RESOLVERFUZZ: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing

Qifan Zhang[†], Xuesong Bai[†], Xiang Li[*✉], Haixin Duan[*§¶], Qi Li[*], and Zhou Li[†✉]

[†]*University of California, Irvine,* [*]*Tsinghua University*
[§]*Zhongguancun Laboratory,* [¶]*Quan Cheng Laboratory*

Abstract

Domain Name System (DNS) is a critical component of the Internet. DNS resolvers, which act as the cache between DNS clients and DNS nameservers, are the central piece of the DNS infrastructure, essential to the scalability of DNS. However, finding the resolver vulnerabilities is non-trivial, and this problem is not well addressed by the existing tools. To list a few reasons, first, most of the known resolver vulnerabilities are non-crash bugs that cannot be directly detected by the existing oracles (or sanitizers). Second, there lacks rigorous specifications to be used as references to classify a test case as a resolver bug. Third, DNS resolvers are stateful, and stateful fuzzing is still challenging due to the large input space.

In this paper, we present a new fuzzing system termed RESOLVERFUZZ to address the aforementioned challenges related to DNS resolvers, with a suite of new techniques being developed. First, RESOLVERFUZZ performs constrained stateful fuzzing by focusing on the short query-response sequence, which has been demonstrated as the most effective way to find resolver bugs, based on our study of the published DNS CVEs. Second, to generate test cases that are more likely to trigger resolver bugs, we combine probabilistic context-free grammar (PCFG) based input generation with byte-level mutation for both queries and responses. Third, we leverage differential testing and clustering to identify non-crash bugs like cache poisoning bugs. We evaluated RESOLVERFUZZ against 6 mainstream DNS software under 4 resolver modes. Overall, we identify 23 vulnerabilities that can result in cache poisoning, resource consumption, and crash attacks. After responsible disclosure, 19 of them have been confirmed or fixed, and 15 CVE numbers have been assigned.

1 Introduction

Domain Name System (DNS) is central to Internet activities, translating human-friendly domain names to machine-friendly IP addresses. When a client user issues a DNS query to an authoritative server with the answers, a DNS *resolver* is often encountered on the resolution path, which provides caching service that is essential to the scalability of DNS infrastructure [26]. Numerous resolvers, including public resolvers like Google DNS and local resolvers set up by ISPs, have been deployed [1], running various DNS software (e.g., BIND and Unbound).

Yet, despite decades of development of DNS infrastructure, resolver vulnerabilities are still continuously uncovered, with some causing severe damage if exploited by attackers. For example, by sending just one client query containing `RRSIG`, the attacker can crash a BIND resolver entirely (CVE-2022-3736). As another example, CVE-2022-2881 could be exploited like the infamous TLS heart-bleed vulnerability to read sensitive memory data. We believe new tools should be developed to effectively uncover resolver vulnerabilities, in order to secure the DNS infrastructure.

Understanding resolver vulnerabilities. As the first step, we try to understand the characteristics of resolver vulnerabilities by mining the published CVEs (Section 2.2). By carefully examining *239* CVEs published from 1999 to 2023 about 6 mainstream resolvers, we found a lot of them are *semantic* bugs that violate high-level rules or invariants [66], leading to cache poisoning [30], resource consumption, etc. Detecting such bugs is still challenging as they usually do not trigger software crash, which is the major target of the existing fuzzers like AFL [14]. Though software fuzzing can be applied to test resolver software, generating meaningful DNS messages is non-trivial due to the complex structure (Section 3). Moreover, DNS resolver runs a *stateful* caching service, but stateful fuzzing has always been a major challenge in network fuzzing [7], due to computational complexity in covering a large state space.

Query-response fuzzing for resolvers. To address the aforementioned challenges unique to DNS resolvers, we develop a new fuzzing system termed RESOLVERFUZZ (Section 4). To accommodate resolver software that is built under different

✉ Corresponding authors. Most of Xiang Li's work was done when visiting UCI as a project specialist.

programming languages and models, we choose *blackbox* fuzzing and monitor the status of a resolver with lightweight tools like cache dump and tcpdump, without code recompilation or binary rewriting [14]. The main insight guiding our design of RESOLVERFUZZ is that a *short* message sequence (e.g., one query from the client and/or one response from the nameserver) is sufficient to trigger a large number of resolver bugs, as revealed from our CVE study, so RESOLVERFUZZ performs *constrained* stateful fuzzing by only mutating a pair of query and response. To generate test cases that are likely to be accepted by the resolvers, we perform grammar-based fuzzing, by generating test cases with *probabilistic context-free grammar (PCFG)* [24], and augmenting the test cases with byte-level mutations. Given that the existing oracles are unsuited to detect the non-crash bugs specific to resolvers, we develop new oracles to detect cache poisoning and resource consumption. Detecting cache poisoning is particularly challenging, due to the lack of rigorous DNS RFCs to strictly define the canonical behaviors of caching. We address this issue by performing *differential testing* [40] and use the cache inconsistency to find potential vulnerabilities. Still, inconsistent behaviors are pervasive among resolvers [62], and many of them are not related to vulnerabilities. Hence, we develop a new bug triaging method based on *bisecting K-means* to cluster the inconsistent test cases, so the manual investigation efforts are greatly reduced. Finally, to increase the fuzzing throughput and avoid affecting the remote DNS nameservers, we build a new test infrastructure that *localizes* the nameserver hierarchy and enables concurrent resolver testing.

Evaluation. We have generated over 700K test cases towards 4 resolver modes (recursive-only, forward-only, CDNS with fallback, and CDNS without fallback) (Section 5). The evaluation results show the test generator has good coverage of valid DNS messages and the oracles are effective in pinpointing resolver bugs. We have discovered *23* vulnerabilities with RESOLVERFUZZ (Section 6). With responsible disclosure, 19 of them have been confirmed or fixed, and 15 CVEs were assigned. We even discovered a very powerful bug (*CP*1 in Section 6.1) that can entirely bypass the bailiwick checking rule, and poison any domain in a TLD zone (e.g., any `.com` domain can be compromised after the bug is exploited). The extended version can be found at [72].

Contributions. Our contributions are summarized below.

- We conduct a comprehensive study of DNS CVEs.
- We develop a new blackbox fuzzing system RESOLVERFUZZ, based on our insights from the CVE study. It performs constrained query-response fuzzing for efficient bug discovery on resolvers.
- We develop and/or adjust a set of techniques, including DNS localization, PCFG-based test generation, differential testing, etc., for RESOLVERFUZZ.
- We evaluate RESOLVERFUZZ against 6 mainstream resolvers and uncover 23 bugs. We discuss our findings with software vendors and received acknowledgment.
- RESOLVERFUZZ is open-sourced [55].

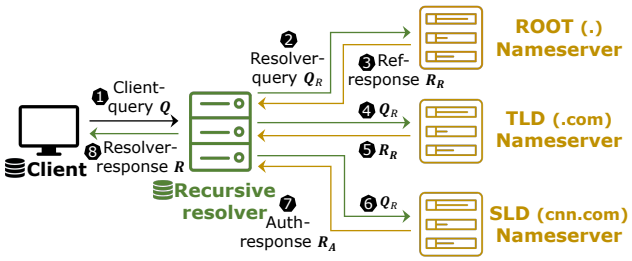


Figure 1: Example of DNS resolution process. "ref-response" and "auth-response" are both considered as "ns-response".

2 Background

2.1 DNS and Resolvers

DNS translates a user-friendly domain name to a numerical IP address. A domain name is written as a sequence of labels separated by ".", e.g., `cnn.com`. To resolve a domain name, a DNS client (or *stub resolver*) usually issues a DNS query to a public DNS (e.g., Google Public DNS or a local recursive resolver (e.g., Comcast ISP resolver), and lets the resolver contact nameservers iteratively. Figure 1 illustrates the process for resolving `www.cnn.com`, during which the nameservers of root (denoted by "."), Top-Level Domain (TLD) `.com`, and Second-Level Domain (SLD) `cnn.com` are contacted. They answer the queries with *resource record sets (RRSets)* in their *zone* configurations.

The format of the DNS query and response follows RFC 1034 [45]. In essence, the query and response share the same set of *sections*, including "Flags", "Question", "Answer", "Authority", and "Additional". "Flags" signals the kind of message (e.g., `QR` represents response and `AA` represents authoritative answer). "Question" encodes the domain name to be resolved and the type of RRSet to be retrieved (e.g., `A` represents IPv4 address, `AAAA` represents IPv6 address, and `NS` represents nameserver domain). When the contacted nameserver has the authoritative answer, the "Answer" section of the response message encodes the requested RRSet. Otherwise, within the response, "Authority" fills *referral* name that is "closer" (e.g., `.com` nameserver is closer to root server in answering queries about `example.com`) to give the authoritative answer, and "Additional" fills *glue records* about the server addresses. Figure 8 shows examples of DNS messages.

In this paper, we name the query from DNS client to resolver as *client-query*, the query from resolver to nameserver as *resolver-query*, response from nameserver to resolver as *ns-response*, response from resolver to client as *resolver-response*. For ns-response, it is classified into *ref-response*

which has the referral record, and *auth-response* which has the authoritative answer.

Resolver modes. The resolver is central to the resolution procedure, which also caches the responses to reduce query latency. The standard mode of resolver runs recursive resolution and contacts nameservers. Alternatively, the resolver can run as *forwarder*, which passes the query to other servers (e.g., upstream recursive resolvers). A resolver can also run the recursive and forwarder mode *concurrently*, with each mode handling a subset of DNS namespace, and such resolver is called *conditional DNS* (CDNS) [3]. For some resolvers, a *fallback* option [9] can be enabled, to reissue the query in the recursive mode when forwarding the query fails.

2.2 Study of DNS CVEs

Here we perform a comprehensive study to understand the distribution and root causes of DNS-related vulnerabilities, which also guides the design of ResolverFuzz. We crawl Common Vulnerabilities Exposures (CVE) databases [42–44] and mainly analyze the CVE reports of six mainstream, open-sourced DNS software, including BIND, Unbound, Knot, PowerDNS, MaraDNS, and Technitium. We elaborate how we analyze CVE reports in Appendix A.

Table 1 lists our study results of DNS CVEs related to resolver modes, and the CVE dates range from 1999 to 2023. We summarize our key findings ($F1$ to $F5$) below.

- **F1: Most of the CVEs are about resolvers.** In total, we identified *291* CVEs related to the 6 studied DNS software (*132* CVEs are related to other DNS software). Among them, *245 (84%)* are about resolvers (e.g., CVE-2019-6477 and CVE-2020-8621 for the recursive and forwarder mode). Only 46 CVEs are about nameservers (e.g., CVE-2020-8619 and CVE-2017-3143).
- **F2: Diversified CVEs among DNS software.** Though BIND dominates in the number of CVEs[1], a prominent number of CVEs have also been found in other software (except Technitium). Moreover, we found *only 13* CVEs among the 245 CVEs affect all software (e.g., NXNSAttack under CVE-2020-12662), suggesting the diverse implementations of DNS software.
- **F3: A significant portion of CVEs are not related to crash.** Like the results from a prior work that studies CVEs in TCP stacks [74], we found the bugs that do not trigger software crash constitute a prominent portion (*109* out of 245 CVEs). The main consequences include cache poisoning (*46* CVEs, e.g., caching illegal records under CVE-2002-2213 and CVE-2006-0527) and resource consumption (*39* CVEs, e.g., spending excessive resources to handle DNS queries under CVE-2022-2795 and CVE-2021-25219). For the 136 crash-related bugs, only 43 are caused by memory corruption such as buffer overflow under CVE-2020-8625 and CVE-2021-25216. Others are mainly triggered by assertion failures (e.g., CVE-2022-0635 and CVE-2022-3080).
- **F4: Nearly every field of a DNS message has related CVEs.** Examples include `query name` (CVE-2020-8617), `query type` (CVE-2022-0667), `query flag` (CVE-2017-15105), `rcode` (CVE-2018-5734), `rdata` (CVE-2013-4854), `TTL` (CVE-2003-0914), etc.
- **F5: Most of the CVEs are triggered with a very short message sequence.** We found *222/245* (91%) CVEs could be triggered by sending *just one client-query or ns-response*. One such example is CVE-2022-3736. For the other CVEs that require longer sequences (e.g., CVE-2022-3924), many client-queries are needed to trigger the bugs. We show their details in Appendix A.

With the above insights, we design ResolverFuzz and elaborate the design choices in Section 3. We acknowledge that our CVE study could suffer from survivorship bias, and we discuss this issue in Section 7.

2.3 Prior Tools for DNS Bug Discovery

Here we survey the related tools that can automatically detect DNS bugs. In Section 8, we survey systems that can detect other network vulnerabilities.

First, we found fuzzing has been applied to test DNS resolvers. SnapFuzz aims to achieve high throughput in fuzzing network applications [4]. It rewrites the tested program for greybox fuzzing and fast asynchronous communication. It was evaluated against a lightweight DNS software Dnsmasq that is usually deployed on routers. 7 crashes were detected within 24 hours. However, it cannot directly detect non-crash bugs. DNS Fuzzer performs byte-level mutation by inserting new bytes into seed DNS queries [17]. Similar to SnapFuzz, it only detects crashes. As far as we know, the most related tool to ResolverFuzz is dns-fuzz-server [63], which performs grammar-based and byte-level mutation on queries and responses. In Section 5.2, we show detailed comparison.

In addition to resolvers, DNS *nameservers* have also been found vulnerable when the zone files installed by the domain owners have mis-configurations. A number of tools were developed to find such mis-configurations [49, 56] . Recently, formal methods have been applied by checking the DNS configurations with formal specifications. G-Root performs formal verification to prove the correctness of configurations and find counterexamples [28]. SCALE jointly generates zone files and corresponding queries that are specified by RFCs to discover implementation inconsistencies [29]. Yet, for DNS resolvers, there lacks rigorous specifications to be used as references for vulnerability discovery [46, 65].

[1]The high number of CVEs of BIND does not necessarily indicate it is more vulnerable. In fact, BIND has the largest market share [31] and has been extensively tested [32].

Table 1: Study results of DNS CVEs for mainstream DNS software.

Software*	# CVE							
	Non-crash				Crash			Total
	Cache Poisoning	Resource Consum.[1]	Others[2]	Total	Non-memory	Memory	Total	
BIND	18	18	11	47	75	22	97	144
Unbound	4	5	4	13	5	8	13	26
Knot Resolver	6	4	0	10	2	0	2	12
PowerDNS Recursor	13	8	9	30	7	6	13	43
MaraDNS	2	3	0	5	4	7	11	16
Technitium	3	1	0	4	0	0	0	4
Total	46	39	24	109	93	43	136	245

*: Recursive or forwarding modes. [1]: Resource consumption.
[2]: An example of other non-crash bugs: CVE-2018-5738 that improperly permits recursion to all clients.
CVE of the forwarding mode only (7 in total): BIND (5), Unbound (0), Knot (1), PowerDNS (0), MaraDNS (0), and Technitium (1).
CVE of the authoritative mode only (46 in total): BIND (19), Unbound (4), Knot (2), PowerDNS (20), MaraDNS (1), and Technitium (0).
CVE of other software (132 in total): Microsoft DNS (90), Simple DNS Plus (1), Dnsmasq (34), CoreDNS (1), NSD (4), Yadifa (1), and TrustDNS (1).

3 Overview of ResolverFuzz

3.1 Problem Definition and Challenges

General threat model and targeted vulnerabilities. We consider a public or local recursive resolver to be targeted by the attacker. The attacker is able to control a downstream DNS client and/or an upstream nameserver on the resolution path of the resolver. Hence, DNS queries and responses in arbitrary format can be issued against the resolver. We consider 4 types of vulnerabilities as they are related to most CVEs (see Table 1) and overview them below. In Section 6, we elaborate the threat model for each type.

- **Cache poisoning.** The attacker tampers resolver's cache and directs victim clients to malicious servers.
- **Resource consumption.** The attacker heavily consumes resolver's resources to impact its service quality.
- **Non-memory crash.** The attacker terminates a resolver without memory corruption, e.g., by sending DNS messages to execute code with assertion failures.
- **Memory crash.** The attacker's DNS messages corrupt the resolver memory and terminate the resolver.

To notice, side-channel vulnerabilities are out of scope of this work, as these vulnerabilities often exist at the layers *below* DNS. In Section 8, we review them under the theme of off-path cache poisoning attacks.

Design goals and challenges. ResolverFuzz aims to uncover the vulnerabilities under the aforementioned threat model. We focus on four types of vulnerabilities including cache poisoning, resource consumption, service crash, and memory corruption, as our survey in Section 2.2 suggests they are the major issues against DNS software. ResolverFuzz should be *efficient* in testing resolvers at high throughput. Moreover, ResolverFuzz should be able to tell whether the test inputs could lead to vulnerability discovery at high *accuracy*. We encounter a few key challenges towards meeting these goals:

- **C1: Efficiency.** Notable latency is expected for a regular DNS resolution, as network communications are needed between the client, resolver, and multiple nameservers. Hence, achieving high throughput for resolver fuzzing is not trivial.
- **C2: Mutation.** The widely used greybox fuzzers like AFL [14] mutate the input with coverage-based metric. However, such metric does not provide sufficient guidance on which *part* of the testing input should be mutated [51], but DNS messages contain many fields that are related to bugs (**F4** in Section 2.2).
- **C3: Stateful fuzzing.** Different from nameservers that run in a stateless mode, resolvers are *stateful* [27], whose states depend on cache records, configurations, etc. Stateful services have been considered a major challenge for network fuzzing [7], due to the large search space of input sequences.
- **C4: Oracle.** Non-crash bugs have a large share in resolver CVEs (**F3** in Section 2.2). However, there lacks an oracle to detect such bugs. Instead, crash bugs can be detected by oracles like AddressSanitizer [59]. Differential testing has been used to uncover semantic bugs that are non-crash, but none of the prior works built the oracle for DNS. Moreover, our empirical analysis suggests inconsistencies among DNS resolvers are common, and many of them do not indicate vulnerabilities. In Section 5.2, we show an example of normal inconsistencies.

3.2 Workflow of ResolverFuzz

ResolverFuzz addresses the aforementioned challenges with 3 newly designed components, which are elaborated in

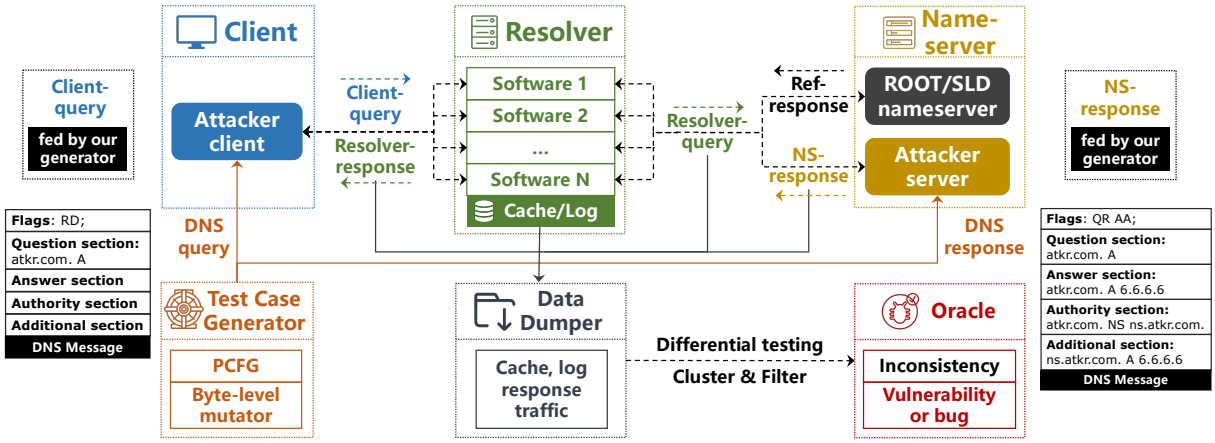


Figure 2: Workflow of RESOLVERFUZZ.

Section 4. The workflow of RESOLVERFUZZ is also illustrated in Figure 2. In the first stage, the testing infrastructure (Section 4.1) loads the resolvers of different implementations and configures the environment. A number of optimization techniques are applied to increase the throughput of DNS queries and responses, addressing **C1**. Then, the tests generator (Section 4.2) performs grammar-based mutation on client-queries and ns-responses, addressing **C2**. Though resolver is stateful (**C3**), according to our study on CVEs (**F5** in Section 2.2), triggering a bug in most cases only requires a short sequence of one client-query and/or one ns-response. Hence, we simplify stateful fuzzing to simulate short sequences. Finally, for data collected by our data dumper, including the cache, log, response, and traffic, we develop different oracles to detect different types of vulnerabilities mentioned in Section 3.1. We apply differential testing to identify inconsistencies among resolvers to capture cache poisoning bugs. To address **C4**, we develop a new method to cluster inconsistencies and alert on the abnormal ones, so the manual efforts in bug investigation will be significantly reduced.

4 Design of RESOLVERFUZZ

4.1 Testing Infrastructure and Optimization

We design a new infrastructure to simulate DNS queries and responses against resolvers and collect the traces to be later analyzed. The infrastructure mainly includes a DNS client and a nameserver that emit messages generated by our fuzzer (named as *attacker client* and *attacker server*), a test scheduler, and a trace collector. Below we first describe these components and their network setup. Then, we overview the whole testing process.

Attacker client and server. Instead of using the off-the-shelf DNS software, we implement the attacker client and server with Python scripts. As such, we are able to send *arbitrary* queries and responses that can even be incompliant with DNS RFCs [45] and reduce the processing latency with lightweight implementations. Specifically for the attacker server, though the standard implementations require a zone file to be hosted to answer the queries with the contained records, we choose to directly generate an ns-response given a client-query and skip the zone file (details are in Section 4.2). We notice that related systems like SCALE [29] reuse the existing DNS software for nameservers and mutate the zone files for testing purposes. Our customized implementation is more flexible in response generation, and even allows low-level manipulation of DNS responses (e.g., answering DNS queries with UDP or TCP).

Test scheduler. This central component initializes DNS components, including the attacker client, attacker server and resolvers. Between each round of test (i.e., one round-trip of DNS resolution), the scheduler resets these components. For efficient and complete resetting, we choose to host these components with lightweight Docker containers [41]. To increase the testing throughput, the scheduler will command the client to send queries to the resolvers *in parallel* and replicate the resolver instances. Specifically, the scheduler groups an attacker client, an attacker server, and resolvers into a *unit*, and runs multiple units concurrently (with different test cases). With container-based isolation, each resolver instance can be tested independently.

Resolvers and analyzed data. To detect the 4 types of vulnerabilities described in Section 3.1, we install a set of monitors inside each resolver container. First, we use a set of tools to export the resolver cache from memory to files (see Section 5.1 for details of the tools) to detect cache-related bugs. Second, we use `tcpdump` to collect the incoming and outgoing network traffic to detect bugs related to resource consumption. Third, we collect the log files generated by the DNS software. Finally, we also monitor the running status of the resolver process to detect service crashes and memory corruption with Linux command `ps`.

In addition to testing the standard recursive mode (termed *recursive-only*), we also test 3 alternative resolver modes:

forward-only, *CDNS without fallback*, and *CDNS with fallback* (explained in Section 2.1). We are motivated to test different modes due to that certain modes, e.g., forward, are more vulnerable as uncovered by previous work [73]. Changing a resolver from one mode to another can be easily done by loading a different configuration file into the resolver container. To notice, we did not test all resolver configurations: e.g., we disable DNSSEC since we found it introduces a large number of inconsistencies among resolvers that are irrelevant to bugs. We discuss this limitation in Section 7.

Network configurations. All the Docker containers are connected to a bridged Docker network assigned with /16 private IP addresses. Under a standard DNS resolution between the client and the nameserver of a registered domain, remote nameservers like root and TLD servers have to be contacted, so DNS round-trips could incur notable latency. Moreover, our tests could trigger bugs on those remote servers, raising ethical concerns. To address these issues, we choose to *localize* the nameservers between the attacker client and attacker server in our lab network. We implement a server to simulate all these nameservers. Each client-query asks about a domain name owned by us, and we use different subdomains to separate the test cases of different resolver modes.

Testing process. 1) The test scheduler initializes the Docker network and n units of simulated DNS infrastructure. 2) The test scheduler obtains test cases from the generator and dispatches them to the attacker client and server. 3) After the start of each resolver container, the internal monitors collect the information about cache dump, network traffic, and process information. 4) When the attacker client receives the resolver-response, the resolver containers will be reset via cache flushing or software restarting, and the corresponding unit will go back to step 2 until all tests are completed.

4.2 Test Case Generator

We generate the test cases from *two* dimensions, including client-queries for our attacker client and ns-responses for our attacker server. Considering a DNS resolver as a stateful service, the ns-response is generated corresponding to a client-query. Though we only simulate a short message sequence (one client-query and one ns-response), we find it sufficient to uncover a variety of bugs, as suggested by **F5** in Section 5.2.

Though we can freely generate an ns-response disregarding the questions embedded in the client-query, our empirical analysis suggests such responses are often quickly dropped by the resolver. As a result, we restrict the ns-response to contain most of the fields (e.g., the "Question" Section and TxID) from the client-query and only add information to sections like "Answer", which significantly reduces the input space. Though under standard DNS resolution, the nameserver generates a response *after* seeing the query, we found this process can be optimized by generating the pair of ns-response and client-query *simultaneously* and dispatching them to the client and nameserver before the resolver is queried[2].

Given that DNS messages have a complex structure, we apply *probabilistic context-free grammar (PCFG)* [24] to generate templates of client-queries and ns-responses first. A PCFG consists of a start symbol (denoted `<start>`), non-terminal symbols (symbols surrounded by <>), terminal symbols, production rules, and rule probabilities. We assign high probabilities to certain fields after analyzing CVEs, so the fuzzing process can be directed towards the code regions that are critical but error-prone[3]. A DNS query is allowed to ask one or more questions (e.g., domain name) in the "Question" section, and a response can contain multiple answering records in the "Answer", "Additional", and "Authority" sections. However, letting the generator add an arbitrary number of questions or answering records will introduce a very large input space, which is also unlikely to trigger new bugs [4]. Hence, we restrict the number of each section to range from 0 to 5 (for "Question", only 1 `QNAME` is inquired). We also force the counting field to contain the correct number of records (e.g., `ANCOUNT` contains the right number of records in "Answer"). Field formats and additional rules are applied to fields like `QNAME` to ensure their validity. In Appendix B, we list the complete PCFG.

Similar to previous work in fuzzing network services [20, 21], we perform byte-level mutation on the PCFG-generated message templates. We are motivated to do so for DNS as recent work showed some DNS implementations fail to correctly decode strings with special characters embedded [22]. We consider mutating each terminal symbol of PCFG with special bytes, such as `\.`, `\000`, `@`, `/`, and `\` (the first four characters are also exploited by [22]). The mutation operators include byte addition, deletion, and replacement. We assign a probability to determine when the mutation should happen. We set the probability to conduct byte-mutation on a PCFG output as 0.1.

4.3 Data Dumper and Oracle

We first dump the traces collected from the resolver containers into the host and conduct data pre-processing. Then, we design 3 oracles to detect the 4 types of resolver bugs.

Data dumper. Several previous works applied blackbox fuzzing on web services [20, 21] by only using information from the requests and responses. Though we also follow the direction of blackbox fuzzing without instrumenting and re-compiling the targeted resolvers, we collect information in addition to requests and responses, including cache dump, resolution traffic, software logs, and process status, to achieve

[2] A client-query can trigger a resolver to process multiple resolver-queries and ns-responses. In this case, we just use the same ns-response.

[3] The code regions of our interests are identified by analyzing CVE reports and reproducing CVE PoCs.

[4] For example, we reviewed the source code of BIND and found it rejects the query with multiple questions. In response, each answer is matched with the question but the number of answers does not impact the logic.

better coverage of bugs. The formats of cache dumps and software logs vary for different resolver software, so we convert them to unified formats with the methods described below.

For cache dumps, we define a new cache structure that uses the cached domain names as the key and common record fields including `class`, `type`, `ttl`, and `rdata` as the values. An alternative approach to cache dump is cache snooping [16], which infers the cache status of one record per query. We choose cache dump because 1) it provides more information like where the cached records come from and cache trust levels and 2) it is also more efficient. For software logs, internal operations and their parameters are usually recorded, which could be utilized to detect abnormal behaviors during resolution, e.g., excessive cache searching. We implement a pattern-matching method to search for log entries of our interests, and categorize them under keys such as `CACHE LOOKUP`, `QUERY`, and `SANITIZE RECORD`. For each test case, we assign the pre-processed cache dump, software logs, network traffic, and process status from all resolvers to it.

Cache poisoning oracle. We design this oracle based on the insights that the cache poisoning attack tampers the cache storage and usually *inserts* forged records to hijack victim domains [30], so the cache records are likely to differ among the resolvers if some are vulnerable. We run *differential testing* to find the cache anomalies. A number of previous work on differential testing rely on a *"golden model"* to compare against. For example, DIFUZZRTL detects bugs in RISC-V CPU cores by comparing their RTL execution results with a golden model OpenRISC Or1ksim [19]. However, we cannot find a golden model for DNS resolvers: even the most widely used resolver BIND has more than 100 CVEs reported. Therefore, assuming the set of software studied by us is $\mathcal{S}$, for a software $s_i \in \mathcal{S}$, we consider s_i is abnormal when its trace differs from any $s_j \in \mathcal{S} \setminus \{s_i\}$.

Specifically for cache, for each test case, we check whether the cache records for all the resolver software are the same, by comparing the records' `NAME`, `TYPE`, and `RDATA` fields. After this stage, we found there are still many test cases with inconsistent cache records based on our evaluation, so we perform another round of bug triage by *clustering* the test cases. We represent $s_i \in \mathcal{S}$ of each test case with the maximum number of different records of the software i with other software. For instance, $< 0,0,0,5 >$ means that software 1 to 3 has the same cache, but software 4 has 5 additional cache records. Then, we apply *Bisecting K-Means*, which outperforms the basic K-Means in entropy measurement [60], on the cache vectors to generate clusters and investigate each cluster to look for vulnerabilities. Clustering is done for test cases under each resolver mode separately.

For the vulnerability analysis, we employ a semi-automatic method. By extracting information from the fields like `NAME`, `TYPE`, `RDATA`, `ZONE`, etc., we create matching rules and use them to separate test cases within a cluster into sub-clusters iteratively. For example, the sub-cluster for bug CP2 described in Section 6.2 is generated based on the existence of the NS record of the domain in the forwarding zone. Then each sub-cluster is manually analyzed to confirm if it is related to vulnerabilities: we randomly sample 1 test case per sub-cluster and try to construct exploit. Though our vulnerability analysis can be done without clustering, we found the investigation overhead is significantly reduced after clustering.

Resource consumption oracle. Previous studies show that attackers have the incentive to disrupt the operation of resolvers or use the resolver to conduct DNS amplification attacks against other servers. We measure the resource consumption with 4 metrics, derived from the resolver's network traffic and software logs, including the number of resolver-queries, the sizes of responses (both ns-response and resolver-response), the resolution timeout, and the frequency of internal operations (e.g., cache search).

For a metric (say m_j), we compute its value distribution within the *same* software (say s_i), and consider a test case abnormal if s_i's value on m_j falls out of the normal range. Specifically, we represent the value distribution as a Cumulative Distribution Function (CDF), and consider the normal range as $[0, \theta]$, where θ is the threshold and we set it to 0.9. For instance, if the frequency of the cache search operations of one test case is higher than 90% of all test cases in CDF, this test case is considered abnormal. Then, similar to cache oracle, we perform an iterative process of random sampling and manual investigation (the clustering stage is skipped as no differential testing is conducted here).

Crash oracle. To detect bugs related to memory and non-memory crash, the resolver container simply monitors the resolver process and considers anomaly happens when the process is not running (i.e., not included in the output of `ps` command). We acknowledge that this oracle is simple and false negatives can happen (e.g., dangling pointers might not trigger a crash). Though more complex oracles like AddressSanitizer [59] can detect more types of memory bugs, they often require re-compilation, which is incompatible with our blackbox fuzzing setting.

5 Evaluation and Results

5.1 Implementation Details

For the tested resolvers, we choose the ones listed in Table 1, and their versions are all latest during our evaluation period (BIND: 9.18.0, Unbound: 1.16.0, Knot Resolver: 5.5.0, PowerDNS Recursor: 4.7.0, MaraDNS: 3.5.0022, and Technitium: 10.0.1). For the testing infrastructure, the attacker client and attacker server generate customized DNS messages using Python language. When running the experiment, we force the reset of a resolver container if it does not respond before a 5-second timeout. The test scheduler manages Docker containers with Python Docker SDK. Each software is compiled based on Ubuntu 22.04 Docker image. Within the re-

solver containers, we are able to dump the cache from 4 software with existing tools or supported APIs: `rndc` for BIND, `unbound-control` for Unbound, `rec_control` for PowerDNS, and Technitium's HTTP API. We are unable to dump or decode the cache from MaraDNS and Knot, so we evaluate them by replaying the tests that are proven to impact other software on them. For the other nameservers in the resolution change, we write them with the Go language (for better performance) and configure their zone files to make our attacker server reachable.

For the tests generator, we use different attacker domain names (all starting with `test-`) to test different resolver modes. For the oracles, we use a Python library scikit-learn to implement the clustering method.

We write in total of 3,649 lines of code (LoC) in Python for the scheduler, tests generator, attacker client, and server. We also write 318 LoC in Go and 203 LoC in JSON file for the other nameservers. We use one workstation to run ResolverFuzz, which has an AMD 5950x CPU with 16 cores, 128 GB memory, and runs on Ubuntu 22.04.

5.2 Experiment Results

In total, ResolverFuzz generates *718.6K* test cases (each case consists of a pair of query and response) within 65.9 hours. The testing inputs are evenly distributed among the 4 resolver modes. The traces collected from the resolver containers occupy 1,892.9 GB of disk space. Below we first describe the analysis results and the runtime performance of ResolverFuzz. Then, we compare ResolverFuzz with the other DNS fuzzers and conduct an ablation study to assess the impact of several design choices. Finally, we conduct a large-scale scanning to discover the open resolvers that are impacted by our discovered vulnerabilities and present the results in Appendix C.

Results from the oracles. The oracles for resource consumption and crash & corruption are relatively simple (shown in Table 2). Here we focus on the cache-related oracle[5]. Among the 718.6K testing case inputs, our differential analysis filters out *461.2K* (64.2%) inputs that trigger identical behaviors on the tested resolvers. The large ratio of the remaining inputs indicates there is a great variety among the resolver implementations, and an inconsistency usually does not imply vulnerability. For example, there are 69,967 testing cases with legitimate differences related to `NSEC3` records. `NSEC3` records [5] are used to indicate a non-exist domain in a secured way, preventing malicious actors from sending fake negative responses to queries. We find that `NSEC3` records are cached aggressively by Unbound, even when the DNSSEC validation option is turned off.

Among the inconsistencies, our clustering method generates 22 clusters, and we investigate each cluster to identify

[5]The cache oracle is able to discover both cache poisoning and cache-related resource consumption bugs, like *RC*2 in Section 6.

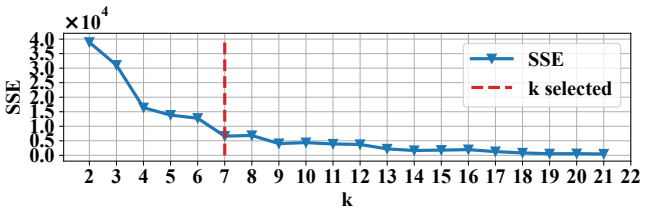
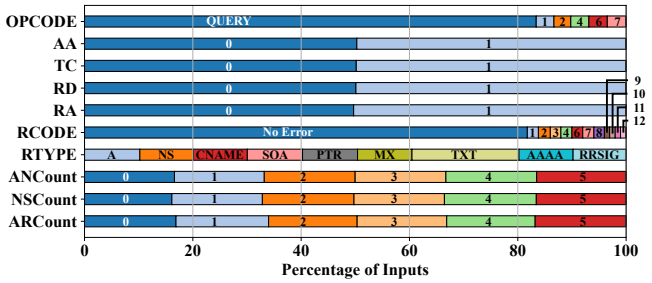
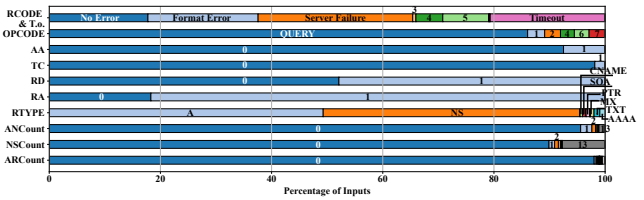


Figure 3: Sum of Squared Error (SSE) to different k for bisecting K-means of *forward-only* cache oracle.



(a) Client-queries and NS-responses.



(b) Resolver-responses. "*RCODE & T.o.*" refers to "RCODE and Timeouts".

Figure 4: Input coverage analysis on: a) client-queries and ns-responses; b) resolver-responses. The client-query and ns-response have the similar distribution for fields from `OPCODE` to `TYPE`. `AN`/`NS`/`ARCOUNT` applies to ns-responses. The values marked on bars are standard DNS values from [45].

the situation(s) about the inconsistencies. For example, the *180K* data points for the *forward-only* mode are grouped into 7 clusters by bisecting K-means. We also identify that only BIND and Unbound implement a fallback mechanism for when the forwarder cannot receive a response with cluster *C13* under column "R2". Overall, the results show that our testing oracles can significantly reduce the manual efforts in locating the vulnerabilities.

Finally, we justify how a key parameter, k for K-means, is selected. For the *forward-only* mode, $k = 7$ because the SSE (sum of squared error) drops significantly at 7 and the slope is gradually flattened after that, as shown in Figure 3. According to elbow method [47], k should be set to such value to achieve the best performance in clustering.

Analysis of tests generation. We perform statistical analysis on test cases generated by ResolverFuzz to understand

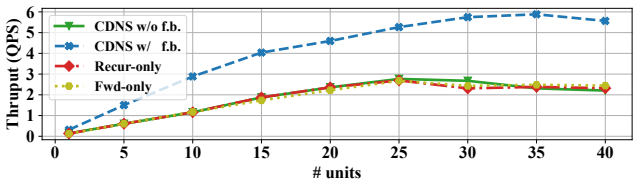


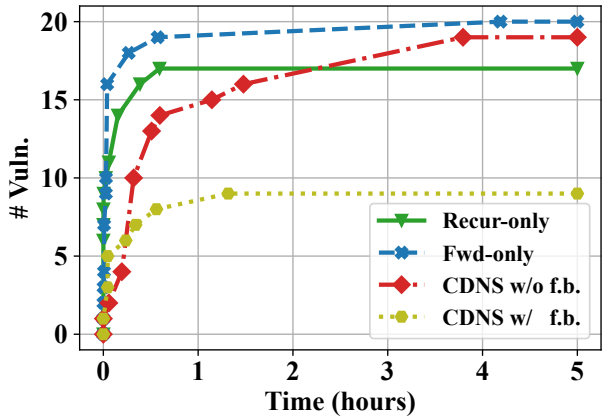
Figure 5: Throughput ("*Thruput*") of 4 modes with regard to the number of units. *CDNS w/o f.b.*, *CDNS w/ f.b.*, *Recur-only* and *Fwd-only* refers to *CDNS without fallback*, *CDNS with fallback*, *Recursive-only*, and *Forward-only*.

their main characteristics. We randomly sample 5K test cases for each mode, 20K in total, and parse the DNS messages of queries and responses. Figure 4(a) shows the distribution of key fields, including `TYPE`, `RCODE`, `OPCODE`, etc. Results show that our fuzzer achieves good coverage of different field values, and the rule probabilities of PCFG ensure certain code logic is tested more intensively. For example, about 80% tests have `OPCODE` set to `QUERY`, the other DNS modes that are not related to resolution (like `NOTIFY`) have much fewer test cases.

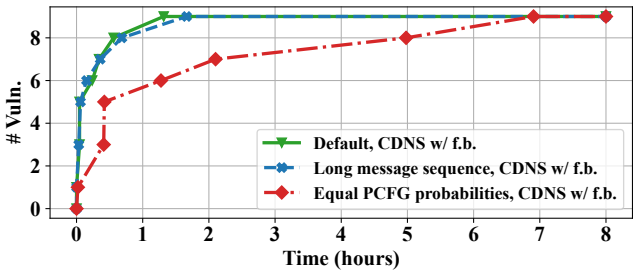
We also inspect the resolver-responses triggered by the client-queries and Figure 4(b) shows the distribution. Only 17.8% of the tests have `RCODE` that equals `NOERROR`, suggesting our test cases are prone to trigger errors, potentially bugs. We have also observed that 20.7% of the test cases reach timeout without getting a reply.

Runtime performance. After measuring the effectiveness of ResolverFuzz, we measure the efficiency of ResolverFuzz, focusing on its testing throughput. Here we employ *Queries per second (QPS)* as the metric, which shows how many cases from the generator are tested per second. We measure the QPS for different resolver modes and different numbers of units, and Figure 5 shows the results. CDNS with fallback mode is only supported by BIND and Unbound. For the 3 other modes, all resolver software is tested. For the default setting (25 units), tests against CDNS with fallback have 2x throughput (5.9 QPS) than the other modes (2.7 - 2.8 QPS). The main reason is that MaraDNS and PowerDNS are slower in responding to client-queries, and ResolverFuzz resetting all resolvers synchronously for each test round. Besides, pre- and post-processing, such as nameserver initialization, slow down the tests and reduce QPS. Still, our result is comparable to other network fuzzers: e.g., T-Reqs achieves 7.9 QPS in HTTP fuzzing ("Request body" of Table 2 in [21]).

We also measure the impact of the unit number on the throughput, and Figure 5 shows the trend. The throughput peaks for "CDNS with fallback" mode when 35 units are used. For the other modes, the throughput peaks at 25 units. Compared with single-unit execution, a 19-time throughput increase is observed. The peak throughput is limited by our workstation setup (only 32 threads can be run concurrently).



(a) Recursive-only, forward-only and CDNS with/without fallback modes.



(b) CDNS with fallback under the default setting, long message sequence setting, and equal PCFG probabilities setting.

Figure 6: The number of vulnerabilities discovered along the time spent by ResolverFuzz.

Trend of vulnerability discovery. We discovered 23 vulnerabilities in total and we elaborate them in Section 6. Here we measure the number of vulnerabilities discovered from all the resolvers with regard to time spent. In Figure 6(a), we demonstrated our result in 4 resolver modes. As a vulnerability often relates to a number of test cases, we use the first test case to log the vulnerability discovery time. After 5 hours, all vulnerabilities were discovered, suggesting ResolverFuzz can discover vulnerabilities efficiently. The number of vulnerabilities discovered differ by modes, since 1) CDNS with fallback mode is only supported by BIND and Unbound, and 2) some vulnerabilities cannot be triggered in all modes.

5.3 Comparison and Ablation Study

Comparison with other fuzzers. Section 2.3 compares ResolverFuzz with other tools that discover DNS bugs briefly. Here, we provide detailed quantitative comparison with baseline DNS fuzzers including dns-fuzz-server, DNS Fuzzer and SnapFuzz. We run these systems and monitor crash, which is the default bug type considered by them.

We first compare with dns-fuzz-server [63], which combines grammar-based and byte-level mutation, but tests a resolver in a *stateless* way. It consists of a fuzz-client, which

sends client-queries, and a fuzz-server, which serves as an nameserver and sends ns-responses, however there is no coordination between these two components. We run dns-fuzzer-server to test BIND in the 4 resolver modes and set the interval between test cases to 0.2 seconds. Each mode was tested for 10 hours (180K test case generated for each mode, similar as RESOLVERFUZZ), but no crash was triggered. We found most of the ns-responses are simply refused by BIND because they do not have matched resolver-queries and client-queries.

Different from dns-fuzzer-server, DNS Fuzzer [17] only implements fuzz-client and only performs byte-level mutation on seed DNS messages. We also test it against BIND and set the interval between test cases to 0.1 seconds, which yield 180K test cases for each resolver mode in 10 hours. Again, no crash was triggered. The traffic analysis shows that most of the mutated client-queries fail to pass the resolution of BIND. The rest of client-queries mostly ended up with timeout because the resolver does not receive ns-responses from a nameserver.

We tried to run SnapFuzz on BIND but it turns out SnapFuzz does not support BIND[6]. Meanwhile, we found SnapFuzz is built on top of AFLNet [53] and speeds up AFLNet by 7x when fuzzing Dnsmasq [4]. Hence, we take an alternative approach to run the AFLNet baseline inside the SnapFuzz repo for 7 days against BIND (also 4 modes), but no crash was detected. Though AFLNet conducts greybox fuzzing to improve the quality of test cases, it only simulates client-queries (implemented by the SnapFuzz repo), which is prone to generate DNS messages easily rejected by the resolver.

To summarize, no crash has been identified by the baselines, suggesting finding crash bugs from the intensively tested resolvers like BIND is non-trivial. RESOLVERFUZZ is able to trigger 1 crash (CC1 described in Section 6.3).

Length of message sequence. RESOLVERFUZZ generates short message sequences based on the insights of our CVE study described in Section 2.2. Here we evaluate whether generating long message sequence can yield new vulnerabilities. We tested BIND under the CDNS with fallback mode and set the new sequence length to 5, such that 5 query-response pairs are sent to BIND in each round. Each message pair is independently generated. In the end, we did not discover any new vulnerability. Interestingly, RESOLVERFUZZ triggers vulnerability faster than the short sequence, as shown in Figure 6(b). This is because all the message pairs in one round share the same cache, so the time spent on querying root and TLD servers can be saved. However, such a performance boost is rather small. Admittedly, the way we generate the message pairs can be optimized by considering their dependencies. However, such change is non-trivial.

PCFG probabilities. We assign different probabilities to different terminals when generating message templates under PCFG, as described in Section 4.2. Here we assess the impact of this design choice, by evaluating a simpler setting that assigns the equal probability to each terminal shown in List 1 and List 2. Again, no new vulnerabilities are discovered and we also found the pace of vulnerability discovery is significantly slowed down, as shown in Figure 6(b). The main reason is that messages are more likely to be rejected before reaching the deep code logic. For instance, the probability of `NOERROR` in `RCODE` is assigned to 0.8 under our default setting. However, if the probability is assigned equally, the probability of `NOERROR` is reduced to 0.091, and most of the responses will be trivially rejected.

[6]SnapFuzz hooks system calls to directly learn when the tested server is able to receive a new request, so there is no need to set a fixed interval between test cases. However, SnapFuzz does not support system calls invoked through `epoll` [57], which is extensively used by BIND.

6 Discovered Vulnerabilities

We identify *23* vulnerabilities with the help of RESOLVERFUZZ: cache poisoning (13), resource consumption (9), and crash & corruption (1). After in-depth discussions with related vendors, 19 of these bugs have been confirmed or fixed, and 15 CVEs have been assigned. As shown in Table 2, we categorize these bugs into 12 classes including *CP*1-*CP*4, *RC*1-*RC*7, and *CC*1, and list the number of vulnerable test cases in the note. Below, we first describe the concrete threat model and then elaborate on each bug group.

6.1 Cache Poisoning Bugs

Concrete threat model. Different from the previous work that either considered recursive mode only [38] or forward mode only [73], we consider the 4 resolver modes as described in Section 2.1 and configure the cache accordingly. Specifically, the target resolver has two DNS zones. Client-queries matching the forwarding zone Z_F are directly forwarded to an upstream server (resolver-queries Q_F), and client-queries matching the recursive zone (Z_R) will trigger recursive resolution (resolver-queries Q_R). Both forwarding and recursive mode *share a global cache*.

When launching an attack, the attacker sends a client-query (Q) to the target resolver, then provides malicious ns-responses (R_{attack}) to the resolver prior to the arrival of legal responses (R_F or R_R). After accepting the malicious responses, the global cache and client would save the tampered answers. When the attacker is not on the resolution path between the client and the intended nameserver, the attacker can conduct IP spoofing and port guessing for the off-path cache poisoning [38, 39, 65]. The concrete threat model is shown in Figure 7.

CP1: Out-of-bailiwick cache poisoning. The bailiwick rule requires that authoritative servers should not return data outside of their controlled zones [12]. For example, responses from `.com` should not include data of other zones like `.net`. Correspondingly, when receiving out-of-bailiwick data, resolvers should discard it before caching. However, when

Table 2: Identified bugs and test cases of six mainstream DNS software.

Software*	Cache poisoning					Resource consumption								Crash& Corruption	Total
	CP1	CP2	CP3	CP4[1]	Tot.[2]	RC1	RC2	RC3	RC4	RC5	RC6	RC7	Tot.	CC1	
BIND	✓†	✗	✓	✓	3	✗	✗	✗	✗	✗	✗	✗	0	✓	4
Unbound	✗	✗	✓	✓†	2	✗	✓	✓	✗	✓	✓	✗	4	-	6
Knot	✓†	✗	✓†	✓†	3	✗	✗	✗	✗	✗	✗	✓†	1	-	4
PowerDNS	✗	✓†	✗	✓†	2	✓†	✗	✓†	✗	✗	✗	✗	2	-	4
MaraDNS	✗	✗	-	✓†	1	✗	✗	✗	✓†	✗	✗	✗	1	-	2
Technitium	✓†	✗	-	✓†	2	✗	✗	✗	✓†	✗	✗	✗	1	-	3
Total	3	1	3	6	13	1	2	1	2	1	1	1	9	1	23

*: Recursive or forwarding modes. [1]: They are triggered by different responses and their cache are inconsistent. [2]: Total. ✓or ✓: Vulnerable.
✓: Discussed but no immediate action. ✓: Confirmed and/or fixed by vendors. ✗: Not vulnerable. †: CVEs assigned. '-': Not applicable.
Amount of test cases: *CP*1 (19), *CP*2 (1,422), *CP*3 (111,328), *CP*4 (7,856), *RC*1 (539,745), *RC*2 (112,126), *RC*3 (88,935), *RC*4 (132), *RC*5 (272) *RC*6 (6,264), *RC*7 (4,448), and *CC*1 (5).

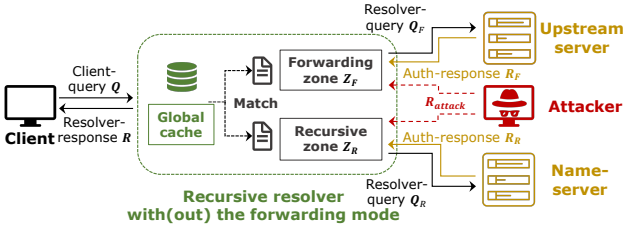


Figure 7: Threat model of cache poisoning bugs.

testing CDNSes with fallback, we found BIND accepts out-of-bailiwick data. Knot and Technitium are also identified to have this issue in further tests.

For a resolver-query Q_F sent to our authoritative server, such as `atkr-fwd.com`, these three resolvers will *cache every record* in the auth-responses generated by our mutator, even including the out-of-bailiwick records shown in Figure 8(a). In this example, since the forged `NS` records of `.com` with a `AA` flag have a higher ranking [12], resolvers would opt to overwrite existing cached records and utilize them for future resolution. Then, following the records of "Authority" and "Additional" Sections, the resolvers would request the attacker's nameserver `ns.atkr-fwd.com` for all queries under the `.com` zone, which allows the attacker to *hijack the entire TLD zone*. After discussion, all affected vendors confirmed this vulnerability and patched their software. We received 3 CVEs and the detailed study is presented in [36].

CP2: In-bailiwick cache poisoning. When the client-query matches a domain name in the forwarding zone, e.g., `vctm-fwd.com`, all software except PowerDNS simply forwards the query to the upstream server and waits for responses. However, PowerDNS first searches its cache for nameservers. If there is a cache hit, it follows the nameserver records and finishes the resolution. Otherwise, it sends resolver-queries to the upstream server. After receiving a response with additional nameserver information like Figure 8(b), PowerDNS will cache every record. Hence, this difference makes cache poisoning more powerful for PowerDNS: when the attacker conducts off-path cache poisoning, she just needs to tamper *one* `NS` record (e.g., `s.vctm-fwd.com`) during forwarding, and all the follow-up queries under the zone of `s.vctm-fwd.com` will be tampered (e.g., redirected to the attacker server `a.k.t.r`). For the other software, the attacker has to tamper *every* query.

CP3: Fragmentation-based cache poisoning. According to [10, 73], attackers could leverage IP fragmentation to ini-

Header: TXID; QR AA;
Question section: atkr-fwd.com. A
Answer section: atkr-fwd.com. A x.x.x.x
Authority section: com. NS ns.atkr-fwd.com.
Additional section: ns.atkr-fwd.com. A **a.t.k.r**

(a) Auth-response for *CP*1.

Header: TXID; QR AA;
Question section: vctm-fwd.com. A
Answer section: vctm-fwd.com. A x.x.x.x
Authority section: s.vctm-fwd.com. NS ns.vctm-fwd.com.
Additional section: ns.vctm-fwd.com. A **a.t.k.r**

(b) Auth-response for *CP*2.

Header: TXID; QR AA;
Answer section: victim.com. A x.x.x.x
victim.com. RRSIG xxx...x

(c) 1st fragment for *CP*3.

Authority section: victim.com. NS ns.victim.com.
Additional section: ns.victim.com. A **a.t.k.r**

(d) spoofed 2rd fragment for *CP*3.

Header: TXID; QR AA;
Question section: s.atkr-rev.com. A
Answer section: s.atkr-rev.com. A **a.t.k.r**
Authority section: s.atkr-rev.com. NS ns.atkr-rev.com.
Additional section: ns.atkr-rev.com. A **a.t.k.r**

(e) Auth-response for *CP*4.

Header: TXID; QR AA;
Question section: s.atkr-rev.com. A
Answer section: (Empty)
Authority section: s.atkr-rev.com. NS ns.atkr-rev.com.
Additional section: ns.atkr-rev.com. A **a.t.k.r**

(f) Ref-response for *CP*4.

Figure 8: DNS responses utilized for cache poisoning attacks. Red parts carry the attack payloads.

tiate a DNS cache poisoning attack. This attack exploits the fact that the second fragment of a fragmented DNS response packet contains neither UDP nor DNS headers, thus it is much easier to spoof this fragment as there is no need to guess the UDP source port or DNS `TXID`. During an attack, attackers first send the spoofed second fragment (e.g., Figure 8(d)) to the target resolver, then issue a query for the victim domain whose nameserver will return fragmented DNS packets. After receiving the first fragment shown in Figure 8(c), the target resolver will resemble it with the previously cached second fragment, resulting in a rogue DNS response to be accepted.

Though RESOLVERFUZZ does not directly generate fragmented packets, this bug was discovered because RESOLVERFUZZ can generate large-size DNS messages (e.g., exceeding the general 1,500 bytes MTU limit for Ethernet) [18]. Through traffic analysis, RESOLVERFUZZ found that BIND, Unbound, and Knot allow fragmented ns-responses to be larger than 1,232 bytes and even 4,096 bytes, whereas the other software only accepts ns-responses less than 1,232 bytes. Attackers could exploit nameservers that return large DNS responses or utilize the techniques in [73] to conduct fragmentation-based cache poisoning attacks.

CP4: Iterative subdomain caching. During fuzzing, we discover that software including BIND, Unbound, Knot, and PowerDNS will accept unsolicited records from auth-responses (e.g., records in the Authority and Additional section Figure 8(e)), while MaraDNS and Technitium will store the records in the ref-responses like Figure 8(f). After caching these records, resolvers will use them to serve future queries. For example, upon receiving a query for `s.atkr-rec.com`, resolvers will send queries straight to the nameserver `ns.atkr-rec.com` rather than iteratively querying the root and TLD servers.

Inspired by this behavior, we introduce a new attack in which the attacker could iteratively inject `NS` records of subdomains into the resolver's cache. Especially, when `NS` records of `s.atkr-rec.com` are about to expire, attackers return nameserver data of `s.s.atkr-rec.com`, enabling the target resolver to still be able to resolve domains under it, and for `s.s.s.atkr-rec.com` so on. In this manner, even if `atkr-rec.com` is revoked from the `.com` zone, the target resolver will continue to resolve a group of subdomains of `atkr-rec.com` for a long time. This attack is an extension of the previous "ghost domain attack" [25] that defeats domain sinkholing [2]. All vendors of tested software have confirmed this vulnerability and some have fixed it. We obtained 5 CVE numbers. The detailed study was presented in [34].

6.2 Resource Consumption Bugs

Concrete threat model. Through a small number of client-queries and/or auth-responses, the attacker occupies a large portion of cache storage, triggers excessive resolver-queries, or consumes large computation overhead on the victim resolver. Figure 9 shows the threat model.

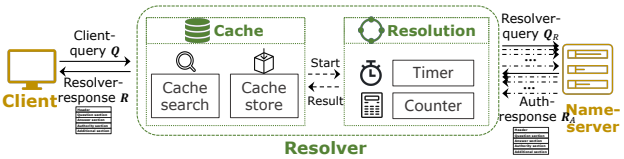


Figure 9: Threat model of resource consumption bugs.

RC1: Excessive cache search operations. When running in forward-only mode, only PowerDNS looks up its local cache for trust anchors and `NS` records before sending it to a server, while the others just forward the client query to upstream servers. For example, for a query of `s.atkr-fwd.com` under the forwarding zone `atkr-fwd.com`, PowerDNS searches its cache following the order of `s.atkr-fwd.com`, `atkr-fwd.com`, `.com`, and the ROOT server until finding an existing `NS` record, by removing one label each round. This process is repeated three times. Therefore, attackers could construct a domain name containing 128 labels (the maximum number) to trick PowerDNS to perform 384 cache search operations, which is much more than other software (i.e., 1).

RC2: Unlimited cache store operations. When receiving auth-responses, Unbound caches all in-bailiwick records from the "Authority" and "Additional" sections into the cache regardless of their validity. According to DNS RFC [6, 45], only `NS`, `SOA`, and DNSSEC-related records are permitted in the "Authority" section, while glue records are allowed in the Additional section. Unbound does not implement this rule and stores all types of records, allowing attackers to squander cache storage. After discussion, Unbound's developers have included a sanitizer to filter out invalid records.

RC3: Ignoring the RD flag. DNS RFCs [45] require resolvers to issue queries to upstream servers only if they receive client queries with the Recursive-Desired flag set (`RD=1`), and refrain from follow-up queries when `RD=0`. However, Unbound and PowerDNS still forward client queries to upstream servers when `RD=0`, under the forwarding mode. The attacker can exploit this bug by directing the follow-up queries to her domain, and point the nameserver in her DNS zone back to the resolver, which results in a query loop and potential amplification attack. The detailed study is presented in [71].

RC4: Following a self-CNAME reference. The `CNAME` record is used to map an alias name to a canonical name (e.g., `www.cnn.com` to `cnn-tls.map.fastly.net`) [45]. Upon detecting a CNAME loop (i.e., responses with a CNAME record having the same alias name and canonical name), the current resolution process should terminate. However, Unbound, MaraDNS, and technitium will chase the CNAME loop for 12, 113, and 289 times. Moreover, technitium will provide the client with a response including 289 CNAME records. Attackers could craft a self-CNAME record to consume resolvers' resources by triggering more outgoing queries and

sending large responses for amplification attacks. MaraDNS and Technitium have confirmed this vulnerability and patched their latest versions. 2 CVE numbers have been assigned.

RC5: Large responses to clients. Large DNS responses have been used extensively in DNS reflection amplification attacks [46]. When analyzing the packet size, our traffic oracle discovered that Unbound enables a maximum packet size of 4,096 bytes, whereas the other software restricts it to less than 1,232 bytes. This suggests Unbound could provide a three-fold amplification ratio compared to other software. After disclosing our findings, Unbound has changed its maximum UDP packet size to 1,232 bytes by default.

RC6: Overlong waiting time over UDP. After receiving an ns-response modified by our byte-level mutator, Unbound continues to issue 9 resolver-queries and waits up to 17 seconds for a legal ns-response. In contrast, other software just discards the invalid ns-responses and does not follow up with resolver-queries. Hence, an attacker can send a large number of such malformatted ns-responses to Unbound, extend the resolver's waiting time and consume its resources.

RC7: Excessive queries for resolution over TCP. Similar to *RC*6, upon receiving an invalid ns-response via TCP, Knot Resolver continues to issue 100 resolver-queries for a legitimate response, thus bypassing the restriction (5 resolver-queries at maximum) used to defend against the powerful NXNSDomain attacks. After discussion, Knot acknowledged and fixed this vulnerability. The detailed study is shown in [37].

6.3 Crash Bugs

Concrete threat model. The attacker sends client-queries or auth-responses to cause memory or non-memory crash.

CC1: Assertion failure when receiving queries. We found an assertion failure occurs when BIND receives the byte-mutated client-queries using the `udp_recv` function, which crashes the resolver service. After reading the source code, we identified this bug in the `udp_recv` dispatching process, which returns a success code but cancels the query in the meantime, violating the assertion that ensures the current `udp_recv` process receives a valid DNS packet. We found BIND fixed this issue in version 9.18.3 [8], but their test case is different from ours.

7 Discussion

Limitations and future work. 1) Our test cases are unable to cover all sorts of DNS messages (e.g., no `DNAME`), as it incurs extensive manual efforts in writing PCFG, and only a subset of DNS message types are actively used. 2) We did not test all DNS-related functionalities, like DNSSEC, due to their different logic from the normal resolver actions, i.e., caching records. 3) We test the stateful resolver with a pair of query and response. Admittedly, some complex bugs that are introduced by long sequences cannot be found. 4) We set a fixed timeout value to 5 seconds, but this is not optimal when the resolver is stuck before the timeout. SnapFuzz enables adaptive timeout on networking software by rewriting its code [4] and our problem can be solved by this approach. 5) We conduct a blackbox fuzzing without relying on the code coverage as feedback. The main reason is that we have not found an ideal metric for the different types of semantic bugs. We plan to continue to explore the combination of coverage-based and grammar-based fuzzing, like [68]. 6) We follow other works (e.g., [74]) to analyze CVEs and assess which type of bugs is more prevalent (e.g., short sequence triggers more bugs). Admittedly, such analysis could suffer from survivorship bias (e.g., bugs triggered by short sequence are easier to find, hence more CVEs). 7) We use differential testing to discover cache poisoning bugs, assuming at least one implementation is correct. However, when the RFC is erroneous and all implementations follow the RFC, such bugs are unlikely to be discovered.

Ethical considerations. 1) Fuzzing the resolver within the standard DNS infrastructure could affect the other remote nameservers, as described in Section 4.1. Hence, we localize the root and TLD servers in our lab network, which improves the efficiency of ResolverFuzz. 2) For the measurement study described in Appendix C, we scanned the whole IPv4 network space. We follow the common practice in Internet-wide scanning, by setting the maximum probing rate to 10 kps and evenly distributing the traffic across the target. Like XMap which also scans IPv4 network space [35], we created a website to receive the opt-out requests and a `PTR` record using our scanner's source IP to show our research intention. We received no opt-out requests during the period of experiments. We admit that such an approach cannot entirely mitigate the ethical issues, as getting informed consent for large-scale Internet measurement is very difficult [50]. We tried our best to follow the best practices.

8 Related Work

Network protocol fuzzing. Various fuzzing techniques have been applied to test network protocols in addition to DNS. AFLNet uses the response code from the server as the feedback to generate sequences of messages [53]. SGFuzz uncovers the state space of a protocol (e.g., HTTP2) from the "enum" variables defined in source code [7]. For TCP, TCP-Fuzz improves the coverage of TCP states with a new branch transition metric [74]. For TLS, grammar-based fuzzing has been applied to generate test cases from specifications [64,67]. Fiterau et al. extended [64] to test DTLS [13]. For HTTP, T-Reqs detects HTTP smuggling vulnerabilities by finding the inconsistencies in how servers split an HTTP request [21]. Frameshifter identifies HTTP/2-to-HTTP/1 protocol conversion anomalies by mutating the HTTP/2 frame sequence [20].

For QUIC, DPIFuzz identifies new elusion methodologies for an attacker to evade QUIC-based Deep Packet Inspection (DPI) [54].

We found none of the prior works can be directly applied to DNS resolvers, due to the different protocol semantics and vulnerability types (e.g., cache poisoning). RESOLVERFUZZ addresses these challenges with a new fuzzing framework and a set of techniques.

Differential testing. RESOLVERFUZZ leverages the inconsistencies between DNS resolvers to identify semantic bugs, in particular cache bugs. Differential testing also exploits this insight to identify semantic bugs, which has been applied in various scenarios A few recent works leverage this technique for network services [61, 69, 70] and we discuss them in details. HDiff generates HTTP messages to detect HTTP Request Smuggling attack, Host of Troubles attack and Cache-Poisoned Denial-of-Service Attack [61]. It extracts syntax rules from RFC and use them to detect implementation discrepancies, however, there lacks precise specifications of resolver behaviors. SYMTCP automatically discovered insertion and evasion TCP packets against DPI middleboxes [69]. It uses symbolic execution to generate TCP state machines of endhosts and observe their discrepancies given TCP packets. Themis tackles a similar problem but detects the discrepancies by comparing the TCP state machines statically and finding the counterexample through a SAT solver [70]. However, DNS resolvers do not have well-defined state machines.

In addition to the efforts of adjusting differential testing to specific scenarios, some works have investigated general strategies. Nezha proposed λ-diversity notation for fuzzing to increase the chances of finding inconsistencies [52]. HyDiff combines symbolic execution with greybox fuzzing to find semantic bugs [48]. In our setting, since we conduct blackbox fuzzing, the metrics and methods proposed by those works do not directly apply.

DNS resolver vulnerabilities. Section 2.2 reviews DNS vulnerabilities from published CVEs. Here, we survey the related academic works. The major interests were centered around cache poisoning bugs, and many found that forwarders are more vulnerable [73]. Jeitner et al. identified semantic inconsistencies in DNS input validation and proposed new string injection attacks for cache poisoning [22]. Recently, Jeitner et al. found special characters can be exploited for DNS cache poisoning attacks against routers [23]. So far, finding resolver bugs requires heavy manual analysis, and RESOLVERFUZZ sheds light on how to automate this process with fuzzing.

When the attacker is not on the resolution path, a malicious response needs to be forged and raced against the legitimate response. In this case, the defense mechanisms based on randomization, like port randomization and TXIDs, have to be bypassed. Attacks like IP fragmentation [18, 73] and ICMP-based side channels [38, 39] have been proposed to achieve such goal. Dai et al. showed that following off-path cache poisoning, Internet resources like IP addresses, domains, certificates, and virtual platforms can be controlled by attackers [11]. The aforementioned vulnerabilities exploit side-channel information of DNS resolution. Though they cannot be directly discovered by RESOLVERFUZZ, combined with the vulnerabilities discovered by RESOLVERFUZZ, more powerful off-path cache poisoning attacks can be enabled (see CP2 of Section 6.1.

9 Conclusion

In this work, we develop a new blackbox fuzzing system RESOLVERFUZZ that is tailored to find DNS resolver vulnerabilities. Based on our study of the published DNS CVEs, RESOLVERFUZZ is designed with a set of novel techniques, including constrained stateful fuzzing, differential testing, and grammar-based fuzzing. Our evaluation results show that RESOLVERFUZZ is effective in finding resolver bugs, with 23 vulnerabilities discovered and 15 CVEs assigned.

Lessons learnt. Despite that DNS resolvers were extensively tested (e.g., BIND has joined Google OSS-Fuzz project to be automatically fuzzed [15]), we can still discover many vulnerabilities in their latest versions. We believe the main reason is that bugs unique to DNS resolvers are still challenging to be discovered with the existing tools, and we hope this study can shed light on this understudied area. Besides, lacking rigorous specifications also contributes to the existence of resolver bugs [46, 65], as reflected by the high number of inconsistencies observed during testing. Like prior work, we encourage the Internet community to work together and develop formal guidance about secured resolver implementations.

Acknowledgement

We thank all the anonymous reviewers and our shepherd for their valuable comments. We thank Fish Wang for his suggestions in differential testing. Authors from Tsinghua University were supported by the National Natural Science Foundation of China (U1836213, U19B2034, 62102218, and 62132011). Authors from UCI were supported by NSF CNS-2047476.

References

[1] M. Allman. Comments on dns robustness. In *Proceedings of the Internet Measurement Conference 2018*, pages 84–90, 2018.

[2] E. Alowaisheq, P. Wang, S. Alrwais, X. Liao, X. Wang, T. Alowaisheq, X. Mi, S. Tang, and B. Liu. Cracking the Wall of Confinement: Understanding and Analyzing Malicious Domain Take-downs. In *NDSS '19*.

[3] A. E. Alvarez. DNS Forwarding and Conditional Forwarding. https://medium.com/tech-jobs-academy/dns-forwarding-and-conditional-forwarding-f3118bc93984, 2016.

[4] A. Andronidis and C. Cadar. Snapfuzz: High-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. RFC 4035: Protocol Modifications for the DNS Security Extensions. *RFC Proposed Standard*.

[6] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Rfc 4033: Dns security introduction and requirements, 2005.

[7] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, Boston, MA, Aug. 2022. USENIX Association.

[8] BIND. "xferquota" system test fails intermittently. https://gitlab.isc.org/isc-projects/bind9/-/issues/3300.

[9] BIND. BIND Document: type forward. https://bind9.readthedocs.io/en/latest/reference.html#namedconf-statement-type%20forward, 2023.

[10] M. Brandt, T. Dai, A. Klein, H. Shulman, and M. Waidner. Domain Validation++ For MitM-Resilient PKI. In *CCS '18*.

[11] T. Dai, P. Jeitner, H. Shulman, and M. Waidner. The Hijackers Guide To The Galaxy: Off-Path Taking Over Internet Resources. In *USENIX Security '21*.

[12] R. Elz and R. Bush. RFC 2181: Clarifications to the DNS Specification. *RFC Proposed Standard*.

[13] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. De Ruiter, K. Sagonas, and J. Somorovsky. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540, 2020.

[14] Google. Fuzzing with afl-fuzz. https://afl-1.readthedocs.io/en/latest/fuzzing.html.

[15] Google. oss-fuzz/projects/bind9 at master · google/oss-fuzz. https://github.com/google/oss-fuzz/tree/master/projects/bind9, 2022.

[16] L. Grangeia. Cache snooping or snooping the cache for fun and profit, 2004.

[17] guyinatuxedo. https://github.com/guyinatuxedo/dns-fuzzer, 2019.

[18] A. Herzberg and H. Shulman. Fragmentation Considered Poisonous, or: One-domain-to-rule-them-all.org. In *CNS '13*.

[19] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[20] B. Jabiyev, S. Sprecher, A. Gavazzi, T. Innocenti, K. Onarlioglu, and E. Kirda. {FRAMESHIFTER}: Security implications of {HTTP/2-to-HTTP/1} conversion anomalies. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1061–1075, 2022.

[21] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda. T-reqs: Http request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1805–1820, 2021.

[22] P. Jeitner and H. Shulman. Injection Attacks Reloaded: Tunnelling Malicious Payloads over DNS. In *USENIX Security '21*.

[23] P. Jeitner, H. Shulman, L. Teichmann, and M. Waidner. {XDRI} attacks-and-how to enhance resilience of residential routers. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4473–4490, 2022.

[24] F. Jelinek, J. D. Lafferty, and R. L. Mercer. *Basic methods of probabilistic context free grammars*. Springer, 1992.

[25] J. Jiang, J. Liang, K. Li, J. Li, H.-X. Duan, and J. Wu. Ghost Domain Names: Revoked Yet Still Resolvable. In *NDSS '12*.

[26] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. Dns performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 153–167, 2001.

[27] S. K. R. Kakarla. *Formal Methods for a Robust Domain Name System*. University of California, Los Angeles, 2022.

[28] S. K. R. Kakarla, R. Beckett, B. Arzani, T. Millstein, and G. Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, pages 310–328, 2020.

[29] S. K. R. Kakarla, R. Beckett, T. Millstein, and G. Varghese. {SCALE}: Automatically finding {RFC} compliance bugs in {DNS} nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 307–323, 2022.

[30] D. Kaminsky. Black ops 2008: It's the end of the cache as we know it. *Black Hat USA*, 2, 2008.

[31] S. M. Kerner. BIND DNS Holds Lead. https://www.serverwatch.com/server-news/bind-dns-holds-lead/, 2020.

[32] J. Knudsen. CyRC Case Study: Securing BIND 9. https://www.synopsys.com/blogs/software-security/cyrc-case-study-securing-bind-9/, 2022.

[33] D. C. Lawrence, W. A. Kumari, and P. Sood. RFC 8767: Serving Stale Data to Improve DNS Resiliency. *RFC Proposed Standard*, 2020.

[34] X. Li, B. Liu, X. Bai, M. Zhang, Q. Zhang, Z. Li, H. Duan, and Q. Li. Ghost Domain Reloaded: Vulnerable Links in Domain Name Delegation and Revocation. In *NDSS '23*.

[35] X. Li, B. Liu, X. Zheng, H. Duan, Q. Li, and Y. Huang. Fast IPv6 Network Periphery Discovery and Security Implications. In *DSN '21*.

[36] X. Li, C. Lu, B. Liu, Q. Zhang, Z. Li, H. Duan, and Q. Li. The Maginot Line: Attacking the Boundary of DNS Caching Protection. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security '23)*, 2023.

[37] X. Li, W. Xu, B. Liu, M. Zhang, Z. Li, J. Zhang, D. Chang, X. Zheng, C. Wang, J. Chen, H. Duan, and Q. Li. TuDoor Attack: Systematically Exploring and Exploiting Logic Vulnerabilities in DNS Response Pre-processing with Malformed Packets. In *Proceedings of 2024 IEEE Symposium on Security and Privacy*, Oakland S&P '24, 2024.

[38] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan. DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels. In *CCS '20*.

[39] K. Man, X. Zhou, and Z. Qian. DNS Cache Poisoning Attack: Resurrections with Side Channels. In *CCS '21*.

[40] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[41] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[42] MITRE. CVE. https://cve.mitre.org/, 2023.

[43] MITRE. CVE Details. https://www.cvedetails.com/, 2023.

[44] MITRE. CVE List. https://www.cve.org/, 2023.

[45] P. V. Mockapetris. RFC 1034: Domain Names - Concepts and Facilities. *RFC Standard*.

[46] S.-J. Moon, Y. Yin, R. A. Sharma, Y. Yuan, J. M. Spring, and V. Sekar. Accurately Measuring Global Risk of Amplification Attacks using AmpMap. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*, 2021.

[47] R. Nainggolan, R. Perangin-angin, E. Simarmata, and A. F. Tarigan. Improved the performance of the k-means cluster using the sum of squared error (sse) optimized by using the elbow method. In *Journal of Physics: Conference Series*, volume 1361, page 012015. IOP Publishing, 2019.

[48] Y. Noller, C. S. Păsăreanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske. Hydiff: Hybrid differential software analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1273–1285. IEEE, 2020.

[49] V. Pappas, P. Fältström, D. Massey, and L. Zhang. Distributed DNS troubleshooting. In *Proceedings of the ACM SIGCOMM workshop on Network troubleshooting: research, theory and operations practice meet malfunctioning reality*, pages 265–270, 2004.

[50] C. Partridge and M. Allman. Ethical considerations in network measurement papers. *Communications of the ACM*, 59(10):58–64, 2016.

[51] H. Peng, Z. Yao, A. A. Sani, D. J. Tian, and M. Payer. Gleefuzz: Fuzzing webgl through error message guided mutation. *USENIX Security'23*, 2023.

[52] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.

[53] V.-T. Pham, M. Böhme, and A. Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[54] G. S. Reen and C. Rossow. Dpifuzz: a differential fuzzing framework to detect dpi elusion strategies for quic. In *Annual Computer Security Applications Conference*, pages 332–344, 2020.

[55] ResolverFuzz. https://github.com/ResolverFuzz/ResolverFuzz, 2023.

[56] A. Romao. Tools for dns debugging. Technical report, RFC 1713, FCCN, November, 1994.

[57] SaBRe. The binary rewriting plugin sabre used by snapfuzz. https://github.com/andronat/SaBRe/blob/4a41a5adaec89235e00adc5d339be308f5c8d57c/plugins/sbr-afl/main.c, 2022.

[58] K. Schomp, T. Callahan, M. Rabinovich, and M. Allman. On measuring the client-side DNS infrastructure. In *IMC '13*.

[59] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 28–28, 2012.

[60] sharadarao1999. Bisecting K-Means Algorithm Introduction. https://www.geeksforgeeks.org/bisecting-k-means-algorithm-introduction/.

[61] K. Shen, J. Lu, Y. Yang, J. Chen, M. Zhang, H. Duan, J. Zhang, and X. Zheng. Hdiff: A semi-automatic framework for discovering semantic gap attack in http implementations. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–13. IEEE, 2022.

[62] A. Singram and K. Umashankar. Implementing dual stack recursive DNS at Microsoft: Challenges and Learning. https://indico.dns-oarc.net/event/42/contributions/904/, 2022.

[63] sischkg. https://github.com/sischkg/dns-fuzz-server, 2019.

[64] J. Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[65] S. Son and V. Shmatikov. The Hitchhiker's Guide to DNS Cache Poisoning. In *SecureComm '10*.

[66] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical software engineering*, 19:1665–1705, 2014.

[67] A. Walz and A. Sikora. Exploiting dissent: towards fuzzing-based differential black-box testing of tls implementations. *IEEE Transactions on Dependable and Secure Computing*, 17(2):278–291, 2017.

[68] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[69] Z. Wang, S. Zhu, Y. Cao, Z. Qian, C. Song, S. V. Krishnamurthy, K. S. Chan, and T. D. Braun. Symtcp: Eluding stateful deep packet inspection with automated discrepancy discovery. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[70] Z. Wang, S. Zhu, K. Man, P. Zhu, Y. Hao, Z. Qian, S. V. Krishnamurthy, T. La Porta, and M. J. De Lucia. Themis: Ambiguity-aware network intrusion detection based on symbolic model comparison. In *Proceedings of the 8th ACM Workshop on Moving Target Defense*, pages 31–32, 2021.

[71] W. Xu, X. Li, C. Lu, B. Liu, J. Zhang, J. Chen, T. Wan, and H. Duan. TsuKing: Coordinating DNS Resolvers and Queries into Potent DoS Amplifiers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[72] Q. Zhang, X. Bai, X. Li, H. Duan, Q. Li, and Z. Li. Resolverfuzz: Automated discovery of dns resolver vulnerabilities with query-response fuzzing. arXiv, 2023.

[73] X. Zheng, C. Lu, J. Peng, Q. Yang, D. Zhou, B. Liu, K. Man, S. Hao, H. Duan, and Z. Qian. Poison Over Troubled Forwarders: A Cache Poisoning Attack Targeting DNS Forwarding Devices. In *USENIX Security '20*.

[74] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu. {TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502, 2021.

A CVE Details

Process of CVE analysis. Our CVE study in Section 2.2 characterizes each CVE by its impacted DNS modes (e.g., resolver or nameserver), impacted software, bug type, and the trigger condition (e.g., the related DNS field and the message sequence length). The first three categories can be readily learnt from the CVE reports. For the last one, we found some software vendors include detailed description about the trigger condition. Besides, we can also synthesize a possible trigger condition from the patch described in the git commits.

CVE-2022-3736. The attacker could trick a BIND resolver, which is configured to answer from a stale cache, to terminate unexpectedly by sending just one `RRSIG` query. Under the stale cache configuration [33], BIND would attempt to return expired records in the cache to clients if authoritative servers do not reply. However, BIND does not provide a complete and correct implementation of the stale cache mechanism for the `RRSIG` query. Upon receiving a `RRSIG` query with the stale option set, BIND terminates due to the assertion failure `qtype != RRSIG`.

CVE-2022-3924. By sending multiple queries to BIND, the attacker would drive BIND with the stale option enabled into a race condition and eventually crash. After receiving a large number of queries for recursive resolution, BIND processes each one from the query queue separately and makes all clients wait for responses. When a new client query is received, limited by the total number of clients, BIND will reply to the client who has the longest waiting time with the `SERVFAIL` response. However, under the stale configuration, a race condition could occur between providing a stale answer and response timeout, which might cause BIND to crash.

B PCFG Details

List 1 and List 2 show the detailed PCFG for generating DNS query and response packet.

```
⟨start⟩ ::= ⟨query⟩
⟨query⟩ ::= ⟨Header⟩⟨Question⟩
⟨Header⟩ ::= ⟨TransactionID⟩⟨Flags⟩⟨RRs⟩
⟨TransactionID⟩ ::= (randomly generated 2-byte hex value)
⟨Flags⟩ ::= ⟨QR⟩⟨OPCODE⟩⟨AA⟩⟨TC⟩⟨RD⟩⟨RA⟩⟨Z⟩⟨AD⟩⟨CD⟩⟨RCODE⟩
⟨QR⟩ ::= 0
⟨OPCODE⟩ ::= QUERY[.80] | IQUERY[.04] | STATUS[.04] |
    NOTIFY[.04] | UPDATE[.04] | DSO[.04]
⟨AA⟩ ::= 0 | 1
⟨TC⟩ ::= 0 | 1
⟨RD⟩ ::= 0 | 1
⟨RA⟩ ::= 0 | 1
⟨Z⟩ ::= 0 | 1
⟨AD⟩ ::= 0 | 1
⟨CD⟩ ::= 0 | 1
⟨RCODE⟩ ::= NOERROR[.80] | FORMERR[.01] | SERVFAIL[.01] |
    NXDOMAIN[.01] | NOTIMP[.01] | REFUSED[.01] | YXDOMAIN
    [.01] | YXRRSET[.01] | NXRRSET[.01] | NOTAUTH[.01] |
    NOTZONE[.01] | DSOTYPENI[.01] | BADVERS[.01] | BADKEY
    [.01] | BADTIME[.01] | BADMODE[.01] | BADNAME[.01] |
    BADALG[.01] | BADTRUNC[.01] | BADCOOKIE[.01]
⟨RRs⟩ ::= ⟨QDCOUNT⟩⟨ANCOUNT⟩⟨NSCOUNT⟩⟨ARCOUNT⟩
⟨QDCOUNT⟩ ::= 1
⟨ANCOUNT⟩ ::= 0
⟨NSCOUNT⟩ ::= 0
⟨ARCOUNT⟩ ::= 0
⟨Question⟩ ::= ⟨QNAME⟩⟨QTYPE⟩⟨QCLASS⟩
⟨QNAME⟩ ::= (base domain)[.40] |
          (sub-domain)[.40] |
          (2-9th sub-domain)[.10] |
          (10-max sub-domain)[.10] |
⟨QTYPE⟩ ::= A | NS | CNAME | SOA | PTR | MX | TXT | AAAA |
     RRSIG | SPF | ANY
⟨QCLASS⟩ ::= IN
```

Listing 1: PCFG for DNS query.

```
⟨start⟩ ::= ⟨response⟩
⟨response⟩ ::= ⟨Header⟩⟨Answer⟩⟨Authority⟩⟨Additional⟩
⟨Header⟩ ::= ⟨Flags⟩⟨RRs⟩
⟨Flags⟩ ::= ⟨QR⟩⟨OPCODE⟩⟨AA⟩⟨TC⟩⟨RD⟩⟨RA⟩⟨Z⟩⟨AD⟩⟨CD⟩⟨RCODE⟩
⟨QR⟩ ::= 1
⟨OPCODE⟩ ::= QUERY[.80] | IQUERY[.04] | STATUS[.04] |
    NOTIFY[.04] | UPDATE[.04] | DSO[.04]
⟨AA⟩ ::= 0 | 1
⟨TC⟩ ::= 0 | 1
⟨RD⟩ ::= 0 | 1
⟨RA⟩ ::= 0 | 1
⟨Z⟩ ::= 0 | 1
⟨AD⟩ ::= 0 | 1
⟨CD⟩ ::= 0 | 1
⟨RCODE⟩ ::= NOERROR[.80] | FORMERR[.01] | SERVFAIL[.01] |
    NXDOMAIN[.01] | NOTIMP[.01] | REFUSED[.01] | YXDOMAIN
    [.01] | YXRRSET[.01] | NXRRSET[.01] | NOTAUTH[.01] |
    NOTZONE[.01] | DSOTYPENI[.01] | BADVERS[.01] | BADKEY
    [.01] | BADTIME[.01] | BADMODE[.01] | BADNAME[.01] |
    BADALG[.01] | BADTRUNC[.01] | BADCOOKIE[.01]
⟨RRs⟩ ::= ⟨ANCOUNT⟩⟨NSCOUNT⟩⟨ARCOUNT⟩
⟨ANCOUNT⟩ ::= 0 | 1 | 2 | 3 | 4 | 5
⟨NSCOUNT⟩ ::= 0 | 1 | 2 | 3 | 4 | 5
⟨ARCOUNT⟩ ::= 0 | 1 | 2 | 3 | 4 | 5
⟨Answer⟩ ::= "" | ⟨Record⟩ | ⟨Record⟩*2 | ⟨Record⟩*3 | ⟨
    Record⟩*4 | ⟨Record⟩*5
⟨Authority⟩ ::= "" | ⟨Record⟩ | ⟨Record⟩*2 | ⟨Record⟩*3
    | ⟨Record⟩*4 | ⟨Record⟩*5
⟨Additional⟩ ::= "" | ⟨Record⟩ | ⟨Record⟩*2 | ⟨Record⟩
    *3 | ⟨Record⟩*4 | ⟨Record⟩*5
```

```
⟨Record⟩ ::= ⟨NAME⟩⟨TYPE⟩⟨CLASS⟩⟨TTL⟩⟨RDLENGTH⟩⟨RDATA⟩
⟨NAME⟩ ::= (domain queried)[.2] |
           (sub-domain)[.2] |
           (same-level domain)[.2] |
           (parent domain)[.2] |
           (unrelated domain)[.2]
⟨TYPE⟩ ::= (TYPE queried)[.50] | A[.05] | CNAME[.05] | SOA
     [.05] | PTR[.05] | MX[.05] | TXT[.05] | AAAA[.05] |
     RRSIG[.05] | SPF[.05]
⟨CLASS⟩ ::= IN
⟨TTL⟩ ::= 60
⟨RDLENGTH⟩ ::= (length of ⟨RDATA⟩)[.90] | (random value
     in [length, 2*length])[.05] | (random value in [0,
     length])[.05]
⟨RDATA⟩ ::= (randomly generated data decided by ⟨TYPE⟩)
```

Listing 2: PCFG for DNS response.

C Large-scale Resolver Scanning

The vulnerabilities identified by RESOLVERFUZZ persist in all the studied DNS software (even in their latest versions), and we are interested in learning the impact if these vulnerabilities are exploited in the wild. To this end, we conduct a large-scale network scanning on open resolvers to discover the potentially vulnerable ones. We discuss the ethical issues and how they are addressed in Section 7.

Methodology of scanning. As revealed by [58], the list of active open resolvers is constantly changing. To obtain an up-to-date list of open resolvers, we conduct active network scanning using `XMap` [35]. `XMap` is a fast Internet-wide scanner that allows customization of probing requests. We use `XMap` to send DNS queries from our lab machines to the whole IPv4 address space. Each query is a UDP packet, issued against port 53, and querying about our controlled domain names. We consider the IPs replying with valid responses to be candidates for open resolvers. To notice, scanning from a few vantage points cannot yield a complete resolver list. However, this is the common practice for measurement studies about Internet services and DNS resolvers.

Due to ethical concerns, we cannot directly test if a resolver is vulnerable by sending packets generated by our fuzzer (otherwise, we might be considered as attackers by resolver operators). Hence, we try to learn software names of open resolvers, and check if they match our studied resolvers. We did not include software versions as we found version information is usually not provided by resolvers, and all our discovered vulnerabilities are effective against the latest versions.

Specifically, we use both `version.bind` query and a DNS fingerprinting tool `fpdns`. `version.bind` is a special DNS query name configured with `TXT` type and `CHAOS` class to display the DNS software information. If `version.bind` returns nothing, we utilize `fpdns` to identify the version. `fpdns` is maintained by the DNS community and covers a wide range of DNS version fingerprints, including mainstream DNS software such as BIND and Unbound.

Table 3: Top 10 regions and AS numbers of the discovered open resolvers during our scanning.

Region	#	%	ASN	#	%
China	665,328	36.7%	4134	245,061	13.5%
USA	142,058	7.8%	4837	128,995	7.1%
India	109,436	6.0%	4847	58,465	3.2%
Russia	82,980	4.6%	17488	53,803	3.0%
South Korea	71,193	3.9%	4538	53,043	2.9%
Indonesia	66,809	3.7%	4766	40,297	2.2%
Brazil	51,904	2.9%	4808	37,304	2.1%
Bangladesh	43,059	2.4%	24560	29,830	1.6%
Iran	41,578	2.3%	58224	29,712	1.6%
Taiwan	27,287	1.5%	45090	24,928	1.4%
# Total regions: 229			**# Total Ases: 25,342**		
# Discovered resolvers: 1,815,017					

Table 4: Software used by the open resolvers.

Software	Identified resolvers		
	`version.bind`	`fpdns`	Total
BIND	35,987	12,420	48,407 (5.7%)
Unbound	9,447	2,265	11,712 (1.4%)
Knot	43	0	43 (0.0%)
PowerDNS	10,787	749	11,536 (1.4%)
Subtotal	56,264	15,434	71,698 (8.4%)
Others	273,458	503,552	777,010 (91.6%)
Total	329,722	518,986	848,708 (100.0%)

Others: Microsoft DNS, Dnsmasq, public DNS services, etc.

Results of scanning. Our scanning ran for 5 days in 2022 and we discovered 1,815,017 open DNS resolvers, including both recursive resolvers and forwarders. We first examined their geo-location and autonomous system (AS) using GeoLite2 database. Table 3 shows that 1.8M resolvers are located in *229* regions with the top three being China (36.7%), USA (7.8%), and India (6.0%). *25,342* AS numbers were associated with these resolvers, showing our list has broad coverage of resolvers.

Out of the 1.8M resolvers, we identified *848,708* (46.8%) returning software information to our probing methods of `version.bind` (329,722, 38.9%) or `fpdns` (518,986, 61.1%), as listed in Table 4. Totally, *71,698* (8.4%) resolvers are potentially impacted by our discovered vulnerabilities, including BIND (48,407), Unbound (11,712), Knot (43), and PowerDNS (11,536). We did not discover resolvers using MaraDNS and Technitium since they do not support the `version.bind` query or have fingerprints in the `fpdns` database. Noticeably, our scanning results give a conservative estimation of the vulnerable resolvers, due to the limitations of our probing methods.