# NCFuzz: Configuration-guided Network Service Fuzzing

XUESONG BAI, University of California at Irvine, USA
HENGKAI YE, Pennsylvania State University, USA
SHENGHAN ZHENG, Dartmouth College, USA
FENGLU ZHANG, China Telecom, China
HONG HU, Pennsylvania State University, USA
ZHOU LI, University of California at Irvine, USA

Network services like FTP and DNS are critical components of modern dependable cyberspace infrastructure. Software fuzzing, especially network protocol fuzzing, is widely used to uncover flaws in these systems. However, conventional fuzzers operate under a single, fixed configuration throughout the fuzzing campaign, leaving the service's rich configuration space unexplored. Incorporating configurations as a dynamic input dimension is challenging due to complex semantics, trigger conditions, and the resulting enlarged search space.

We tackle the problem of finding bugs under non-default configurations, termed ConfBug, by designing a new fuzzer called NCFuzz. The non-default configurations can be esoteric but the administrators may enable them, which cannot be exercised by conventional fuzzers. With the assumption of the availability of software documentation that describes configuration options, NCFuzz leverages two key observations: 1) software documentation contains rich information about configurations; 2) interactions between configuration and network messages can be tracked through code instrumentation and data-flow analysis. Using these insights, NCFuzz uses configuration knowledge and configuration-network-message relation to guide the fuzzer towards new software states. The quality and completeness of the documentation will impact the effectiveness of NCFuzz. Evaluations on six network service implementations show NCFuzz achieves higher coverage than baseline fuzzers. Five ConfBug were discovered during fuzzing.

CCS Concepts: • **Security and privacy → Security protocols**; **Software security engineering**; **Domain-specific security and privacy architectures**.

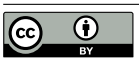
Additional Key Words and Phrases: Software fuzzing, Network services, Configurations

**ACM Reference Format:**
Xuesong Bai, Hengkai Ye, Shenghan Zheng, Fenglu Zhang, Hong Hu, and Zhou Li. 2026. NCFuzz: Configuration-guided Network Service Fuzzing. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA158 (October 2026), 23 pages. https://doi.org/10.1145/3832249

## 1 Introduction

Network services that implement Internet protocols (e.g., FTP, DNS, and HTTP) form the backbone of modern cyberspace infrastructure. These services are widely deployed, persistent, and frequently exposed to untrusted traffic from the Internet. Consequently, vulnerabilities in a single popular implementation can have a destructive impact on network operations. For example, BIND 9 provides

Authors' Contact Information: Xuesong Bai, University of California at Irvine, Irvine, USA, xuesong.bai@uci.edu; Hengkai Ye, Pennsylvania State University, State College, USA, hengkai@psu.edu; Shenghan Zheng, Dartmouth College, Hanover, USA, shenghan.zheng.gr@dartmouth.edu; Fenglu Zhang, China Telecom, Beijing, China, zhangfl15@chinatelecom.cn; Hong Hu, Pennsylvania State University, State College, USA, honghu@psu.edu; Zhou Li, University of California at Irvine, Irvine, USA, zhou.li@uci.edu.




ACM 2994-970X/2026/10-ARTISSTA158
https://doi.org/10.1145/3832249

a feature-rich, RFC-compliant implementation of the Domain Name System (DNS), and reportedly holds 65% of the market share [19]. A security flaw in such a component may affect millions of users worldwide. Moreover, high-profile attacks targeting network services also occur frequently [5, 30].

Given the high stakes, both academia and industry have invested significant efforts in enhancing the security and resilience of these critical applications, among which *software fuzzing* has become a popular technique for automated vulnerability discovery on network protocols [33]. The conventional approach for network fuzzing involves sending a large volume of random inputs (i.e., network messages) and monitoring for abnormal behaviors, such as service crashes [38, 59]. During the fuzzing process, the fuzzer, acting as an attacker, sends mutated network messages to the target implementation. However, this approach treats the service configuration as *fixed*: the fuzzer operates under a single (typically default) configuration throughout the campaign, even though the service often features a high degree of *configurability* (e.g., enabling or disabling modules) that is managed by the *administrator*. The attacker's inputs and the administrator's configurations operate in orthogonal dimensions, and none of the existing fuzzers are capable of mutating them together.

In fact, our survey of network service CVEs (detailed in Section 2.2) reveals that a substantial portion of the CVEs are triggered only under *non-default configurations*, which we define as ConfBug. Despite their prevalence, ConfBug has received limited attention in automated vulnerability discovery, existing fuzzing frameworks (e.g., oss-fuzz [11], profuzzbench [35]) typically rely on a default or single configuration for bootstrapping.

**Challenges in automated discovery of ConfBug.** Automatically discovering ConfBug requires exploring a joint input space consisting of (i) administrator configurations and (ii) attacker-controlled network messages. This setting introduces three key challenges.

**(C1) Complex triggering conditions.** ConfBug are often reachable only under a conjunction of constraints: multiple configuration options must be set to specific values, their *combination* must be valid (due to dependencies and conditional checks), and the service must receive a network request or a sequence of requests that are relevant to the updated configurations.

**(C2) Complex configuration setups.** Real-world network services expose large and heterogeneous configuration spaces (e.g., BIND 9 has hundreds of options) with diverse types (booleans, integers, strings, file paths, and nested blocks) and rich dependencies. Naively mutating options frequently yields invalid configurations that prevent the service from being bootstrapped, wasting fuzzing energy.

**(C3) Compounded input space for fuzzing.** Network messages already induce a vast mutation space due to protocol grammars and statefulness [38]. Adding configuration as a second mutable dimension leads to a combinatorial explosion. A practical fuzzer must prioritize which options to mutate and which message-configuration combinations to test.

**Design overview of NCFuzz.** To address the aforementioned challenges, we design NCFuzz following a simple but under-explored principle: ConfBug are exposed by *pairs* of administrator configurations and attacker message sequences. Accordingly, NCFuzz treats each test case as a paired seed $(C, M)$ and co-evolves the two inputs with runtime feedback and configuration semantics (Figure 2, Section 3.1).

- **Runtime configuration feedback.** NCFuzz instruments the target and performs configuration locating and tracking to recover a mapping between configuration options and their runtime usage sites. This yields an execution-time signal indicating which options are actually exercised and which configuration-governed regions remain unexplored.
- **Semantics-aware configuration synthesis.** NCFuzz parses the official configuration documentation to build a configuration database, enabling *validity-preserving* configuration

generation and mutation, avoiding the common failure mode where random option flips produce non-starting services.
- **LLM-assisted paired seed generation.** Using the extracted configuration database, NCFuzz leverages LLMs to generate diverse seeds consisting of *paired* configuration seeds and message seeds that target configuration-dependent functionality.
- **Configuration-aware fuzzing loop.** Finally, NCFuzz supports configuration-aware mutation and scheduling. It utilizes feedback signals to prioritize new seed pairs, focusing fuzzing energy on message-configuration pairs that are most likely to reach configuration-governed paths and trigger ConfBug.

While a few prior works, like ConfigFuzz [65] and CarpetFuzz [51], incorporate configuration options into the fuzzing search space, neither of them tracks at runtime whether a chosen configuration actually influences the program's execution on a given input, or models the input-configuration interaction. LLM has been used for fuzzing, like ChatAFL [32], where LLM is utilized to enrich seed corpus, and surpass coverage plateau, they target text-based protocols and only focus on network message dimension. We discuss these works in Section 6.

**Evaluation.** We evaluate the performance of NCFuzz on a benchmark consisting of 6 network service implementations (BIND 9, Unbound, LightFTP, ProFTPD, PureFTPD and OpenSSH) under 3 protocols (DNS, FTP and SSH). Adapting NCFuzz to large, complex targets such as BIND 9, Unbound, and OpenSSH requires substantial engineering effort, which we detail in Section 4. We compare NCFuzz against three baseline network fuzzers: AFLNet [38], NSFuzz [39] and ChatAFL [32]. Across most targets, NCFuzz achieves notable improvements in branch coverage, state coverage, and state-transition coverage. For example, compared with AFLNet, NCFuzz improves coverage by 25.64% on average and achieves 28.64× speed-up across the 6 services. The gains are more pronounced on larger codebases, e.g., over 30% branch coverage improvement on Unbound and 112.7× speed-up on BIND 9. We also discovered 5 ConfBug (3 from BIND 9, 1 from Unbound and 1 from LightFTP).

**Limitations and threats to external validity.** The effectiveness of NCFuzz relies on the availability, correctness, and completeness of software documentation that describes configurations. Inconsistent description of configurations will negatively impact the results of NCFuzz. We discuss these issues in details in Section 5.

**Contributions.** Our contributions are summarized below.

- We perform a systematic study of vulnerabilities triggered under non-default configurations, termed ConfBug, and develop the first configuration-guided fuzzer, NCFuzz, to detect them automatically.
- We adapt techniques from text analysis and program analysis to guide NCFuzz to reach code regions governed by configurations.
- We evaluate NCFuzz on a benchmark consisting of 6 implementations of network services. The results demonstrate the effectiveness of NCFuzz.
- We will release the source code of NCFuzz and the new benchmark.

## 2 Background

In this section, we first provide an overview of the software fuzzing workflow and explain how it is adapted to network applications. Then, we describe common practices to configure network services and introduce ConfBug. We also use a real-world example to demonstrate how a ConfBug is exploited.

### 2.1 Software Fuzzing

Fuzzing is a popular testing technique that randomly generates massive inputs and feeds them into the software under test (SUT) to uncover vulnerabilities and unexpected behaviors [33]. *Coverage-based greybox fuzzing* has attracted widespread attention [10, 27, 59] and has led to the discovery of numerous critical vulnerabilities [44]. Given a set of sample inputs, (the *seed corpus*), the fuzzer adds them into an input queue. (1) Based on the *input scheduling* algorithm, it picks the next promising input from the queue, and (2) conducts *random mutations* to generate a new input. 3) Then, the fuzzer executes the program with the newly generated input. The program could be hardened by *security sanitizers (or oracles)* [28, 42, 43] to report abnormal behaviors. (4) If the program crashes or triggers a warning, a bug report will be produced. (5) After execution, the fuzzer checks whether the execution triggers previously unseen code. If so, it will add the input to the input queue for further mutation.

It is not surprising that fuzzing has been used to discover vulnerabilities in network services at scale. For example, AFL can be applied to fuzz network services by using socket to feed input. However, since network services are often stateful and follow complex network protocols, the original stateless AFL cannot test these programs effectively. To maintain and explore states, researchers have proposed leveraging response code [38], response types [29], program variables [2, 39] and weird peers [3] to infer the service states and/or increase the state coverage. To generate syntax-valid network packets, the community developed protocol-specific fuzzers, for targets including TLS[46, 50], DTLS [8], HTTP [16–18], QUIC [40], DNS [63] and intent-based networking [20]. Recent work [32] also leverages large language models (LLMs) to extract protocol grammars and guide network fuzzers.

### 2.2 Service Configurations and Vulnerabilities

To customize a network service to meet specific deployment requirements, network service software often provides configurations outside of its code for service administrators. There are four major sources for configurations. 1) **Configuration files** are used as the major storage to keep configurations. For example, BIND 9 users can set various options via files `named.conf` and `rndc.conf`, to define zones, enable security features like DNSSEC, setup custom plugins, and so on. 2) **Command-line arguments** are for administrators to temporarily configure the service during program bootstrapping. For example, launching BIND 9 with `-u uid` will change user ID of the server to `uid` after completing privileged operations. 3) **Environment variables** allow service administrators to configure a set of binaries serving the same functionalities concurrently. For example, `SSLKEYLOGFILE` in BIND 9 defines the file path for recording TLS pre-master secrets. 4) Some software expose its internal states through **APIs**, which can be updated directly with library calls or external tools while the software is running. An example is rndc [36], which is an external tool to control the BIND 9 server through commands for updating zones, modifying logging modes, and reloading configurations, etc.

In this work, we focus on configuration files as most of the network services we surveyed support this source. For other sources, a service often has replicated options in the configuration files. For example, `ssl_key_log` in `named.conf` replicates `SSLKEYLOGFILE` in environment variables and `options/user` replicates `-u uid` in command-line arguments. Yet, we did find some software that does not support configuration files. For example, Live555 only supports command-line arguments or its APIs. The effort to tailor these cases is expected to be moderate, which mainly involves the generation and mutation of configuration seeds.

Updated configurations usually take effect after the service restarts, but prominent overhead is introduced when fuzzing under various configurations repeatedly. In Section 4.1, we describe a technique to reduce such overhead.

**Definition and survey of ConfBug.** In this work, we aim to discover bugs that are triggered under *non-default configurations* in network services. We focus on the setting that the service administrator chooses a *valid* configuration, and the bug is triggered when the service receives one or a sequence of network messages. We term such bugs ConfBug and we are interested in ConfBug under valid configurations because they are more likely to be exploited by a *network adversary*, who can construct network messages in arbitrary format, but cannot directly manipulate the configurations that are maintained by the administrator.

To understand the prevalence of ConfBug in network services, we conducted a manual investigation of publicly available CVEs in 9 popular open-source network service implementations [35, 63], and summarize the results in Table 1. In total, we found 469 CVEs reported from 1999 to 2024. Among them, 142 were triggered under non-default configurations, accounting for *more than a quarter* of all vulnerabilities. This result highlights the necessity to uncover ConfBug automatically.

## 2.3 An Example of ConfBug

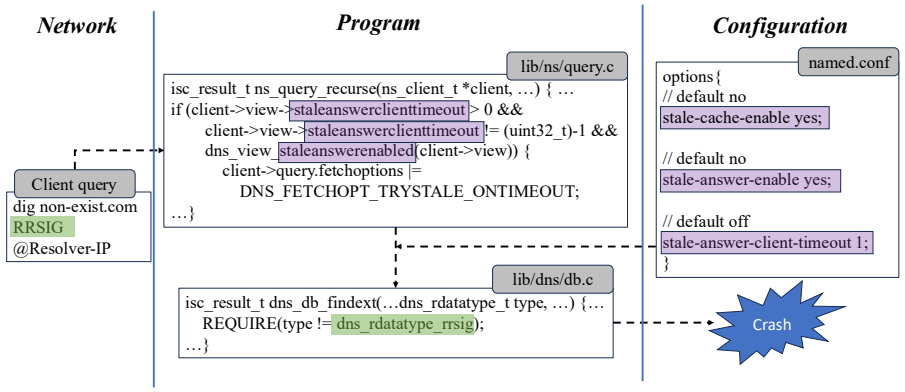



Fig. 1. Details of BIND 9 CVE-2022-3736 (CVSS: 7.5 High). When BIND 9 is configured to handle stale cache (under `named.conf`), an attacker can crash the entire service with *just one malicious client query (*`RRSIG`*)*.

Table 1. Statistics of network service CVEs. Each CVE is manually analyzed. "#All" and "#ConfBug" are count of all CVEs and CVEs triggered by non-default configurations.

| | #All | #ConfBug |
|---|---|---|
| **BIND 9 (DNS)** | 163 | 57 |
| **Unbound (DNS)** | 26 | 3 |
| **Knot (DNS)** | 12 | 6 |
| **Dnsmasq (DNS)** | 38 | 9 |
| **ProFTPD (FTP)** | 46 | 6 |
| **PureFTPD (FTP)** | 12 | 3 |
| **Kamailio (SIP)** | 9 | 2 |
| **OpenSSH (SSH)** | 113 | 45 |
| **Nginx (HTTP)** | 50 | 11 |

We use CVE-2022-3736 of BIND 9 [15] as the example to explain ConfBug. BIND 9 is a popular DNS server software that supports most DNS functionalities (e.g., forwarder, resolver and nameserver). This vulnerability was related to the *stale cache* mechanism introduced to version 9.12.0 in 2018 [14]. Stale cache, proposed in RFC 8767 [21], allows DNS resolvers to answer a DNS query with an expired record when the resolver cannot obtain an updated record from the upstream authoritative servers, so the volume of query failures will be reduced.

To enable this feature, the network administrator needs to edit the BIND 9 configuration file `named.conf`, and set proper values for three options,

namely `stale-cache-enable`, `stale-answer -enable` and `stale-answer-client-timeout`, to `yes`, `yes` and a positive integer (e.g., `1`). To accommodate this feature, BIND 9 changes its function `ns_query_recurse` that processes client's query to search the stale cache, which leverages a function `dns_db_findext` to search DNS records internally in its database. It turns out BIND 9 cannot support all record types simultaneously under this function after the stale cache mechanism is introduced, so assertion checks (`REQUIRE`) are inserted to guarantee certain record types, like `RRSIG`, will not be encountered in certain code path: `RRSIG` is generally sent alongside the corresponding DNS records (e.g., `A`, `AAAA`, `MX`, etc.), so direct `RRSIG` query is uncommon, which hides the bug. However, an attacker can craft a single `RRSIG` query for a *non-existent* domain name (or wildcard domain name) and trick the BIND 9 resolver to search the stale cache, triggering the assertion failure and resulting in severe disruption of services. In Figure 1, we illustrate the vulnerability.

## 3 Design of NCFuzz

In this section, we first overview the challenges of finding ConfBug automatically and how NCFuzz addresses the challenges (Section 3.1). Then, we elaborate the design of each component of NCFuzz (Section 3.2 to Section 3.5).

### 3.1 Challenges and Workflow

Based on our preliminary study, finding ConfBug automatically needs to address three critical challenges (**C1** to **C3**). **(C1) Complex triggering conditions**: as shown in the example of Section 2.3, the vulnerable code can only be reached when 1) the three configuration options (`stale-cache-enable`, `stale-answer-enable` and `stale-answer-client-timeout`) are set to non-default values; 2) their value combination is valid (see the `if` statement in Figure 1); 3) a network message is sent with certain content (a `RRSIG` query for non-existent domain name). Satisfying the conditions altogether is non-trivial. **(C2) Complex configuration setups**: Some configuration setups are quite complex to provide comprehensive support of server functionalities. Take BIND 9 as an example. Based on its document, at least *239* configuration options can be specified [13], covering all sorts of value types (e.g., integers, booleans, strings and even command blocks). Generating a valid configuration with correct syntax and meaningful semantics is non-trivial. Wrong configurations will terminate a service prematurely before being able to process network messages. **(C3) Compounded input space for fuzzing**: Network messages have already introduced a paramount input space for fuzzing due to their unlimited sizes and complex protocol grammars [38]. Configuration introduces another input dimension, blindly mutating all configuration options offers little benefit for bug finding because the mutation space grows explosively.

So far, none of the prior network fuzzers have been able to effectively discover ConfBug across different protocols. Generic network fuzzers like AFLNet [38], NSFuzz [39], ChatAFL [32] and SGFuzz [2], only mutate the network messages, while the configuration remains fixed (typically at its default) throughout the campaign. EnvFuzz is a recent work that can mutate program environments, where configuration is considered one of the environment factors [31]. However, its discovered bugs are all triggered by *invalid* configuration values (e.g., a corrupted `sshd_config` option that crashes OpenSSH), which is different from our definition of ConfBug (see Section 2.2).

We design NCFuzz to explore new code paths related to non-default configurations, aiming to capture ConfBug under fuzzing. Though the trigger conditions are sophisticated (**C1**), the latent connections between code and configurations can be tracked at runtime when the code is instrumented (e.g., with LLVM), and the revealed connections can be leveraged to guide the mutation. To effectively explore the space of valid configurations (**C2**), NCFuzz parses the configuration document of the tested software and extracts key information about its enclosed configuration options with the help of large language models (LLM). Such information aids the generation of

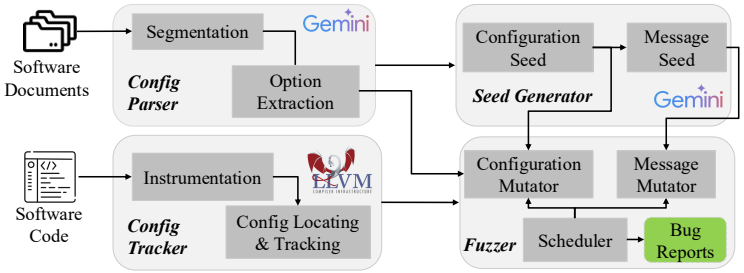



Fig. 2. The workflow of NCFuzz.

configuration seeds and inference of their valid value ranges. To increase the chances of activating functionalities related to custom configurations with network messages (**C3**), we customize the mutator and scheduler of the generic network fuzzer, so that the network input and configuration input are changed in coordination. We elaborate the key components of NCFuzz below, and the workflow is illustrated in Figure 2.

### 3.2 Configuration Locating & Tracking

To improve the effectiveness of ConfBug discovery, NCFuzz allocates additional fuzzing energy to network messages that trigger previously unseen configuration-governed behaviors. We consider a configuration option as *interesting* if it has not been observed in previous executions or if it is used in a previously unexplored program location. Since large network services contain a substantial number of dispersed configuration-usage sites, an automated approach is required to locate these sites and instrument them to collect configuration-related feedback at runtime.

To build an effective tool, we first manually examine several network service applications to understand the common workflow to use configurations. We identify two main patterns as follows. 1) Network service applications generally use a dedicated function to load configurations from external sources, parse them to get correct values, and store them into in-memory objects. We define these functions as *configuration-loading* functions. For example, BIND 9 implements and invokes function `named_config_get` to process configuration files and initialize configuration variables. 2) Configuration options are commonly stored as global variables or structure members. The former is widely used by standalone configuration options, while the latter is for a set of closely related options. For example, in BFTPD, the timeout settings are stored as global variables `control_timeout` and `data_timeout`. Configuration options related to addresses are concentrated in the structure `myaddr`, such as `myaddr.sin_port` and `myaddr.sin_addr.s_addr`. These configurations are propagated along the program logic, and are finally used as an operand of comparison operations (e.g., `if` statement). Based on these two patterns, we implement a configuration tracker to automatically locate and track configuration usage. The program analysis is twofold, corresponding to the two patterns.

**Configuration locating.** The tracker first identifies the callsites of the configuration-loading functions and locates the objects that store the loaded configurations according to the pre-defined rules. Configuration-loading functions are identified with lightweight keyword-based searching (e.g., matching `config`). Listing 1 demonstrates how `stale-answer-client-timeout`, a configuration option in software BIND 9, is loaded from the configuration file. Specifically, the tracker identifies `named_config_get` as the loading function and `cfg_obj_t **obj` as the object that holds the parsed configuration.

The tracker then traverses the LLVM intermediate representation (IR) to locate all callsites of `named_config_get`. For each callsite, the tracker extracts the `name` (e.g., `@.str.691`) to recover the

configuration name and labels the corresponding `obj` value (e.g., `%37`) as the analysis source. Next, the tracker performs data-flow analysis to determine the global variable or structure member that stores the configuration value. In the illustrated example, the source is processed by `cfg_obj_asuint32`, propagated to `%1076` (line 19), and finally stored in `%1078` (line 22). By examining the destination of the `Store` instruction, the tracker infers that the value is written to the 86th member of the `dns_view` structure (line 21), corresponding to `dns_view->staleanswerclienttimeout` in the source code.

```
1  Rule:
2  Function: named_config_get
3  Configuration: (1, char *name)
4  Object: (2, cfg_obj_t **obj)
5  ------------------------------------------------------
6  Source Code:
7  result = named_config_get(maps, "stale-answer-client-timeout", &obj);
8  ...
9  } else {
10     view->staleanswerclienttimeout = cfg_obj_asuint32(obj);
11 }
12 ------------------------------------------------------
13 LLVM IR:
14 @.str.691 = [28 x i8] c"stale-answer-client-timeout\00"
15 ...
16 %1061 = call i32 @named_config_get(%struct.cfg_obj** %1060, i8* getelementptr inbounds ([28 x i8],
   ↪  [28 x i8]* @.str.691, i64 0, i64 0), %struct.cfg_obj** %37)
17 ...
18 %1075 = load %struct.cfg_obj*, %struct.cfg_obj** %37
19 %1076 = call i32 @cfg_obj_asuint32(%struct.cfg_obj* %1075)
20 %1077 = load %struct.dns_view*, %struct.dns_view** %11
21 %1078 = getelementptr %struct.dns_view, %struct.dns_view* %1077, i32 0, i32 85
22 store i32 %1076, i32* %1078
```

Listing 1. Example of configuration identification.

**Configuration tracking.** After establishing the mapping from configuration names to in-memory objects, the tracker scans the LLVM IR to identify accesses to these objects. For example, in Listing 2, the tracker detects that an access to `dns_view->staleanswerclienttimeout` is made by inspecting the structure name and member index in the `GetElementPtr` instruction (line 9). It then conducts a second data-flow analysis from the accessed object until it is used in a comparison operation. In the example, `dns_view->staleanswerclienttimeout` is loaded into `%109` and subsequently gets checked by `switch i32 %109`. The `switch` statement in the LLVM IR corresponds to the first two checks of the `if` statement in the source code. Accordingly, the tracker inserts a runtime function before this `switch` statement to notify NCFuzz that `dns_view->staleanswerclienttimeout` has been checked. As a result, NCFuzz will retain the corresponding input as interesting and allocates more energy to mutating the associated configuration option `stale-answer-client-timeout`.

### 3.3 LLM-aided Configuration Parsing

Although software documents often explain the usage and valid values of configuration options, which can be leveraged to guide the analysis of NCFuzz, parsing the documents is still a non-trivial task with manual analysis or standard NLP tools, due to that the documents are often written in unstructured and narrative format (e.g., the BIND 9 reference [13]). In this task, we explore the usage of large language models (LLM) as a *universal* parser for different software documents, motivated by recent successes of LLM-based software document analysis [47].

```
Source Code:
if (client->view->staleanswerclienttimeout > 0 &&
    client->view->staleanswerclienttimeout != (uint32_t)-1 &&
    dns_view_staleanswerenabled(client->view))
{ ... }
-----------------------------------------------------
LLVM IR:
%108 = getelementptr %struct.dns_view, %struct.dns_view* %107, i64 0, i32 85
%109 = load i32, i32* %108
switch i32 %109, label %110 [
  i32 0, label %116
  i32 -1, label %116        ]
```

Listing 2. Example of configuration tracking.

Initially, we attempted to feed an LLM with the entire software document and let it extract all configuration options and their key attributes. However, we found the quality of parsing degrades as the input document grows, and a similar observation was also reported by Levy et al. [24]. We address this problem by first prompting the LLM to segment the document into blocks, guided by a few examples (i.e., few-shot learning [4]). Each configuration block is expected to describe one option, and we feed each block into the LLM again to extract a list of fields.

Specifically, we consider the fields including: **Name**, **Type** (value types such as boolean, numeric, and duration), **Default value**, **Description**, **Limit** (min and max values), **Relationship** (dependencies with other options), **Format** (e.g., "128-bit random secret"), and **Explanation** (LLM's explanation of the parsing result, following the idea of chain-of-thought [54]). Note that except for **Name** and **Explanation**, all other fields can be empty if they are not specified in the documentation. Listing 4 shows the prompt to parse configuration. As an example, LLM's response after parsing the `edns-tcp-keepalive-timeout` option documented in Listing 3 is shown in Listing 5.

### 3.4 LLM-aided Seed Generation

A configuration-guided fuzzer requires an initial corpus that spans both input dimensions: administrator configurations, and attacker-controlled network messages. Accordingly, NCFuzz constructs *paired* seeds $(C, M)$, where $C$ is a syntactically valid configuration and $M$ is a protocol message sequence that is likely to trigger configuration-governed functionality. The primary objective is to improve exploration in the early fuzzing stage by (i) bootstrapping the target with diverse, valid configurations and (ii) providing messages that reach feature-specific handlers.

**Configuration seed generation.** We generate configuration seeds by prompting an LLM with the structured option database extracted in Section 3.3. To encourage diversity, the prompt conditions the LLM on high-level feature intents (e.g., DNSSEC validation, access control, recursion/forwarding for BIND 9), which biases the generated configurations towards enabling distinct subsystems. The resulting configurations serve as starting points for subsequent validity-preserving mutation. An example prompt is provided in Listing 6.

**Message seed generation.** Given a configuration seed $C$, NCFuzz generates network message seeds that are compatible with $C$ and are likely to traverse configuration-dependent code paths. We observe that LLM alone is still prone to generating messages that fail grammar checks or inter-field dependencies, especially for complex protocols like DNS. Based on the observation, we adopt an LLM-assisted approach similar in spirit to prior work on seed generation (e.g., ChatAFL [32]), but enforce a stronger *output structure* for complex protocols. Specifically, for each protocol we provide

```
# Example of an option description from manual
edns-tcp-keepalive-timeout: <msec>
Overrides tcp-idle-timeout when edns-tcp-keepalive is enabled. If the client supports the EDNS
↪  TCP Keepalive option, If the client supports the EDNS TCP Keepalive option, Unbound sends the
↪  timeout value to the client to encourage it to close the connection before the server times
↪  out.
Default: 120000 (2 minutes)
```

Listing 3. Example of a configuration option from BIND 9 manual

```
# Prompt for option parsing
# System Instruction:
You are an experienced network engineer, Given the software document of {<software>}, summarize
↪  the requested content to json.
# User Prompt
The configuration option is {<option description from manual>}
...
"name": the option name
"description":  a description of this option...{<example>}
"limit": the range for the option value...{<examplar range>}
"type": the value type of the option...{<list of possible types>}
"format": give clear definition of the format of its acceptable value.
"default": the default value of the option... {<instructions to find default value>}
"relationship": list the names of all associated options...{<example>}
...
Please output your summarization in JSON format and provide a separate explanation for each item
↪  in json.
```

Listing 4. One-shot prompt for Configuration Extraction

```
{   "name": "edns-tcp-keepalive-timeout",
    "description": "Overrides tcp-idle-timeout when edns-tcp-keepalive is enabled...",
    "type": "3", # duration
    "default": 120000,
    "limit": null,
    "relationship": ["tcp-idle-timeout", "edns-tcp-keepalive"]}
{<**Explanation:**>}
```

Listing 5. Example response for Configuration Extraction

```
{<Parsed configuration options>}
# system instruction
Suppose you are a network infrastructure engineer, now it's your job to test {<software>} through
↪  fuzzing.
# user prompt
You need to generate a configuration file that can allow the fuzzing target has multiple network
↪  states. Below are some rules to determine whether the configuration can explore more network
↪  states for {<software>}: {<software_requirement>}
# software_requirement example
Mixed Forwarding and Recursive Setup: ...
DNSSEC Validation: ...
Access Controls and ACLs: ...
Logging and Rate Limiting: ...
```

Listing 6. Prompt for Configuration Generation

an abstract message template and prompt the LLM to instantiate the template (i.e., “fill the blanks“) instead of producing free-form packets. This design constrains the LLM output space and improves

the syntactic validity of generated messages. The templates are derived from packets produced in a controlled environment and are translated to concrete packets using protocol libraries such as Scapy [41].

**Seed editing and filtering.** Though we carefully designed prompt templates to generate valid configurations and messages, we still found invalid seeds are produced. Since invalid seeds can dominate fuzzing time by repeatedly failing or preventing the service from starting, NCFuzz performs early-stage filtering on the seeds.

For message seeds, we observe that, the LLM might generate messages violating field syntax or cross-field consistency. We therefore apply lightweight consistency checks for quick filtering. Here we use DNS message as the example. 1) The numerical fields are edited to match the desired size, e.g., 16 bit for DNS transaction ID. 2) The count field is edited to match the number of corresponding records, e.g., `QDCOUNT` in DNS query equaling the number of queried domains in the `Question` section. 3) Each DNS record is edited to satisfy basic syntax, e.g., MX record containing only one domain name and a preference value. After rule-based correction, we decode each candidate message and filter out the messages that fail decoding.

For configuration seeds, static checking is very difficult due to the lack of checkers like protocol libraries on messages, and largely implementation-dependent. We therefore apply conservative repairs (e.g., filling missing values with defaults when available) and then validate configurations dynamically by launching the service and discarding configurations that lead to early termination.

Overall, this procedure yields a seed corpus that is both diverse and execution-ready, which improves initial coverage and provides effective starting points for the configuration-aware mutation and scheduling mechanisms in Section 3.5.

### 3.5 Configuration-aware Mutation and Scheduling

As highlighted in Section 3.1, the large message-configuration input space poses unique challenges in discovering ConfBug. To address this challenge, we designed new mutation and scheduling mechanisms, guided by other components of NCFuzz. The sub-components are elaborated below.

**Configuration mutator.** We rely on configuration mutation to explore the input space of the configuration dimension. We design the mutation mechanism based on the insights that given network message $M$, mutating the configuration options $C_{Tracked}$ on the code paths that process it, which are recorded by the configuration tracker, has higher chances to uncover new code paths related to configurations. Hence, when configuration mutation is supposed to happen, we compute the set difference between $C_{Tracked}$ and the options enclosed by the current configuration seed $C$, and choose a random option $c \in C_{Tracked} \setminus C$ as a candidate for mutation.

Though we design the configuration tracker to discover all related options during execution, we found some options are not tracked (discussed in Section 5). On the other hand, we found the **Relationship** field returned by our document parser describes the option dependencies and we augment $C_{Tracked}$ with this information to mitigate false negatives. Assuming $C_{Extractor}$ represents the set of dependent options for all options in $C_{Tracked}$ based on the document parser, the mutated option $c \in C_{Tracked} \setminus C \cup C_{Extractor}$.

For each option, depending on its inferred type, we apply different mutation strategies. For a **boolean** option, its value will be flipped based on how boolean true and false are defined (e.g., *"yes"* and *"no"*, *"on"* and *"off"*). For **numeric** option and **duration** option, the values are chosen from either 1) the **Default value** field, or 2) from the series of power of 2 within the **Limit** field if it is not empty or the type value range (e.g., 0 to 4,294,967,295 for unsigned integer) if otherwise. For **string** options, when valid values are inferred, a random one will be chosen from them. We also leverage LLM to generate sample values based on the **format** field, e.g., 128-bit DNS cookie string.

Similarly for **structured** options like IP address and ACL list, we leverage the LLM to generate the valid values. Some options point to resources that have to be existent, e.g., `directory` and `cert-file`, and we fill their values with pre-defined constants.

**Algorithm 1:** Pseudo-code for the scheduler.

```
1  Input: C configuration seeds, M message seeds, P service
     program, protocol state machine S
2  Output: MC_x interesting seed pairs
3  repeat
4      State s ← ChooseState(S) ;
5      (Message M, Config C) ← ChooseSeed(C, M, s) ;
6      for i from 1 to AssignEnergy(M) do
7          foreach option o_j ∈ ListOptions(C) do
8              C' ← ConfigMutate(o_j, C) ;
9              R ← SendToServer(P, M, C') ;
10             if Sanitizers(P, M, C') then
11                 MC_x ← MC_x ∪ {(M, C')} ;
12             else
13                 if IsInteresting(M, C', P, S) then
14                     Score s ← UpdateScore(M, C') ;
15                     AddSeed(M, C', s, M, C) ;
16                     S ← UpdateStateMachine(S, R) ;
17                 end
18             end
19         end
20         M' ← MessageMutate(M) ;
21         R ← SendToServer(P, M', C) ;
22         if Sanitizers(M', C, P) then
23             MC_x ← MC_x ∪ {(M', C)} ;
24         else
25             if IsInteresting(M', C, P, S) then
26                 Score s ← UpdateScore(M', C) ;
27                 AddSeed(M', C, s, M, C) ;
28                 S ← UpdateStateMachine(S, R) ;
29             end
30         end
31     end
32 until timeout reached or abort-signal;
```

**Network message mutator.** Given a configuration, we select the message seeds that are paired with it, based on the output of the seed generator, and we mutate the seeds following the strategy of AFLNet [38], which has proven effective in exploring protocol states. In essence, AFLNet keeps a protocol state machine (a state is represented by response code, timeout/crash outcome, and the encoded coverage map) for each tested protocol when fuzzing. It puts high priority to the network messages leading to new or rare state transitions. AFLNet mutates *sequences* of messages and it inherits mutation operators derived from AFL like bit flip and introduces protocol-specific mutations like message replacement.

**Input scheduler.** Since the seed generator generates pairs of configuration $C$ and messages $M$, we *alternate* configuration mutation and message mutation within each fuzzing round. We adopt this alternation strategy for two reasons. First, although the number of protocol message variants well exceeds the number of configuration options, each configuration mutation is substantially more impactful (it can unlock entirely new code regions). Alternating an option ensures that newly activated configuration paths are exercised by message mutations promptly, maximizing the synergy between the two dimensions. Second, our feedback-driven scoring mechanism naturally balances exploration: if a configuration mutation yields no new coverage, the resulting seed receives a low score, and future energy is redirected to message mutations, effectively implementing an adaptive ratio without explicit weighting.

We design the scheduler based on AFLNet, and below we focus on describing our changes. Algorithm 1 shows the pseudo-code of the scheduler. First, we follow AFLNet's logic of choosing a message sequence seed $M$ based on the observed protocol state and a configuration $C$ paired with it. Energy (i.e., mutation rounds) will be assigned to $M$, and for each round, we start with configuration mutation, which enumerates every option $o_j \in \text{ListOptions}(C)$, where $\text{Option}(C)$ represents the set of options under $C$, and mutates $o_i$. Assuming $C'$ is generated after mutating $o_i$, and we will send the new pair $MC_1 = (M, C')$ to the network service and observe whether new configuration options are triggered based on the configuration tracker. If so, the *highest* seed score will be assigned to $MC_1$, encouraging our fuzzer to choose $MC_1$ to discover more configuration

options. The seed score will still be increased if new code paths, including new functions and new code branches, are discovered. $MC_1$ will be saved to the seed queue if new configurations or paths are discovered.

After the configuration mutation, message mutation will be applied, which derives $M'$ from $M$, and we send $MC_2 = (M', C)$ to the service. The score assignment logic is the same as the configuration mutation, and $MC_2$ will be saved to the seed queue if it leads to interesting discoveries.

The finish of message mutation concludes one fuzzing round, and the service response will be used to update state machine and re-assign fuzzing energy. During fuzzing, interesting execution outcomes flagged by the sanitizers, like crash, will always be reported to the human analyzer.

## 4 Evaluation

We examine the following research questions:

**RQ1:** How effective is NCFuzz in increasing code coverage?
**RQ2:** How does each component contribute to the overall performance of NCFuzz?
**RQ3:** Can NCFuzz discover ConfBug?

### 4.1 Evaluation Setup

**Implementation details.** For configuration tracking (Section 3.2), we implement a customized LLVM pass to analyze configuration-related code and instrument the system under test (SUT) during compilation. The customized instrumentation will slightly increase the binary size (less than 1MB for our SUTs). We edit the mutation and scheduling code of AFLNet to support our approach described in Section 3.5. The components that interact with the LLM are written in Python. NCFuzz has 24k LoC in C/C++ and 4.8k LoC in Python in total. We use the same sanitizers as AFL, which detect crashes and hangs. For each reported crash or hang, we manually verify whether it is indeed a ConfBug.

Table 2. The list of subjects under test (SUT).

| Protocol | Software | #LoC | Version |
|---|---|---|---|
| DNS | BIND 9* | 421k | 9.18.0 |
| | Unbound* | 204k | 1.22.0 |
| FTP | LightFTP | 4.7k | 5980ea1 |
| | ProFTPD | 242k | 61e621e |
| | PureFTPD | 32k | 10122d9 |
| SSH | OpenSSH* | 144k | 1c0d813 |

*: Newly integrated or significantly updated.

We choose Google Gemini 1.5 Pro as the LLM [12], because it has large input and output token limits (1,048,576 and 8,192) that are sufficient to process every configuration document we use (the longest is the BIND 9 reference [13], which has around 71K tokens). For seed generation, we command NCFuzz to generate 150 configuration seeds for the tested protocols (DNS, FTP and SSH). From each configuration seed, we ask the LLM to generate 3 paired message seeds.

Changing configurations often results in service restart and we try to alleviate the overhead by taking advantage of the AFL forkserver mechanism [60]. Specifically, we insert the function `__AFL_INIT()` into a specific region in the SUT right before the SUT reads configurations. Therefore, each time a seed is tested, the AFL forkserver will fork a child thread which starts at `__AFL_INIT()`, reading configurations.

**Systems under test (SUT).** Table 2 presents the SUT used in our evaluation. All of these software systems support configuration files. The 3 FTP implementations and OpenSSH are also tested in ProFuzzBench [35], a widely used benchmark for evaluating network fuzzers. We spend prominent efforts in adapting BIND 9 and Unbound to NCFuzz and other baseline fuzzers. For OpenSSH, we found the OpenSSH version used by ProFuzzBench is quite old, so we adapted the newer OpenSSH, which also incurred prominent engineering efforts.

**Baseline fuzzers.** For the baseline fuzzers to compare with NCFuzz, we selected AFLNet [38], NSFuzz [39], and ChatAFL [32]. AFLNet is a popular stateful protocol fuzzer, also integrated by ProFuzzBench. NSFuzz extends AFLNet by identifying state variables through static analysis. ChatAFL [32] extends the framework of AFLNet and leverages LLM for seed generation and mutation oracle. While there are other network fuzzers as surveyed in Section 2.1, adapting our SUTs to them or running fair comparison is more challenging. For example, SGFuzz [2] is based on LibFuzzer [27], a very different fuzzing architecture.

**Environment.** We use one machine to conduct the experiment in this paper. The machine is equipped with two AMD EPYC 9354 32-Core Processors, 756G memory, Ubuntu 22.04 OS.

### 4.2 Coverage Analysis

We fuzz each SUT for 24 hours per trial with the 4 fuzzers: NCFuzz, AFLNet, NSFuzz, and ChatAFL. Ten trials are run for each SUT and we report the average metrics, including code coverage, state coverage, state transition coverage, and configuration coverage, as detailed below. The first three metrics are commonly used by network protocol fuzzers (e.g., [32]), while the last one is unique to our problem setting.

Table 3. Average Branch Coverage for 10 runs of 24 hours. "Improv" is short for Improve.

| Subject | NCFuzz | Comparison with AFLNet | | | | Comparison with NSFuzz | | | | Comparison with ChatAFL | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | AFLNet | Improv | Speed-up | $\hat{A}_{12}$ | NSFuzz | Improv | Speed-up | $\hat{A}_{12}$ | ChatAFL | Improv | Speed-up | $\hat{A}_{12}$ |
| BIND 9 | 23190.6 | 20453.7 | 13.38% | 112.7× | 1.00 | 16240.8 | 42.79% | 31.6× | 1.00 | N/A | N/A | N/A | N/A |
| Unbound | 6524.4 | 4983.7 | 30.93% | 5.57× | 1.00 | 4435.2 | 47.10% | 10.82× | 1.00 | N/A | N/A | N/A | N/A |
| LightFTP | 184 | 117.7 | 56.33% | ∞ | 1.00 | 387.6 | -52.53% | N/A | 0 | 119 | 54.62% | ∞ | 1.00 |
| ProFTPD | 5806.4 | 5121.1 | 13.38% | 8.31× | 1.00 | 5096.3 | 13.93% | 8.53× | 1.00 | 5065.2 | 14.63% | 9.36× | 0.92 |
| PureFTPD | 1199.22 | 1061.3 | 13.00% | 1.67× | 0.98 | 1059.3 | 13.21% | 1.94× | 0.98 | 996.3 | 20.37% | 1.00× | 0.97 |
| OpenSSH | 3493.3 | 2756.3 | 26.83% | 14.96× | 1.00 | 2846 | 22.74% | 17.68× | 1.00 | N/A | N/A | N/A | N/A |

N/A: for ChatAFL, it cannot generate valid DNS and SSH messages; for Speed-up of NSFuzz, NCFuzz has lower coverage, the value becomes invalid.
∞: when NCFuzz covers the same number of branches at first round, speed-up becomes ∞.

Table 4. Average number of states for 10 runs of 24 hours. BIND 9 and Unbound are not shown because the DNS states cannot be counted by the fuzzers.

| Subject | NCFuzz | AFLNet | Improv | NSFuzz | Improv | ChatAFL | Improv |
|---|---|---|---|---|---|---|---|
| LightFTP | 24 | 23.4 | 2.56% | N/A | N/A | 23.7 | 1.26% |
| ProFTPD | 27 | 20.8 | 29.8% | 23.3 | 15.88% | 27.5 | -1.82% |
| PureFTPD | 28.75 | 27.3 | 5.31% | 22.2 | 29.5% | 28.9 | -0.51% |
| OpenSSH | 117 | 59 | 198.30% | 75.25 | 55.48% | N/A | N/A |

N/A: for ChatAFL, no valid SSH messages can be generated; for NSFuzz, its states cannot be counted under LightFTP.

**Branch coverage.** Table 3 shows average branch coverage achieved by the fuzzers. To assess the improvements brought by NCFuzz, the results include the percentage increase in branch coverage (*Improv*), the time comparison in reaching the same branch coverage (*Speed-up*), and the effect size indicated by the metric $\hat{A}_{12}$ [1], are also reported.

NCFuzz surpassed the baselines across all SUTs in most cases. Take BIND 9 as an example. NCFuzz covered 13.38% more branches than AFLNet, also achieving this faster with a 112.7× speed-up. When compared to NSFuzz, NCFuzz showed a substantial improvement of 42.79% in branch coverage on the same SUT, with a significant speed-up of 31.6×. For Unbound, NCFuzz improved branch coverage by 30.93% over AFLNet and 47.10% over NSFuzz. The effect sizes, particularly $\hat{A}_{12} = 1.00$, reinforce NCFuzz 's performance advantage against both baselines in terms of branch coverage and speed. The only exception is LightFTP, on which NSFuzz achieves 2× more branch coverage than NCFuzz, though NCFuzz is still better than AFLNet and ChatAFL. Because LightFTP

Table 5. Average number of state transitions for 10 runs of 24 hours. BIND 9 and Unbound are not shown because the DNS states cannot be counted by the fuzzers.

| Subject | NCFuzz | Comparison with AFLNet | | | | Comparison with NSFuzz | | | | Comparison with ChatAFL | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | AFLNet | Improv | Speed-up | $\hat{A}_{12}$ | NSFuzz | Improv | Speed-up | $\hat{A}_{12}$ | ChatAFL | Improv | Speed-up | $\hat{A}_{12}$ |
| LightFTP | 184.25 | 189.5 | -2.77% | N/A | 0.52 | N/A | N/A | N/A | N/A | 204.7 | -10.00% | N/A | 0.06 |
| ProFTPD | 265.3 | 147.3 | 80.10% | 43.45× | 1.00 | 164.6 | 57.35% | 45.74× | 1.00 | 262.6 | 1.02% | 2.04× | 0.44 |
| PureFTPD | 263.55 | 221.7 | 18.88% | 3.28× | 0.87 | 149.8 | 75.93% | 13.94× | 1.00 | 276.1 | -4.55% | N/A | 0.37 |
| OpenSSH | 139.4 | 63.8 | 218.49% | 8.84× | 1.00 | 81.5 | 71.04% | 8.38× | 1.00 | N/A | N/A | N/A | N/A |

N/A: for ChatAFL, no valid SSH messages can be generated; for NSFuzz, its states cannot be counted under LightFTP; when NCFuzz has smaller coverage, N/A is filled.

has fewer configuration options, expanding configuration coverage has limited impact on its results. Conversely, NSFuzz emphasizes exploring network states, therefore performs better when configuration exploration is less critical. Notably, ChatAFL cannot generate valid DNS and SSH messages because it directly uses LLM's responses as seeds and grammar.

**State coverage.** Table 4 presents the average number of states detected by the fuzzers. For all fuzzers, the SUT states are inferred from the response code, while NSFuzz also uses static analysis to extract states from internal state variables. Yet, for BIND 9 and Unbound, none of the fuzzers are able to collect state statistics, likely due to that DNS is a stateless protocol, we therefore focus on the other SUTs. NCFuzz outperformed AFLNet and NSFuzz across the remaining SUT, particularly OpenSSH: NCFuzz's coverage is 198.3% higher than that of AFLNet, and it achieved a 55.48% increase compared to NSFuzz. The results suggest some service states do depend on custom configurations. ChatAFL has state coverage very similar to NCFuzz, due to the contributions of its LLM design in state exploration.

**State transition coverage.** Table 5 shows the average state transitions, which reflects how actively a fuzzer explores protocol states. Compared to AFLNet and NSFuzz, NCFuzz achieves the best results except on LightFTP (a slight decrease of -2.77%). It has slightly worse coverage than ChatAFL on FTP software (e.g., -10% decrease on LightFTP).

Table 6. Average number of triggered configuration options.

| Software | Triggered Options | All Options |
|---|---|---|
| BIND 9 | 43 | 64 |
| Unbound | 187 | 215 |
| LightFTP | 11 | 12 |
| ProFTPD | 26 | 26 |
| PureFTPD | 47 | 48 |
| OpenSSH | 85 | 94 |

**Configuration option coverage.** Through generating configuration seeds, tracking configurations at runtime, and mutating configuration options, we expect more options to be activated under NCFuzz. Table 6 shows the average number of triggered options and all effective options. Note that for BIND 9, we focus on functionalities related to serving DNS queries, so only a subset of options (64) are counted as "all options". The results show that NCFuzz is able to trigger most options for any software, in some cases reaching 100% trigger rate (e.g., ProFTPD).

**Answering RQ1**

NCFuzz demonstrates greater branch coverage than all baselines (e.g., 25.64% more improvement and 28.64x speed-up than AFLNet) and greater state and transition coverage than AFLNet and NSFuzz. It triggers 86.9% configuration options.

### 4.3 Ablation Study

**Contribution by components.** To evaluate how each component in NCFuzz contributes to the overall performance gain of fuzzer, we conduct the following study. Since NCFuzz mainly extends

Table 7. Ablation study of branch coverage for 10 runs.

| Software | CL0 | CL1 Improv | CL2 Improv | CL3 Improv |
|---|---|---|---|---|
| BIND 9 | 20453.7 | 12.54% | 5.27% | 13.38% |
| Unbound | 4983.7 | 26.87% | 7.99% | 30.93% |
| LightFTP | 117.7 | 48.98% | 1.43% | 56.33% |
| ProFTPD | 5121.1 | 5.88% | -1.7% | 13.38% |
| PureFTPD | 1061.3 | 8.83% | 2.23% | 13.00% |
| OpenSSH | 2756.3 | -21.23% | 16.10% | 26.83% |

AFLNet with different designs, we treat the original AFLNet as the base system and gradually add other components. Below we list the 4 modes of NCFuzz.

- **CL0**: The original AFLNet.
- **CL1**: The original AFLNet with configuration tracking, mutation and scheduling ($D_1$).
- **CL2**: The original AFLNet with LLM-generated seeds ($D_2$).
- **CL3**: AFLNet with $D_1$ and $D_2$, which forms NCFuzz.

Table 7 shows the improvement in branch coverage. Due to the space limit, we list only the *Improve* metric for CL1, CL2, and CL3, based on comparison with CL0.

For **CL1**, the branch coverage shows that all evaluated SUTs could benefit from this design except OpenSSH. The largest improvements are observed on BIND 9 and Unbound, with 12.54% and 26.87% branch coverage increase. In fact, BIND 9 and Unbound have large codebases (both over 200K LoC) and many configuration options, which might benefit more from configuration-oriented tracking, mutation, and scheduling. LightFTP, ProFTPD, and PureFTPD benefit only slightly from design $D_1$, as the FTP protocol is relatively simple and configuration options are not central to FTP protocol logic.

For **CL2**, the improvements are moderate for BIND 9, Unbound, LightFTP, and PureFTP. There is even a decrease on ProFTPD. Although the quality of seed corpus is critical to some fuzzers, it is not crucial to NCFuzz and AFLNet by itself. OpenSSH does observe a high increase (16.10%), as stateful mutation alone is insufficient to cover states across diverse functionalities.

With **CL1** and **CL2** combined (**CL3**), all SUTs could benefit to reach higher coverage and faster discovery. Interestingly, for OpenSSH, by adding LLM-generated seeds (**CL2**), the negative impact caused by **CL1** (-21.23%) is reversed (26.83%), suggesting that starting from high-quality seeds amplifies the effectiveness of mutation and scheduling.

Table 8. Branch coverage from naive two-step fuzzing method.

| Software | AFLNet | Two-step naive method | | | NCFuzz |
|---|---|---|---|---|---|
| | | Lowest | Highest | Average | |
| BIND 9 | 20453.7 | 13111 | 19041 | 16973.3 | 23190.6 |
| Unbound | 4983.7 | 5660 | 6633 | 6437.8 | 6524.4 |
| LightFTP | 117.7 | 180 | 180 | 180 | 184 |
| ProFTPD | 5121.1 | 3400 | 5729 | 4523.5 | 5806.4 |
| PureFTPD | 1061.3 | 227 | 805 | 474.8 | 1199.22 |
| OpenSSH | 2756.3 | 1468 | 2810 | 2631.9 | 3493.3 |

**Contribution from the combined configuration-message fuzzing.** We developed a non-trivial configuration-message fuzzing strategy to explore the compounded fuzzing input space. On the other hand, a straightforward approach could be implementing a simpler *two-step* strategy. First, the fuzzer generates configuration files independently. Then tests the service running generated

configurations with a standard network fuzzer. Here, we compare our approach with such a naive approach to assess its contribution.

Specifically, we use Gemini 2.0 Flash to generate diverse valid configurations independently[1], and then run AFLNet under each generated configuration, without runtime configuration tracking or coordinated mutation. To ensured a fair comparison, we distributed 240 cpu-hours (in the main experiment, each fuzzer runs 10 simultaneous instances for 24h) to 10 validated configurations for each software, and collected the branch coverage as shown in Table 8.

While this baseline avoids the complexity of co-evolving configurations and messages, it misses the key synergy that NCFuzz exploits: the runtime feedback from the configuration tracker guides *which* options to mutate based on message-level execution, and vice versa. Compared to NCFuzz (CL3), the average branch coverage of this approach degrades for all SUTs.

On the other hand, adding non-default configuration into seeds does increase the state coverage for most cases. We count the number of states for the two-step method, and 23.2, 26.7, 28.4 and 105 states are reached for LightFTP, ProFTPD, PureFTPD and OpenSSH in average. Compared to the AFLNet column Table 4, only the number for PureFTPD drops (from 23.4 to 23.2).

**Quality of generated seeds.** Here we assess the performance of LLM, in particular Gemini, for seed generation tasks. For configuration generation, only 6 out of 150 seeds are invalid. The major issue is missing option values. For example, BIND 9 supports alias names for configuration options, which leads to an alias option (e.g., `allow-query {allow-list;};`) for a real option with a concrete value (e.g., `acl "allow-list {127.0.0.1;}"`). LLM might not be able to generate a valid option pair.

For message generation, 100 out of 450 seeds are invalid. The major issue is message format. For instance, the field `dns_header` is missing from a DNS message seed, or a string value is assigned to an integer field. Our message editor is able to fix 76 seeds with simple heuristics, leaving 24 seeds unusable (22 DNS seeds and 2 FTP seeds).

**Overhead and monetary cost of LLM.** Using LLM introduces extra latency and fees charged by the LLM providers, but we found the cost is acceptable. With Gemini 2.0 Flash model ($0.1 / M input tokens, $0.4 / M output tokens), the total cost incurred for documentation parsing and seed generation was $0.3 per software. The LLM-related tasks could finish within 10-20 minutes, including document parsing and seed generation & validation.

**Answering RQ2**

The improvement from configuration-oriented tracking, mutation, and scheduling is higher than that from LLM-generated seeds (on average 13.64% vs. 5.22% branch coverage increase), while their combination amplifies the improvement (25.64%). 96% of configuration seeds and 94.7% of message seeds generated by the LLM are usable either directly or after simple editing.

### 4.4 Discovered ConfBug

Here we describe the ConfBug that are confirmed after we inspect the crashes and hangs. None of the ConfBug can be identified by the baseline fuzzers. The three BIND 9 bugs matched prior CVEs while the LightFTP and Unbound bugs are new at the time of fuzzing.

- **BIND 9 crash #1.** NCFuzz rediscovered CVE-2022-3736 (CVSS: 7.5) on BIND 9, which has been described in Section 2.3, by generating the `RRSIG` query and mutating the configuration seed to set the 3 options to their designated values.

[1]Gemini 1.5 Pro is unavailable at the time of this experiment so we choose its sibling version.

- **BIND 9 crash #2.** For BIND 9, the configuration option `nxdomain-redirect` is used to append a specified suffix to the queried domain name, if the name does not exist. However, when it is enabled and BIND 9 receives a query for reverse zone record (i.e., `PTR` record), the software would crash immediately. NCFuzz discovered this bug and we found it matches the description of CVE-2023-5517 (CVSS: 7.5).
- **BIND 9 crash #3.** If the stale cache feature is enabled in BIND 9 and option `stale-answer-client-timeout` is set to a non-zero value, BIND 9 will crash unexpectedly when it receives more queries than the `recursive-clients` quota allows. NCFuzz discovered this bug and we found it matches the description of CVE-2022-3924 (CVSS: 7.5).
- **LightFTP crash.** For LightFTP, there is one configuration option `maxusers` to limit the maximum number of connections on the LightFTP server. When its value is set to 0, the server will not allow any connections, but keep the thread running. This bug is introduced by one patch from ProFuzzBench that causes the thread to crash at one connection attempt[2].
- **Unbound hang.** For Unbound, when DNS queries go through TCP connections (non-default, as DNS mostly goes through UDP), a large, malformed DNS message will cause the Unbound service to keep the TCP connection open and occupy a port for an extended period, which leads to prominent resource consumption. When we were about to report this bug, we found it was reported by another party (listed as an issue of Unbound repo).

**Answering RQ3**

NCFuzz can discover new ConfBug and rediscover ConfBug with complex trigger conditions.

## 5 Discussion

Here we discuss the limitations of this work and future plans. First, while LLM can extract configuration options accurately and generate valid seeds in most cases, we do observe some errors, which we have summarized in Section 4.3 ("Quality of generated seeds"). Combining results from different LLMs might alleviate this issue. Another potential solution is LLM fine-tuning, which has shown promise in aligning an LLM (e.g., Llama) for network applications [56].

```
# False negative due to local variable obj.
result = named_config_get(maps, "stale-cache-enable", &obj);
INSIST(result == ISC_R_SUCCESS);
if (cfg_obj_asboolean(obj)) {
    obj = NULL;
    result = named_config_get(maps, "max-stale-ttl", &obj);
    INSIST(result == ISC_R_SUCCESS);
    max_stale_ttl = ISC_MAX(cfg_obj_asduration(obj), 1);
}
# ----------------------------------------
# False negative due to loading DATAPORT20.
if (!strcasecmp(config_getoption("DATAPORT20"), "yes")) {
    ...
    local.sin_port = htons(20);
}
```

Listing 7. Two false negatives by the configuration tracker.

[2]We tried to contact the maintainers but found the repo is inactive now.

Second, though the configuration tracker is capable of tracking most configurations, there remain a few cases that need to be supported. An example is shown in Listing 7, though `max-stale-ttl` is on the execution path from `stale-cache-enable`, it is missed because `stale-cache-enable` is loaded into a local variable `obj` and impacts `max-stale-ttl` through control flow. Listing 7 shows another example of loading `DATAPORT20` from the configuration file. The loaded configuration is compared to the string *"yes"* and the result, a boolean value, is directly used as a condition flag in the `if` statement. Subsequently, `local.sin_port` is assigned the value 20. There is no direct dataflow between `DATAPORT20` and `local.sin_port`, and the tracker fails to identify the in-memory object associated with this configuration option. To support these cases, we could leverage LLM to analyze the code near the configuration-loading function callsite when no direct dataflow is reported. In fact, some recent works have shown promise of LLM-aided code analysis [34, 64].

Third, NCFuzz assumes the availability of sufficiently detailed software documentation that can be parsed by an LLM. Among our six SUTs, all provide official configuration references, which is common for mature network services. In practice, documentation may be incomplete, outdated, or incorrect (e.g., a few such errors in BIND 9's reference [13] were inconsistent with the implementation, which led to invalid configuration seeds). For software without high-quality documentations, the quality of extracted options and generated seeds would degrade, limiting the effectiveness of NCFuzz. A potential mitigation is to combine documentation with static analysis to reduce reliance on external documents.

Finally, because NCFuzz is inherently designed to target bugs triggered under non-default configurations, the discovered bugs may have limited real-world exploitability if administrators seldom enable the corresponding features. However, we argue that once a configuration option is documented and supported, administrators *can* enable it, and finding and fixing its related bugs are necessary. Besides, the three BIND 9 CVEs we rediscovered all received high severity ratings (CVSS 7.5) and are relevant to important DNS features (e.g., stale cache and NXDomain).

## 6 Related Work

In Section 2 and Section 3.1, we review the closely related works in network fuzzing and network configurations. Here we review other related works.

**LLM for vulnerability discovery and fuzzing.** In this work, we leverage LLM to parse software documents and generate fuzzing seeds. Recently, LLM has also shown preliminary successes in vulnerability detection. Researchers have attempted to devise effective prompts to guide LLMs in vulnerability detection with zero-shot and few-shot learning [9, 66]. Retrieval-augmented generation (RAG) was leveraged to further improve the effectiveness by augmenting the LLM with knowledge of existing vulnerabilities (e.g., CVEs) [7, 67]. Beyond the pre-trained models, fine-tuning LLMs with more relevant vulnerability samples has been explored [6, 22, 55, 57, 58]. Program analysis has also been leveraged to extract structural features of a program to enhance LLM's understanding of the code semantics [26, 37, 52, 62].

For LLM-assisted fuzzing, ChatAFL [32] leverages LLM to extract protocol grammars and generate message seeds for network fuzzing. ChatAFL targets *text-based* protocols (RSTP, FTP, SMTP) where the LLM can directly produce syntactically valid messages. However, for protocols with *binary wire formats* (e.g., DNS, SSH), we found that ChatAFL's approach cannot generate structurally valid inputs because binary encodings, checksums, and length fields cannot be reliably produced by text generation alone. ChatAFL focuses on utilizing LLMs to build protocol message grammars, improve message validity, and surpass coverage plateau, while NCFuzz focuses on using LLM to parse *configuration documentation*, extract structured configuration knowledge, and generated *paired* configuration-message seeds that target configuration-dependent functionality. NCFuzz also supports generating binary-format protocols with protocol-specific libraries, rather than relying

on LLM text generation only. The novelty lies not in applying LLMs to fuzzing per se, but in targeting the configuration dimension and using extracted configuration semantics to co-evolve configurations and messages with runtime feedback.

**Mis-configuration detection.** This area can be confused with the target of our project. Previous works have shown that human errors in configurations widely exist in deployed systems like databases and routers, and techniques such as code analysis [48, 49, 61] and fuzzing [25, 45] have been applied. One type of mis-configuration, namely *silent misconfiguration* [61], appears to be similar to ConfBug. In this case, the mis-configuration does not lead to error message or exception and the program can boot successfully, though logic-level bugs emerge later through implicit interactions. ConfBug encompasses silent misconfiguration, as code bugs related to correct configurations are also considered.

**Fuzzing under command-line options.** A few works explored how to fuzz a program under various command-line options (e.g., `-l` and `-F`) [23, 51, 53, 65]. For example, OSmart constructs option impact graph by analyzing the program code, which guides the generation of valid command-line options for fuzzing [53]. We note that program options are often provided along with command-line input, so they are in the same dimension. But in our case, the inputs and configurations are in different dimensions, provided by different parties (i.e., attacker and service administrator).

ConfigFuzz [65] is the closest prior work that incorporates configuration options into the fuzzing search space. It performs *syntactic integration*, which encodes options as bytes alongside the input without modeling the *runtime interaction* between configurations and inputs. NCFuzz differs from ConfigFuzz in three key aspects: (1) NCFuzz targets *network services* where inputs are protocol message sequences, not single file inputs, introducing challenges in coordinating configuration changes with protocol state exploration; (2) NCFuzz leverages LLM-based document parsing to extract structured configuration knowledge (types, ranges, dependencies), whereas ConfigFuzz relies on simpler command-line option formats; and (3) NCFuzz introduces runtime configuration tracking to provide feedback on which configuration is used, and which configuration-governed paths are exercised, creating a *closed feedback loop* between configuration mutation, message mutation, and coverage guidance that ConfigFuzz does not provide.

**Program option extraction.** A similar task is to extract the relations *between* program options (e.g., conflicts and dependencies) from documentation, which was explored by CarpetFuzz [51]. Program options reside in the same dimension, but we aim to learn the input-configuration relations from two different dimensions, which is more challenging.

## 7 Conclusion

To aid administrators in adjusting the functionalities of network services during runtime, configuration files are typically provided. The custom configurations written by the administrators introduce complex trigger conditions in front of the vulnerabilities, which complicates their discovery by software fuzzers that operate under a fixed configuration. We found that vulnerabilities under custom configurations, or ConfBug, are not unusual, and we developed the first system, NCFuzz, to find ConfBug at scale, building on the recent advances in LLM and program analysis (LLVM-based instrumentation). The evaluation results on 6 service implementations covering 3 major protocols, shows that NCFuzz achieves notable improvements in exploring code paths related to non-default configurations. In particular, NCFuzz achieves higher branch coverage than 3 baselines (AFLNet, NSFuzz, and ChatAFL) and higher state and state-transition coverage compared to AFLNet and NSFuzz. Five ConfBug were discovered by NCFuzz. The effectivenss of NCFuzz relies on the availability of software documentation, the quality, and completeness of the documentation may impact the outcome of NCFuzz.

## Data Availability

This study does not use or generate any external datasets. All results are derived from the methodology and experimental framework described in the paper. No additional data is required to reproduce the findings beyond the information provided in the manuscript.

## Acknowledgments

Zhou and Xuesong were supported in part by NSF under Grant CNS-2047476. Hong and Hengkai were supported in part by ONR under Grant No. N00014-26-1-2218 and NSF under Grants CNS-2339848 and CNS-2247652. Fenglu's work was done when visiting UCI. We also thank Jacob Lee, David Ning, and Cara Failer for their contributions, who were supported under NSF CNS-2047476 REU Supplements. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the sponsors.

## References

[1] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[2] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[3] Nils Bars, Moritz Schloegel, Nico Schiller, Lukas Bernhard, and Thorsten Holz. No peer, no cry: Network application fuzzing via fault injection. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 750–764, 2024.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[5] Ronan David. The overwhelming threat of DNS attacks on the finance industry. https://www.globalbankingandfinance.com/the-overwhelming-threat-of-dns-attacks-on-the-finance-industry/, 2023.

[6] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

[7] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147*, 2024.

[8] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2523–2540, 2020.

[9] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. Chatgpt for vulnerability detection, classification, and repair: How far are we? In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 632–636. IEEE, 2023.

[10] Google. Honggfuzz. https://google.github.io/honggfuzz/.

[11] Google. OSS-Fuzz - Continuous Fuzzing For Open Source Software. https://github.com/google/oss-fuzz.

[12] Google. Gemini 1.5 Pro. https://deepmind.google/technologies/gemini/pro/, 2024.

[13] Internet Systems Consortium. Bind 9 Configuration Reference. https://bind9.readthedocs.io/en/stable/reference.html, 2023.

[14] ISC. Release Notes for BIND Version 9.12.0. https://downloads.isc.org/isc/bind9/9.12.0/RELEASE-NOTES-bind-9.12.0.pdf, 2021.

[15] ISC. BIND . https://www.isc.org/bind/, 2024.

[16] Bahruz Jabiyev, Anthony Gavazzi, Kaan Onarlioglu, and Engin Kirda. Gudifu: Guided differential fuzzing for http request parsing discrepancies. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 235–247, 2024.

[17] Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu, and Engin Kirda. {FRAMESHIFTER}: Security implications of {HTTP/2-to-HTTP/1} conversion anomalies. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1061–1075, 2022.

[18] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin 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.
[19] Sean Michael Kerner. BIND DNS Holds Lead. https://www.serverwatch.com/server-news/bind-dns-holds-lead/, 2020.
[20] Jiwon Kim, Benjamin E Ujcich, and Dave Jing Tian. Intender: Fuzzing {Intent-Based} networking with {Intent-State} transition guidance. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4463–4480, 2023.
[21] David C Lawrence, Warren Kumari, and Puneet Sood. Serving Stale Data to Improve DNS Resiliency. RFC 8767, March 2020.
[22] Triet Huynh Minh Le, Xiaoning Du, and M Ali Babar. Are latent vulnerabilities hidden gems for software vulnerability prediction? an empirical study. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 716–727. IEEE, 2024.
[23] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. Power: Program option-aware fuzzer for high bug detection ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 220–231. IEEE, 2022.
[24] Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848*, 2024.
[25] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. Ecfuzz: Effective configuration fuzzing for large-scale systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.
[26] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. Pre-training by predicting program dependencies for vulnerability analysis tasks. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
[27] LLVM. LibFuzzer - A Library for Coverage-guided Fuzz Testing. http://llvm.org/docs/LibFuzzer.html.
[28] LLVM. Undefined Behavior Sanitizer (UBSan). https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2023.
[29] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4481–4498, 2023.
[30] Jill McKeon. DNS NXDOMAIN Flood DDoS Attacks Impacting Healthcare, HC3 Warns. https://healthitsecurity.com/news/dns-nxdomain-flood-ddos-attacks-impacting-healthcare-hc3-warns, 2023.
[31] Ruijie Meng, Gregory J Duck, and Abhik Roychoudhury. Program environment fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 720–734, 2024.
[32] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.
[33] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, December 1990.
[34] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
[35] Roberto Natella and Van-Thuan Pham. Profuzzbench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, pages 662–665, 2021.
[36] Oracle. Commands of BIND 9 rndc.
[37] Tao Peng, Shixu Chen, Fei Zhu, Junwei Tang, Junping Liu, and Xinrong Hu. Ptlvd: Program slicing and transformer-based line-level vulnerability detection system. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 162–173. IEEE, 2023.
[38] Van-Thuan Pham, Marcel Böhme, and Abhik 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.
[39] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Trans. Softw. Eng. Methodol.*, 32(6), September 2023.
[40] Gaganjeet Singh Reen and Christian Rossow. Dpifuzz: a differential fuzzing framework to detect dpi elusion strategies for quic. In *Annual Computer Security Applications Conference*, pages 332–344, 2020.
[41] Scapy. Scapy. https://scapy.net/, 2024.
[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, 2012.
[43] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.
[44] Kostya Serebryany. Sanitize, Fuzz, and Harden Your C++ Code. San Francisco, CA, 2016. USENIX Association.
[45] Shiwen Shan, Yintong Huo, Yuxin Su, Yichen Li, Dan Li, and Zibin Zheng. Face it yourselves: An llm-based two-stage strategy to localize configuration errors via logs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 13–25, 2024.

[46] Juraj 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.
[47] Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman Nath. Hotgpt: How to make software documentation more useful with a large language model? In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 87–93, 2023.
[48] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020.
[49] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. Campion: debugging router configuration differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 748–761, 2021.
[50] Andreas Walz and Axel 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.
[51] Dawei Wang, Ying Li, Zhiyu Zhang, and Kai Chen. {CarpetFuzz}: Automatic program option constraint extraction from documentation for fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1919–1936, 2023.
[52] Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
[53] Kelin Wang, Mengda Chen, Liang He, Purui Su, Yan Cai, Jiongyi Chen, Bin Zhang, Chao Feng, and Chaojing Tang. Osmart: Whitebox program option fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 705–719, 2024.
[54] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
[55] Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–357. IEEE, 2023.
[56] Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. Netllm: Adapting large language models for networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 661–678, 2024.
[57] Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024.
[58] Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang. Does data sampling improve deep learning-based vulnerability detection? yeas! and nays! In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2287–2298. IEEE, 2023.
[59] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl.
[60] Michal Zalewski. Fuzzing Random Programs Without Execve(). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2019.
[61] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.
[62] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering*, 49(8):4196–4212, 2023.
[63] Qifan Zhang, Xuesong Bai, Xiang Li, Haixin Duan, Qi Li, and Zhou Li. ResolverFuzz: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing. In *Proceedings of the 33rd USENIX Security Symposium (USENIX 2024)*, Philadelphia, PA, August 2024.
[64] Yichi Zhang. Detecting code comment inconsistencies using llm and program analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 683–685, 2024.
[65] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. Fuzzing configurations of program options. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–21, 2023.
[66] Xin Zhou, Duc-Manh Tran, Thanh Le-Cong, Ting Zhang, Ivana Clairine Irsan, Joshua Sumarlin, Bach Le, and David Lo. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. *arXiv preprint arXiv:2407.16235*, 2024.
[67] Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 47–51, 2024.

Received 2026-01-30; accepted 2026-06-25