captured results that are always different, such as the resolver names. In each comparison the DNS ID and edge coverage information are incomparable.

The 38 rules cover between 2017 to 6555 (28.2 % to 91.8 %) of the differences identified by the fuzzer, fluctuating based on the compared resolvers. Most similar is the pair of BIND9 (v9.18) and PowerDNS Recursor. Both are mature implementations and adhere to the DNS standard well. The worst pairing is Knot

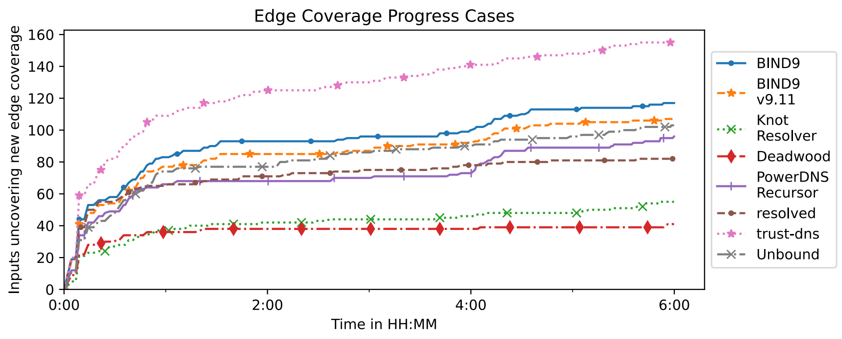

**Fig. 3.** The number of times new coverage edges are found over time.

Resolver and trust-dns. The trust-dns recursor is quite new and not yet well developed. As such it still has many missing features and bugs, all resulting in differences. These numbers translate directly into differences not covered by our rules. The pairing of Knot Resolver and trust-dns leads with 5067 (71.0 %) uncovered differences, while BIND9 and PowerDNS Recursor have only 490 (6.9 %). The gap towards the total of 7140 is caused by inputs where the comparison was non-deterministic. We observed the highest rate here between both BIND9 configurations with 143 (2.0 %) non-deterministic comparisons.

**Fuzzing progress:** We furthermore evaluate how much coverage our fuzzer achieves in the resolvers' code. This way we can understand when fuzzing saturates, i.e., no longer reveals new results. The coverage percentage increases sharply at the beginning, but then slows down but never stops. No resolver reaches a high edge coverage, with the highest being around 16.0 %. This can be explained by the fact that we only measure the coverage between sending a query and receiving a response. Any startup, background, or shutdown code is not included in the coverage. While we aimed to remove as many untested features as possible during compilation time, many features are still enabled but never activated by our fuzzer, such as DNSSEC, DNS forwarding or authoritative mode, and even complex features like Response Policy Zones (RPZs).

Figure 3 shows how often new edges are found over time. It shows more clearly, that during the entire period, progress is made. The main part falls into the first two hours, which is longer than the previous picture suggests. The continuous progress is a good indicator showing that even after a longer time ResolFuzz still makes progress and we have not yet reached the limits of it.

## 3.3 Bug and Vulnerability Disclosure

We reported all findings to the respective projects, except for one, which had no contact information. All projects acknowledged our findings. Most quickly fixed the issues, even releasing a security advisory RUSTSEC-2023-0041 [26], except for PowerDNS Recursor which deemed the risk as acceptable.

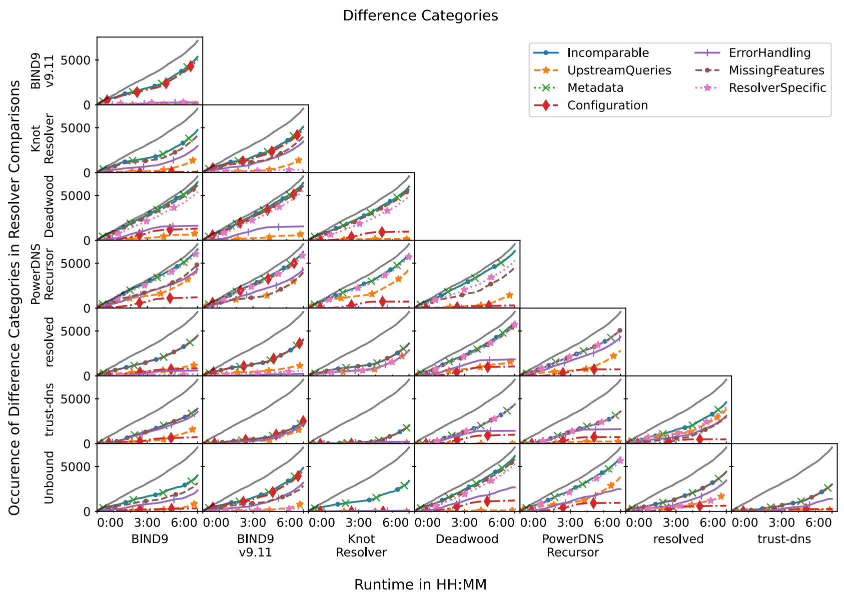

**Fig. 4.** The number of explainable differences between the resolver outputs. Only explainable differences are shown. The gray line shows the total number of fuzzing inputs tested. The gap between the gray line and the colored lines shows the number of unexplainable differences. The categories are explained in Table 2.

## 4 Limitations

We now discuss limitations that arise out of the design decisions we took to address the complex challenges of fuzzing DNS resolvers.

First of all, our fuzzer has no notion of time. Indeed, faking time jumps requires knowledge of the resolver internals as these determine when time jumps can be inserted and which parts they affect—internals we wanted to abstract from. The Deckard [7] testing framework managed to solve some of the problems and could be an inspiration for future work.

We use cache snooping as a generic mechanism to learn about the resolver's cache state. In some cases this fails, e.g., for specific query classes or resolvers, such that we skip the records during diffing. Resolver-specific ways to retrieve the cache status (e.g., via CLI tools or log files) would require resolver adjustments.

Furthermore, we did not implement all DNS features, such as DNSSEC [10, 22–24, 33]. Cryptographic operations are hard to fuzz, but may still represent interesting attack targets for malicious actors. Some part-way solutions are possible, like using a single validity bit and then dynamically signing the records, but we leave this for future work. Likewise, we ignored RPZ that can be used to block certain domains or IP addresses. They work by sending further queries and then changing or blocking the original request.

**Table 3.** Comparison of existing DNS testing tools, with a focus on fuzzing. The *Client* and *Server* columns indicate whether the project ships with a client or server component. *Search Strategy* indicates the strategy used for generating queries. Randomized means that all modifications are chosen by a random number generator, without any feedback. *Targets* indicates what the tool is looking for.

| **Project** | **Cli.** | **Srv.** | **Search Strategy** | **Targets** |
|---|---|---|---|---|
| ResolFuzz | yes | yes | Evolutionary mutations guided by code coverage and differential testing | Crashes and differential testing |
| Deckard [7] | yes | yes | None. Scripted communication. | Configurable checking. Comparison to fixed values. |
| dns-fuzz-server [27] | yes | yes | Randomized | None |
| dns-fuzzer [17] | yes | no | Randomized | Crashes |
| honggfuzz [29] | yes | no | Evolutionary, Code coverage feedback | Crashes |
| IBDNS [32] | no | yes | Fixed, based on query | None. Not a full tool. |
| nmap dns-fuzz [8] | yes | no | Randomized | Crashes |
| SCALE/Ferret [14] | yes | no | Statespace guided input generation | Diff. comparison between servers and formal model. |

We use a minimal configuration for each resolver, leaving most options as their upstream default. Configurations we changed include the IP address to listen, the root server hints, and the reduction of timeouts to speed up fuzzing. We deem this configuration realistic and modification necessary for fuzzing. We publish all our source code including configurations.

Recursive DNS resolution is the most complex part between clients, stub resolvers, and AuthNS. Stub resolvers could be tested similarly to the recursive resolvers, except they only forward the query to the recursive resolver.

## 5 Related Work

Automatic testing of network services has been subject to several other related works [2,8,11,14,16,17,19,20,27,29,32,34]. However, we are the first to create an advanced fuzzer for DNS resolvers. Indeed, DNS is a particularly challenging setting, as the resolver (i.e., the network service) acts as server and client at the same time—but most existing fuzzing frameworks test only single connections. Other projects solve DNS resolver fuzzing only partially. Related work can be grouped into two main categories: DNS testing and evaluation tools, and network fuzzers. We separately cover formal models for AuthNSes.

**DNS testing:** Due to the complexities in DNS and the difficulty of covering a large input space, only few projects exist for testing or fuzzing DNS. Table 3 provides an overview of existing projects and their capabilities. Existing tools often only target crashes in the DNS resolver [8,17,29]. This makes them unsuitable for finding more complex failure conditions, which we can identify with the differential testing approach. Some projects use basic randomization for input generation. dns-fuzz-server [27] only creates random queries and responses, but has no target conditions, like crashes, it is looking for. ResolFuzz uses a more advanced evolutionary algorithm, which is better suited for complex inputs.

The Deckard [7] project is noteworthy in that it is a full testing framework for DNS resolvers. It has extensive customization options, for query generations, mocking AuthNSes, and checking the responses. This extensiveness is great for writing detailed tests, but they often rely on the resolver implementation and are not portable between different resolvers. Relying on scripted tests also means unknown behavior cannot be revealed.

The Intentionally Broken DNS Server (IBDNS) [32] is a new project by Afnic for testing DNS resolvers or DNS tools. It applies known defects to existing zone files and serves them to clients. Its goal is to test DNS tools and DNS resolvers. IBDNS is not public so we cannot describe it in detail.

Other DNS test frameworks are either old and unmaintained [28], only test against a reference implementation [9], or are commercial with no public information [30]. Testing against a reference output can be useful for limited features or regression testing, but is not viable for covering larger input spaces, because of a missing reference implementation.

**Fuzzing Authoritative DNS Servers:** Most relevant to ResolFuzz are the projects by Kakarla et al. [13,14]. They tackle the challenge of fuzzing AuthNSes in the two papers GRoot [13] and SCALE [14]. In GRoot they lay the foundations by creating a formal model for the semantics of authoritative DNS. The model creates equivalence classes (EC) for a given DNS zone file. Each EC captures distinct behavior, like the difference between two different existing labels, and combines variants like queries that match the same wildcard record. With this model, they can symbolically execute these ECs and find bugs in the zones, like lame delegations, if the sub-zone has no reachable nameserver, or rewriting loops when `CNAME`/`DNAME` records form a loop leading to unresolvable names. SCALE builds an executable version of the GRoot model, with the ability to create zone files and matching queries. Using symbolic execution they find a wide variety of behaviors and create matching test cases using a constraint solver. That is an expensive process, so the created zone files are limited to four records. The test cases are fed into various AuthNSes and the responses are checked for compliance with the RFCs. The AuthNSes only agree in 35.0 % on the same answer. The rest is grouped by fingerprints, to make investigations easier. Using SCALE they could identify 30 new bugs.

**Network service fuzzing:** Apart from DNS with its special requirements, there are other network fuzzing approaches, which come in many variants.

AFLnet [19] is a greybox fuzzer. It has a corpus of network exchanges, which it mutates and sends to the target. AFLnet is guided by code coverage and learns a state machine for the target server. Fuzzing uses a forkserver, which allows for fast restarts and parallelization. SnapFuzz [2] is an iteration of AFLnet with increased performance, achieved with new binary rewriting. The rewriting replaces file system accesses with a custom in-memory implementation, replaces the TCP and UDP socket calls with UNIX domain sockets, and optimizes the forkserver. Both AFLnet and SnapFuzz are designed to fuzz single client-server connections, which makes them unsuitable for DNS resolvers.

Lin et al. [16] use GANs to infer the protocol of a black box network service. The GAN learns how to generate attack packets for a protocol, not only the protocol syntax. They only tested their approach on stateless protocols so far, which makes it unsuitable for DNS.

Hoque et al. [11] use a model checker for finding temporal semantic bugs in protocols. From a protocol implementation, they extract a finite state machine describing it. The temporal properties are protocol specific and require expert knowledge, in contrast to ResolFuzz.

Differential fuzzing has been used successfully in contexts other than DNS. TCP-Fuzz [34] by Zou et al. uses differential checking as one bug detection method. They develop an input creation strategy that keeps track of dependencies between packets and system calls. The coverage metric is based on the state transitions of the TCP stack. DPIFuzz [20] by Reen and Rossow uses differential fuzzing for QUIC. It generates QUIC frames, and mutates them with shuffling, duplication, and deleting data. While both works are related due to their differential checking approach, they are not directly comparable. TCP and QUIC work point-to-point and on a well-defined sequence of packets. A DNS resolver has many point-to-point connections and there is no clear sequence of packets, as each point-to-point connection runs independently.

## 6 Conclusion

Our analyses encourages more research on securing DNS resolvers, a critical part of the internet infrastructure. With ResolFuzz, our differential fuzzer, we found multiple security-relevant bugs in both well-established and new resolver implementations. DNS' lack of a formal model makes semantic analysis hard, but we showed that differential fuzzing with specialized matching rules can be a powerful tool to find bugs. We publish ResolvFuzz as open-source in the hope that it will help to improve the security of DNS resolvers. This work provides a starting point for further research into the security of DNS resolvers and we hope to uncover more bugs through improved insights into resolver decisions and covering more complex deployment scenarios.

**Data Availability** To ease reproducibility, we have released i) the full source code of ResolFuzz, ii) the build scripts and configuration files for the resolvers, iii) the raw data of our evaluation, iv) a list of found differences, and v) scripts for further analysis. The code and data are available online at https://github.com/dns-differential-fuzzing/dns-differential-fuzzing.

**Acknowledgements** Our sincere thanks belongs to the anonymous reviewers for their valuable feedback and suggestions which helped to improve the paper Furthermore, we thank the Saarbrücken Graduate School of Computer Science for their support.

## References

1. Andrews, M.P., Huque, S., Wouters, P., Wessels, D.: DNS Glue Requirements in Referral Responses. Internet-Draft draft-ietf-dnsop-glue-is-not-optional-08, Internet Engineering Task Force (Feb 2023), https://datatracker.ietf.org/doc/draft-ietf-dnsop-glue-is-not-optional/08/, work in Progress
2. Andronidis, A., Cadar, C.: SnapFuzz: high-throughput fuzzing of network applications. In: ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (2022). https://doi.org/10.1145/3533767.3534376
3. Cambus, F.: DNS related RFCs, https://www.statdns.com/rfc/
4. Consortium, I.S.: General DNS reference information, https://bind9.readthedocs.io/en/latest/general.html
5. Contavalli, C., van der Gaast, W., Lawrence, D.C., Kumari, W.A.: Client Subnet in DNS Queries. RFC 7871 (May 2016). https://doi.org/10.17487/RFC7871, https://www.rfc-editor.org/info/rfc7871
6. CVE-2022-48256. Available from MITRE, CVE-ID CVE-2022-48256. (Jan 2023), http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-48256
7. CZ.NIC: Deckard, https://gitlab.nic.cz/knot/deckard/
8. dns-fuzz in nmap, https://nmap.org/nsedoc/scripts/dns-fuzz.html
9. Ereche, M.V.: Dns completitude and compliance testing (Oct 2020), https://github.com/mave007/dns_completitude_and_compliance/tree/2a18967d103d232e9072c4474e8c731dc3d79f7a
10. Hoffman, P.E.: DNS Security Extensions (DNSSEC). RFC 9364 (Feb 2023). https://doi.org/10.17487/RFC9364, https://www.rfc-editor.org/info/rfc9364
11. Hoque, M.E., Chowdhury, O., Chau, S.Y., Nita-Rotaru, C., Li, N.: Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In: 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (2017). https://doi.org/10.1109/DSN.2017.36
12. IANA: Root files, https://www.iana.org/domains/root/files
13. Kakarla, S.K.R., Beckett, R., Arzani, B., Millstein, T.D., Varghese, G.: GRooT: proactive verification of DNS configurations. In: SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications (2020). https://doi.org/10.1145/3387514.3405871
14. Kakarla, S.K.R., Beckett, R., Millstein, T.D., Varghese, G.: SCALE: automatically finding RFC compliance bugs in DNS nameservers. In: 19th USENIX Symposium on Networked Systems Design and Implementation (2022)
15. Labs, N.: Unbound – RFC compliance, https://nlnetlabs.nl/projects/unbound/rfc-compliance/
16. Lin, Z., Moon, S., Zarate, C.M., Mulagalapalli, R., Kulandaivel, S., Fanti, G., Sekar, V.: Towards oblivious network analysis using generative adversarial networks. In: Proceedings of the 18th ACM Workshop on Hot Topics in Networks (2019). https://doi.org/10.1145/3365609.3365854
17. Meinke, R.: dns-fuzzer (Mar 2019), https://github.com/guyinatuxedo/dns-fuzzer/tree/6487b0053d9ee227b515490b9e00289b15a1bbd5
18. Mockapetris, P.: Domain names – concepts and facilities. RFC 1034 (Nov 1987). https://doi.org/10.17487/RFC1034, https://www.rfc-editor.org/info/rfc1034
19. Pham, V., Böhme, M., Roychoudhury, A.: AFLNET: A greybox fuzzer for network protocols. In: 13th IEEE International Conference on Software Testing (2020). https://doi.org/10.1109/ICST46399.2020.00062

20. Reen, G.S., Rossow, C.: Dpifuzz: A differential fuzzing framework to detect DPI elusion strategies for QUIC. In: ACSAC '20: Annual Computer Security Applications Conference (2020). https://doi.org/10.1145/3427228.3427662
21. RFC editor search DNS, https://www.rfc-editor.org/search/rfc_search_detail.php?title=DNS&page=All
22. Rose, S., Larson, M., Massey, D., Austein, R., Arends, R.: DNS Security Introduction and Requirements. RFC 4033 (Mar 2005). https://doi.org/10.17487/RFC4033, https://www.rfc-editor.org/info/rfc4033
23. Rose, S., Larson, M., Massey, D., Austein, R., Arends, R.: Protocol Modifications for the DNS Security Extensions. RFC 4035 (Mar 2005). https://doi.org/10.17487/RFC4035, https://www.rfc-editor.org/info/rfc4035
24. Rose, S., Larson, M., Massey, D., Austein, R., Arends, R.: Resource Records for the DNS Security Extensions. RFC 4034 (Mar 2005). https://doi.org/10.17487/RFC4034, https://www.rfc-editor.org/info/rfc4034
25. Rossow, C.: Amplification hell: Revisiting network protocols for DDoS abuse. In: 21st Annual Network and Distributed System Security Symposium (2014)
26. Remote attackers can cause denial-of-service (packet loops) with crafted dns packets, https://rustsec.org/advisories/RUSTSEC-2023-0041.html
27. Sakaguchi, T.: dns-fuzz-server (Sep 2019), https://github.com/sischkg/dns-fuzz-server/tree/6f45079014e745537c2f564fdad069974e727da1
28. Standcore: Standcore dns conformance, https://www.standcore.com/dnsconformance.tgz
29. Swiecki, R.: Honggfuzz bind9 (Nov 2020), https://github.com/google/honggfuzz/tree/37e8e813c9daa94dff29654b262268481d8c53ee/examples/bind
30. Synopsys: Dns server test suite data sheet, https://www.synopsys.com/software-integrity/security-testing/fuzz-testing/defensics/protocols/dns-server.html
31. The Clang Team: Sanitizercoverage, https://clang.llvm.org/docs/SanitizerCoverage.html
32. van der Wal, M.: Introducing ibdns: The intentionally broken dns server (Oct 2022), https://indico.dns-oarc.net/event/44/contributions/949/
33. Weiler, S., Blacka, D.: Clarifications and Implementation Notes for DNS Security (DNSSEC). RFC 6840 (Feb 2013). https://doi.org/10.17487/RFC6840, https://www.rfc-editor.org/info/rfc6840
34. Zou, Y., Bai, J., Zhou, J., Tan, J., Qin, C., Hu, S.: TCP-Fuzz: detecting memory and semantic bugs in TCP stacks with fuzzing. In: 2021 USENIX Annual Technical Conference (2021)