# CSFuzzer: A Grey-Box Fuzzer for Network Protocol using Context-aware State Feedback

Xiangpu Song[a], Yingpei Zeng[b,*], Jianliang Wu[c], Hao Li[a], Chaoshun Zuo[d], Qingchuan Zhao[e] and Shanqing Guo[a,*]

[a]School of Cyber Science and Technology, Shangdong University, Qingdao 266237, China
[b]School of Cyberspace, Hangzhou Dianzi University, Hangzhou 310000, China
[c]School of Computing Science, Simon Fraser University, Canada
[d]Department of Computer Science and Engineering, The Ohio State University, America
[e]Department of Computer Science, City University of Hong Kong, China

ARTICLE INFO

*Keywords*:
Protocol fuzzing
State feedback
Software testing

ABSTRACT

Code coverage-guided fuzzers have achieved great success in discovering vulnerabilities, but since code coverage does not adequately describe protocol states, they are not effective enough for protocol fuzzing. Although there has been some work introducing state feedback to guide state exploration in protocol fuzzing, they ignore the complexity of protocol state space, e.g., state variables have different categories and are diverse in data type and number, facing the challenges of inaccurate state variable identification and low fuzzing efficiency.

In this paper, we propose a novel context-aware state-guided fuzzing approach, CSFuzzer, to address the above challenges. CSFuzzer first divides the state variables into two categories, i.e., protocol-state variables and sub-state variables based on the context of the states, and automatically identifies and distinguishes these two categories of state variables from code. Then, CSFuzzer uses a new state coverage metric named *context-aware state transition coverage* to more efficiently guide fuzzing. We have implemented a prototype of CSFuzzer and evaluated it on 12 open-source protocol programs. Our experiments show that CSFuzzer outperforms the existing state-of-the-art fuzzers in terms of code and state coverage as well as fuzzing efficiency. CSFuzzer successfully discovered 10 zero-day vulnerabilities, which have been confirmed by the stakeholders and assigned 9 CVEs/CNVDs.

## 1. Introduction

Fuzzing is one of the most popular ways of finding software vulnerabilities and has been extensively studied [2, 29, 21, 38, 58]. Code-Coverage-Guided (CCG) fuzzing, like AFL [2], is one of the most popular strategies in these studies. Despite the great success, it is inefficient for fuzzing protocol programs because code coverage does not adequately represent the protocol state space [20, 27, 21].

Notably, recent research introduced another strategy, e.g., State-Coverage-Guided (SCG) fuzzing, to facilitate state exploration and made great progress [26]. State-guided fuzzers not only preserve the fuzzing inputs that cover new control-flow edges as seeds but also retain the inputs that cover new states. To guide fuzzing based on state coverage, fuzzers need to identify the variables that indicate the states of a protocol, i.e., state variables. Existing research identifies the state variables using the response code field in the packet [41] or enum-type variables in the source code [21] or even annotate the variables manually [20]. Since the state variables are diverse in the data type and number, the performance of existing automatic state variable identification approaches [27, 21, 57] is reduced by the inaccuracy of variable identification. In addition, existing state-guided fuzzers either use state path coverage [21] or state transition coverage [41, 57, 42] to guide fuzzing and thus are faced with the challenges of seed explosion and low feedback sensitivity.

To address these challenges, we design CSFuzzer by proposing a novel approach to identify protocol state variables and a new context-aware state coverage metric to guide fuzzing. We first analyze 40 open-source protocol programs containing more than 20 protocols to pinpoint three characteristics of state variables. These characteristics include: (1) variables could affect program control flow upon state changes [21, 57] (i.e., control flow effects); (2) the semantics of variable names are related to the protocol attributes described in RFC documents for code readability reasons [24, 44] (i.e., semantics feature), such as using `msg_type` to indicate the message type; and (3) variables are used in code statements with high code complexity and call functions with the same prefix or suffix names (i.e., code feature). We first identify state variables based on these three characteristics and divide the variables into two types, namely *protocol-state variables* and *sub-state variables* based on the execution context of the program. Then, to better utilize the identified state variables and balance fuzzing practicality and efficiency, we design a new state coverage metric named *context-aware state transition coverage* (*CAST-Coverage*) to guide fuzzing. Specifically, we use state transition coverage to track the two categories of state variables separately and consider the protocol state as the

*Corresponding author
songxiangpu@mail.sdu.edu.cn (X. Song); yzeng@hdu.edu.cn (Y. Zeng); wujl@sfu.ca (J. Wu); 202020883@mail.sdu.edu.cn (H. Li); zuo.118@osu.edu (C. Zuo); qizhao@cityu.edu.hk (Q. Zhao); guoshanqing@sdu.edu.cn (S. Guo)

execution context of its corresponding sub-states. Lastly, we introduce a rarity-preferred seed energy scheduling strategy to facilitate state exploration.

We have implemented CSFuzzer based on AFL [2] and evaluated it on 12 widely-used protocol programs in terms of accuracy of state variable identification, branch and state coverage, efficiency, and effectiveness. We compare CSFuzzer with eight state-of-the-art fuzzers, i.e., AFL [2], AFL++ [29], AFLNET [41], StateAFL [38], IJON [20], SGFUZZ [21], ChatAFL [36], and NSFuzz [42]. Our experiments indicate that CSFuzzer outperforms these fuzzers by identifying 177.8% more state variables than SGFUZZ, achieving 38.4% to 809.1% more state coverage and 3.3% to 40.2% more branch coverage than other fuzzers in the 24-hour campaign. CSFuzzer exhibits superior performance in terms of effectiveness and efficiency in detecting vulnerabilities, discovering 21 vulnerabilities—more than any other fuzzer in our experiments. It also achieves a speedup of 3.4x to 44.6x in vulnerability discovery compared to the baselines. Moreover, CSFuzzer discovered 10 new vulnerabilities in real-world protocol programs with 9 CVEs/CNVDs assigned. We provide a link to our code repository for review purposes, i.e., https://anonymous.4open.science/r/csfuzzer-8646/.

In summary, this paper makes the following contributions:

- We propose a novel state variable identification approach based on three characteristics of the state variable, i.e., control flow effects, semantics feature, and code feature.
- We propose a new coverage metric, i.e., context-aware state transition coverage to keep track of two categories of state variables and facilitate state space exploration based on a rarity-preferred energy scheduling strategy.
- We implement a prototype of CSFuzzer and evaluate it on 12 protocol programs against eight state-of-the-art fuzzers. Our evaluation shows that CSFuzzer significantly outperforms other fuzzers. Additionally, CSFuzzer discovered 10 new vulnerabilities with 9 CVEs/CNVDs assigned.

## 2. Motivation

In this section, we use a use-after-free vulnerability *CVE-2019-15232* that occurs in Live555, a popular RTSP [13] (Real Time Streaming Protocol) protocol implementation, to present the motivation and intuition behind our approach.

Listing 1 shows the simplified code that contains the vulnerability. This vulnerability would be triggered if the server receives two messages requesting the same WebM [8] stream file with identical track IDs, e.g., input Sequence 3 in Figure 1 ($\langle$*SETUP*, webm, track1$\rangle \rightarrow \langle$*SETUP*, webm, track1$\rangle$). Specifically, after receiving the first message, the server invokes the parsing function in line 3. Then, it calls the function in line 14 to retrieve the relevant parameters for the current stream. After that, a new demux `trackSource` object

```
1  void handleRequestBytes(...) {
2    if (strcmp(cmdName, "SETUP") == 0) {  // M1 in Fig.1
3      clientSession->handleCmd_SETUP(...);
4      playAfterSetup = clientSession->fStreamAfterSETUP;
5    } else if (strcmp(cmdName, "PLAY") == 0)
6      clientSession->handleCmd_PLAY(...);
7    else if (strcmp(cmdName, "TEARDOWN") == 0)
8      clientSession->handleCmd_TEARDOWN(...);
9    if (playAfterSetup && strcmp(streamName, url) == 0) {...}
10   if (clientSession != NULL) { handleCmd_withinSession(...); }}
11 void RTSPClientSession::handleCmd_SETUP(...) {
12   if (trackId != NULL)
13     subsession->deleteStream(SessionId, trackId); // UAF free (S1
         and S2 in Fig.1 )
14   subsession->getStreamParameters(...); } // S3 and S4 in Fig.1
15 // called in getStreamParameters() only if stream is webm
16 FramedSource* MatroDemux::newDemuxedTrackByTrackNum(...) {
17   trackSource = new MatroDemuxedTrack(envir(), ...);  } // UAF new
         (S4 in Fig.1)
18 void handleCmd_withinSession(...) {
19   if (!strcmp(streamName, urlPreSuffix))
20     if (!strcmp(trackId, urlSuffix)) {...} }
```

Listing 1: Simplified code in RTSP protocol implementation.

is created in line 17. When receiving the second message, the server processes the message in the same way as in the first step. However, since the track ID already exists, the server proceeds to free the stream object (line 13) and then attempts to create a demux object (line 17). Lastly, the vulnerability is triggered because the object returned by `envir()` has been freed in line 13.

Figure 1 illustrates three possible message sequences involving three state variables: `cmdName`, `trackId`, and `streamName` as shown in Listing 1, including state transitions corresponding to each message. The nodes *M1* and *S1* − *S4* are different values of these three variables, indicating different states. The `cmdName` denotes the message type and directly affects the protocol state transition, and its value `SETUP` is represented by Node *M1*. The `trackId` is a finer-grained state variable describing the track identifier of the requested file and its value, like `track1` and `track2`, corresponds to nodes *S1* and *S2*, representing audio and video streams, respectively. Similarly, `streamName` is also a finer-grained state variable, indicating the requested streaming file. Its value `mpeg` and `webm` showed by nodes *S3* and *S4*, representing MPEG [10] and WebM type files, respectively. The sequence labeled `Sequence 3` depicts the state transition associated with the vulnerability scenario.

Existing state-guided fuzzing faces the following two limitations in discovering such vulnerability.

**Existing state-identification approaches are coarse-grained**. Some approaches infer protocol state by identifying certain fields at fixed offsets in packets [17, 41, 32, 42, 40], such as response code or message type. However, the response code alone cannot capture fine-grained internal state transitions, because the response code for different messages, which trigger different internal state transitions, could be the same, e.g., RTSP messages [40]. The fields that describe the message type (e.g., `cmdName` in Listing 1) itself are also not accurate enough. For example, if we only infer the state according to `cmdName`, other finer-grained state transitions, such as *S1* − *S4*, will not be captured, making it difficult

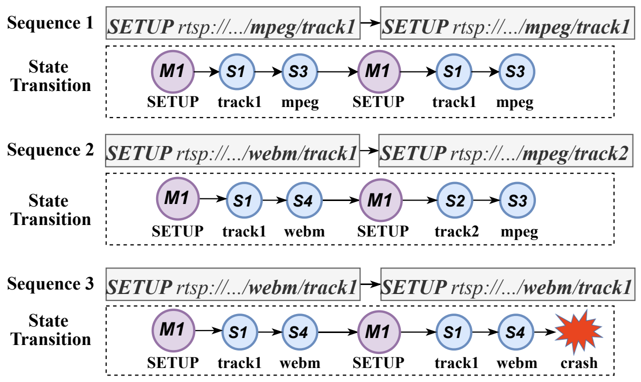

**Figure 1:** State transition consisting of three key state variables in motivation example.

to explore this finer-grained state space. Notably, in many protocols, state variables include not only those that are explicitly represented as parameters in messages but also internal variables maintained by the program, which may not appear in message fields. Therefore, some recent work improves state variable identification [20, 21, 42], which is more precise than the former approaches, to infer states. However, these approaches can only identify state variables of a limited number of types. For example, SGFUZZ cannot identify these three state variables in Listing 1 because all of them are string types, which are not supported. NSFuzz only focuses on variables that represent the state of the network service and therefore cannot identify them. Thus, existing fuzzers cannot observe the state transitions in Figure 1. As such, a finer-grained state variable identification is beneficial for state fuzzers to explore more state space yet unavailable.
**Existing state-tracking approaches ignore dependencies between state variables, resulting in low efficiency.** Even assuming that existing approaches can identify all the states in Figure 1, they ignore the fact that fine-grained state values like *S1*−*S4* are subordinate to the state value like *M1*. Specifically, *state path coverage* [21] (*SP-Coverage*) tracks the whole path of state transition and has the highest feedback sensitivity [52]. Since the number of interactions with a protocol implementation can be arbitrarily large, the sequence of visited states tracked by SP-Coverage can potentially be infinitely long. Therefore, it is prone to seed explosion when there are many states executed in the program, preserving too many seeds that have little variability and are unlikely to find new errors [49, 27]. *State transition coverage* [57] (*ST-Coverage*) can mitigate seed explosion effectively, similar to the edge coverage used by AFL [2], but it may reduce the feedback sensitivity [49]. Specifically, ST-Coverage tracks transitions from the previous state to the current state, thus it can only preserve the context between a state and its neighboring states while losing the context between a state and its subordinate states. For example, it cannot distinguish between the same state transitions consisting of `streamName` and `trackId` in the function `handleCmd_withinSession` in line 18 when they are accessed in different protocol messages, such as `PLAY` and `TEARDOWN`, since these two variables are not adjacent to the state variable `cmdName`. This scenario is particularly common in protocols where there may be multiple identical fields in different kinds of packets.

## 3. Challenge and Solution

To design a state guidance approach that can effectively improve feedback sensitivity and fuzzing efficiency, there are several challenges need to be addressed.

To construct a fine-grained state guidance approach, an intuitive approach is to identify more state variables in protocol implementations, such as *trackId* and *streamName* in Section 2, but this leads to the first challenge: **how to identify more different categories of protocol-relevant state variables?** Theoretically, we could consider all variables affecting the control flow as state variables, but this usually results in a decrease in fuzzer performance due to state explosion. Moreover, unlike existing approaches [20, 21], which only consider specific types of variables or manual annotation, the ideal approach must handle a wide variety and large quantity of protocol-relevant state variables automatically.
**Solution:** To address this challenge, we first analyzed 40 protocol implementations and summarized the common coding features found in these implementations. Subsequently, based on the initial study, we first extract all the code statements that used state variables as the scope of identification and then identify different state variables through code features and variable names.

After identifying state variables, we need to track the values of state variables to guide the fuzzer for state space exploration. Existing state feedback approaches such as SP-Coverage and ST-Coverage either cause server seed explosion or loss of fine-grained state transition details as shown in Section 2. Therefore, this leads to the second challenge: **how to track state variables in an effective way that could improve the feedback sensitivity while maintaining the fuzzing performance?** An ideal state feedback should be able to balance the feedback sensitivity and fuzzing performance, and it should distinguish between the same state transitions consisting of *steamName* and *trackId* in line 18 of Listing 1 when they are accessed in different protocol messages.
**Solution:** To address this challenge, we design a novel state feedback approach to track state variables and guide the fuzzer. Specifically, we first classify identified state variables into two categories based on code feature analysis and the degree of their influence on the control flow. Then, we track their state values in different ways to calculate the final state coverage. Lastly, we design a rarity-preferred energy scheduling strategy to guide the fuzzer's exploration of the protocol state space.

## 4. CSFuzzer Design

In this section, we first present a general overview of the design of CSFuzzer, and then introduce the design details

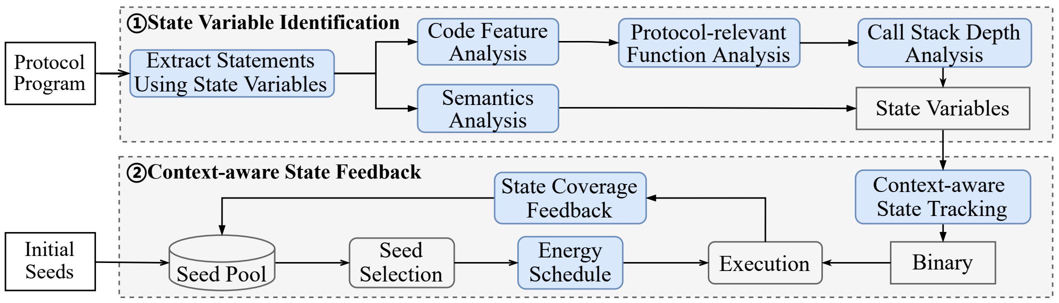
①State Variable Identification
Protocol Program
Extract Statements Using State Variables
Code Feature Analysis
Semantics Analysis
Protocol-relevant Function Analysis
Call Stack Depth Analysis
State Variables
②Context-aware State Feedback
State Coverage Feedback
Context-aware State Tracking
Initial Seeds
Seed Pool
Seed Selection
Energy Schedule
Execution
Binary

**Figure 2:** Overview of the CSFuzzer.

of each functional module individually according to the workflow.

## 4.1. Overview

Figure 2 shows an overview workflow of CSFuzzer, divided into two main phases in the order of execution: state variable identification, and context-aware state feedback.

**State Variable Identification.** A protocol can be modeled as a state machine described by a directed graph, $G = (V, E)$ [43]. The set of vertices $V$ in $G$ represents a set of protocol states specified by RFC documents [39] while the set of edges $E$ represents the operation that triggers the transitions between protocol states in $V$. Programs commonly use variables to describe state [21]. Thus, we follow such concepts of the state machine and classify state-related variables into two categories: *protocol-state variables* (abbreviated as *pstate-variables*) and *sub-state variables* (abbreviated as *sstate-variables*). The ***pstate-variables*** are defined as variables representing the protocol state ($V$) or the variables corresponding to the operations (in $E$) triggering the protocol state transition, such as `cmdName` in Listing 1. The ***sstate-variables***, on the other hand, are those non-pstate-variables that describe other message fields as well as protocol-related program operations, and whose values can lead to the execution of different control flows (i.e., execution branches) while processing messages, such as `trackId` and `streamName`.

CSFuzzer automatically identifies both pstate-variables and sstate-variables based on features obtained from empirical analysis of protocol implementations. CSFuzzer first extracts the statements that directly affect the control flow as the scope of state variable identification. Then, it identifies state variables used for dispatching protocol behaviors by analyzing their distinctive code features. Subsequently, CSFuzzer analyzes whether the function in which the variable is located is related to message processing through function backtracking, and the depth of the call stack to distinguish pstate-variables from sstate-variables. Lastly, CSFuzzer uses semantic analysis to further identify those sstate-variables that are not characterized by the distinctive code feature in statements.

**Context-aware State Feedback.** To track the pstate-variables and sstate-variables and provide feedback that is aware of the protocol context, CSFuzzer first instruments the program to obtain state values depending on the categories of identified state variables. Then, it generates state coverage of pstate-variables and sstate-variables based on ST-Coverage and uses the current pstate-variable as the context in the coverage calculation of sstate-variables (i.e., CAST-Coverage). Lastly, CSFuzzer puts the input that finds new state coverage into the seed pool at the end of each fuzzing and uses a rarity-preferred energy scheduling strategy to allocate more energy to seeds that cover less-explored states.

## 4.2. State Variable Identification

### 4.2.1. Overall Algorithm

The identification process is shown in Algorithm 1, which consists of three steps: I) extraction of potential state variables from branch statements and indirect call statements that directly affect control flows (Section 4.2.2); II) identification of candidate pstate-variables and sstate-variables by analyzing code features and function locations of these statements, as well as the semantics of the variable names (Section 4.2.3, 4.2.4, 4.2.6); III) validation of pstate-variables and sstate-variables through analyzing the call stack depth of the statements of each variable in candidate pstate-variables (Section 4.2.5).

**Algorithm 1** The Process of State Variable Identification

```
Input: Module (the Module of LLVM IR)
Output: PStateVars and SStateVars
    // Step1: Extract potential state variables from branch and indirect call statements
 1: for function ∈ Module do
 2:     BranchStmts, InDirectStmts ← GETSTATEMENT(function)        ▷ Section 4.2.2
 3:     for statement ∈ (BranchStmts,InDirectStmts) do
 4:         Var ← GETSTATEVAR(statement)
    // Step2: Identify candidate pstate-variables and sstate-variables
 5:         if CHECKCODEFEATURE(statement) then                    ▷ Section 4.2.3
 6:             if ISPROTCOLRELEVANT(function) then                ▷ Section 4.2.4
 7:                 CandidatePStateVars.ADD(Var)
 8:             end if
 9:         else if ISSUBSEMANTICS(Var) then                       ▷ Section 4.2.6
10:             SStateVars.ADD(Var)
11:         end if
12:     end for
13: end for
    // Step3: Validate state variables in candidate pstate-variables
14: StackDepths ← GETFUNCSTACKDEPTH(CandidatePStateVars)  ▷ Section 4.2.5
15: for Var ∈ CandidatePStateVars do
16:     if ISMINDEPTH(Var,StackDepths) then
17:         PStateVars.ADD(Var)
18:     else
19:         SStateVars.ADD(Var)
20:     end if
21: end for
```

### 4.2.2. *Extract Statements that Use State Variables*

It is impractical to identify state variables at all code locations since too many states would slow fuzzing performance [27]. On the other hand, tracking states in explicit assignment statements [21] may cause seed explosion, e.g., a 32-bit integer variable representing a field in the packet, which has $2^{32}$ different values. Therefore, it is crucial to identify state variables at appropriate locations.

One characteristic of state variables is that they affect changes in the control flow when the state changes [21, 57]. The developers often use branch statements, such as *if* or *switch* statements [21], and indirect function call statements to implement such code logic. Furthermore, these statements already limit the number of valid values of state variables by the number of branches, such as the *case* statements in *switch* statements. Thus, we extract these statements as the scope for subsequent identification of the state variables, and consider variables in the scope that affect the execution branches of these statements as the potential state variables.

We complete the extraction based on the LLVM IR since IR allows us to avoid issues with uninitialized values [27]. For *switch* statements, we extract each `SwitchInst` instruction that directly represents the *switch* statement in the function. For indirect call statements, the program usually stores functions in a global array and then uses the array to call the corresponding function, as shown in Listing 2. Therefore, based on this observation, we extract function call instructions (e.g., `call`) whose call target could point to a global array. For *if* statements, LLVM does not have instructions that represent them, but instead of using a combination of `icmp` and `br` instructions, while the name of the successors begins with `if.`. For example, LLVM IR uses `if.then` and `if.else` or `if.end` to name the successors when the condition is true and false, respectively. Thus we examine each combination of `icmp` and `br` instructions and treat those combinations whose successor names contain `if.` as if statements and save them.

### 4.2.3. *Analyze the Code Features of State Variable*

To identify pstate-variables and sstate-variables, we first studied 40 protocol implementations shown in Table 1 to understand the characteristics of these two categories of state variables, and then automatically identified them based on these characteristics.

We first used CodeQL [4] to identify the entry point where the program receives a message. Then, we analyzed the program for variable values, function names, and code comments to identify key elements such as variables describing the message type and functions associated with different message processing. We scrutinized these elements and determined whether they had names and descriptions consistent with the state or message types outlined in RFC documents and program documentation. Next, we traced the direction of the message flow to locate the dispatch code fragments where the program calls the corresponding processing codes based on these variables.

| Protocol | Subject | Protocol | Subject |
|---|---|---|---|
| **FTP** | BFTPD, LightFTP, ProFTPD, uFTP, Siim/ftp | **DNS** | BIND 9 |
| **RTSP** | Live555, RtspServer, media-server | **COAP** | libcoap |
| **SSH** | OpenSSH, libssh2 | **SIP** | Kamailio |
| **SMTP** | Exim, OpenSMTPD | **DAAP** | OwnTone |
| **IMAP** | cyrus-imapd | **MQTT** | Mosquitto |
| **NNTP** | WendzelNNTP, cyrus-imapd | **DTLS** | TinyDTLS |
| **HTTP** | H2O, httpd, nghttp2, nghttp3 | **LWM2M** | Wakaama |
| **DHCP** | dhcpserver, ddhcpd, udhcp, OpenHarmony/dhcp | **IPP** | ippsample |
| **BGP** | Quagga, BIRD | **Customized** | Curl |
| **TLS/SSL** | OpenSSL, BoringSSL, wolfSSL, botan, MatrixSSL | **VNC** | libvncserver |

**Table 1**
Protocol implementations for state-variable analysis.

Overall, we found that the names of most state variables are semantically characterized [57, 44] during our analysis, i.e., developers often use keywords describing protocol properties in RFC documents and their variants to name state variables, which can also be used to identify them (c.f., Section 4.2.6). Then, we found that 35 programs use branch statements and 5 programs use indirect call statements to implement the dispatch processing logic based on pstate-variables (i.e., state machine). This result can also demonstrate that *switch* and *if* statements are the most common way to implement state machines [45] even in protocol implementations since it is simple and readable, accounting for 90% of our analysis. Similarly, some sstate-variables are also used to implement the logic for dispatching non-protocol state-level behavior through these statements in different protocol states. More importantly, both pstate-variables and sstate-variables in these statements responsible for dispatching logic have distinctive code features as the following that help us identify them.

The state variables in branch statements for dispatching protocol behavior exhibit three distinctive code features, illustrated in line 2~8 of Listing 1. It shows a code snippet for state variable `cmdName` in *if* statements, which are similar to *switch* statements. (1) The first feature is that the number of *if* statements and *case* statements that use pstate-variables is consistent with the number of protocol states (*threshold 1*), e.g., lines 2, 5, and 7 in Listing 1. (2) Another feature is that most branches (*threshold 2*) that call functions have a common prefix or suffix between their names, such as `handleCmd_`. This practice is commonly employed by developers who encapsulate processing logic for various stages into a set of functions following similar naming conventions. (3) Some developers may opt to incorporate processing code directly into branches rather than utilizing functions, elevating the code complexity in these scenarios. To assess this complexity (*threshold 3*), we employ Cyclomatic complexity [50] as the metric. Consequently, given a branch statement scenario, if it fulfills the (1) + (2) or (1) + (3) criteria, we consider the variable in the conditional statement like `cmdName` as a preliminary result of the candidate state variables for further analysis. In the initial study, the pstate-variables in 37 protocol implementations adhere to the criterion (1) + (2), while in three implementations, they adhere to the other criterion.

```
// global array that stores functions like command_xxx
const struct command commands[] = {
  {"USER", ..., command_user, ...},
  {"PASS", ..., command_pass, ...},
  {"LIST", ..., command_list, ...}, ... };
int parsecmd(char *str){
  for (i=0; commands[i].name; i++){  // i: pstate-variable
    // str matches the array commands[i]
    if (!strncasecmp(str, commands[i].name, ...){
      // invoke the corresponding function
      commands[i].function(str); }}}
```

Listing 2: Simplified code of indirect call statement in BFTPD.

We set three thresholds for the above features to identify these candidate variables, which is explained in Section 6.1.

Further, the state variables in indirect call statements also exhibit similar code features in the global array called by the statements as Listing 2 shows. (1) First, the length of the global array that holds the function is equal to the number of protocol states (*threshold 1*), such as `commands`, where each function corresponds to the processing code for each protocol state. (2) Most function names in the array have the same prefix or suffix string, such as `command_` (*threshold 2*). Thus, when faced with such an indirect call statement that satisfies these two characteristics, we consider the variable used to control the function call as a preliminary result of candidate state variables, such as the variable `i` in Listing 2. We set the same thresholds as threshold 1 and threshold 2 in branch statements.

It is worth mentioning that with the code feature analysis, we can identify not only pstate-variables located in the state machine, but also those sstate-variables that are used for dispatching other important protocol operations, even if they have no semantic features, such as `next4Bytes` in Live555, which is a key program state for parsing streaming media files. We will distinguish between pstate-variables and sstate-variables in the candidate results in subsequent steps.

#### 4.2.4. Analyze if the Function is Protocol-relevant

Although we have identified a few candidate pstate-variables through code feature analysis (Section 4.2.3), some results are concentrated in non-protocol message processing modules, leading to false positives. Therefore, we design an approach to filter candidate pstate-variables to ensure they are protocol-relevant.

We identify candidate pstate-variables by backtracking upward from the function where the variable is located, ensuring that these variables are within the scope of message processing functions. Specifically, CSFuzzer first locates the entry function of processing functions, which often corresponds to the network event loop function because most protocol programs constantly listen for network events within the loop, e.g., 31 of 40 protocol programs in our analysis fall into this category. Thus, we employ dynamic and static analysis to identify such entry functions, similar to NSFuzz [42]. Breakpoints are set on system calls responsible for receiving data (e.g., `read` and `recv`) to capture the runtime call stack. Then, we identify the functions containing I/O loops like `select` and `poll_wait`, and compare them to the call stack by scanning upwards from the bottom of the stack (e.g., `main`) to locate the first identical function that contains the I/O loops, which is designated as the entry function. This is because, during the service processing stage, the network event loop is usually near the bottom of the extracted call stack [42], which helps exclude unrelated program loops. In addition, for protocol programs that do not implement an event loop, we use the `main` function of the program as the entry function.

We then trace back through the call graph to the entry function, commencing from the function where the candidate variable is located. Our goal is to determine whether this candidate variable falls within the scope of the protocol module. Thus, if the trace is successful, we consider the variable as a candidate pstate-variable and proceed with the subsequent analysis.

#### 4.2.5. Analyze the Call Stack Depth of State Variables

The code feature analysis (Section 4.2.3) can also identify sstate-variables that are used to dispatch non-protocol state-level behaviors, such as parsing field-level properties of packets, thus we further use a dynamic analysis approach to classify these state variables.

We identify pstate-variables in candidate results by gauging the level of impact of variables on the control flow, which can be reflected in the depth of the runtime call stack associated with these candidate variables. Specifically, CSFuzzer first set breakpoints at the code locations corresponding to these variables, and then replay packets to extract each runtime call stack, enabling us to calculate the depth of these stacks. We consider all variables that affect the control flow to the greatest extent as the pstate-variables, specifically those residing in functions closest to the bottom of the runtime call stack. In cases where multiple variables share the same depth near the bottom of the stack, they are all treated as pstate-variables. The intuition is that the protocol program will first process pstate-variables and then dispatch corresponding message parsing logic based on its value. For example, for two variables close to the bottom and top of the stack respectively, the latter's influence on the program is usually limited to the interior of one or more functions. In contrast, the variable close to the bottom of the stack can continue to influence the rest of the program after the latter variable has finished its life cycle.

#### 4.2.6. Analyze the Semantics of Sub-State Variable Names

While we have identified some sstate-variables with particular code features, there are quite a few unidentified sstate-variables that may not have such features. As previously analyzed, the semantics of variable names related to the protocol is an important feature of state variables, so we design a semantic analysis approach to identify remaining sstate-variables that do not show distinctive code features. Note that we do not identify pstate-variables by this method, as it would be easy to identify sstate-variables as pstate-variables and cause seed explosion.

We summarised high-frequency keywords describing protocol state and attributes from Curl [5] and its corresponding RFC documents since it is representative and implements multiple protocols, and then we extended the keyword set by adding near-synonyms and acronyms, to generate a state variable dictionary with 35 keywords for analyzing the semantics of state variable names. We consider storing high-frequency and generalized state keywords rather than all attributes of each protocol, avoiding an overload of domain-specific knowledge. The complete content of the dictionary is as follows, which is divided into three categories. (1) Data Units: operation, message (msg), pdu, request (req), response, stream, association (assoc), transaction, packet. (2) Protocol State: state, status, code, results, outcome. (3) Protocol Properties: version, attribute (attr), mode, flag, qos, type, kind, method, approach, category, command (cmd), option, property, track, extension, role. The content in brackets indicates the abbreviation of the keywords.

We design two methods to analyze the semantics of the variable: the sub-string matching algorithm and the cosine similarity algorithm. First, we use the sub-string matching algorithm and the dictionary to identify sstate-variables, and if the keyword in the dictionary is part of the variable name, we consider the variable to be a sstate-variable. Second, to improve the flexibility of identification, we use the cosine similarity algorithm based on the *word2vec* [37] to identify synonymous names and consider the variable to be a sstate-variable if the cosine of the angle between the two words exceeds 0.5. To train the word2vec model, we constructed a text corpus containing textual descriptions of the protocol attributes from 30 RFC documents. This corpus served as the basis for training the CBOW-type model. The training process used the default parameters as those provided by Google [7].

## 4.3. Context-aware State Feedback

In this section, we describe how CSFuzzer constructs context-aware state feedback based on identified state variables to facilitate state exploration.

### *4.3.1. Context-aware State Tracking (CAST-Coverage)*

After identifying the state variables, another key challenge is how to track the state values while balancing feedback sensitivity and fuzzing efficiency.

We first consider ST-Coverage to track states rather than SP-Coverage because the latter is prone to seed explosion [49] and the efficiency is crucial for protocol fuzzing [19, 57, 34]. As mentioned before, ST-Coverage loses most of the context between states, reducing the sensitivity and accuracy of feedback. To be efficient and accurate, we propose CAST-Coverage to track these two categories of state variables. CAST-Coverage is inspired by *state-based context coverage* [1], which is a coverage metric to reflect the scenarios where the same class behavior performs in different class states in object-oriented software. In the scenario of protocol fuzzing, a sub-state is like a class behavior, while a state is similar to a class state in state-based context coverage. Accordingly, CAST-Coverage can reflect different cases where the protocol is in the same sub-state but different protocol states.

$$\text{ST-Coverage} = \sum_i B\left(f\left(S_i, S_{i+1}\right)\right) \tag{1}$$

$$\text{CAST-Coverage} = \sum_i B\left(f\left(P_i, P_{i+1}\right)\right) + \sum_i B\left(f\left(Q_i, Q_{i+1}\right) \oplus P_{\text{last}}\right) \tag{2}$$

To calculate the CAST-Coverage, CSFuzzer keeps track of the pstate-variables and sstate-variables in different ways. We first introduce the formal calculation of ST-Coverage, as shown in Equation (1), where $S$ denotes the union of all possible values of pstate-variables and sstate-variables, and $S_i$ and $S_{i+1}$ indicate the two different state values that are adjacent to each other during program execution, respectively. ST-Coverage calculates the adjacent state transition values of $S_i$ and $S_{i+1}$ using the transition calculation function $f$ and save them as indexes in the bitmap $B$. Thus, ST-Coverage is represented by the state bitmap. Equation (2) shows the formal process of CAST-Coverage, where $P$ and $Q$ are the state values of pstate-variables and sstate-variables respectively. CAST-Coverage first uses ST-Coverage to capture protocol state transitions constructed by pstate-variables. Subsequently, to capture fine-grained state transitions within different protocol states, we maintain a global variable $P_{last}$ to record the most recent pstate-variable executed as the execution context of subsequent sstate-variables, and then it will participate in the calculation of state transitions of sstate-variables through the exclusive or operation $\oplus$. In CAST-Coverage, we use the function *ijon_hashint* in IJON [20] to implement the transition calculation function $f$. With this approach, CSFuzzer will treat *S4* → *S1* in `Sequence 3` in Figure 1 as a distinct state after receiving `Sequence 1` and `Sequence 2` and preserve such input like `Sequence 3` to the corpus, whereas ST-Coverage loses perception of state transitions in `Sequence 3`.

### *4.3.2. State Exploration Strategy*

CSFuzzer applies two approaches to utilize state coverage to facilitate state exploration.

**State Coverage Feedback.** CSFuzzer collects state coverage at each fuzzing execution via the shared memory called *state_memory* to enhance the fuzzer's state awareness, where each location of the memory corresponds to a state. Thus, CSFuzzer can save those inputs that cover new state coverage, even if they do not cover the new code coverage, to the seed pool to be used as a new starting point for iteratively exploring the state space.

CSFuzzer uses another shared memory called *historical_state_memory* of the same size as *state_memory* collecting state coverage to record the historical state coverage throughout the fuzzing. In *historical_state_memory*, each position corresponds to a specific state, with the value at that position representing the number of state hits (i.e., the number of times each state is hit by all inputs during fuzzing). We evaluate whether a state is rarely explored by comparing its hit count to the average number of state hits across *historical_state_memory*. If the count falls below this average, the state is considered rare. We will assign different energies to seeds based on state coverage to facilitate the exploration of rare state spaces.

**Energy Schedule.** Given a previously executed seed, we calculate the ratio of rare states in the historical state coverage for that seed as follows.

$$rareRatio = \frac{\sum_{i \in rareStates, stateArray[i]>0} 1}{\sum_{i \in allStates, stateArray[i]>0} 1} \tag{3}$$

where *rareStates* stores the set of rare states, *stateArray* is the array that holds the state coverage for the given seed, and *allStates* stores the set of all explored states during fuzzing.

CSFuzzer implements a rarity-preferred energy scheduling strategy to guide the fuzzer to spend more time exploring less-explored state space since seldomly exercised states harbor more undiscovered adjacent states or code logics [21, 56].

We then decide on the energy to be assigned based on the magnitude of the *rareRatio*, as shown in the following.

$$E_e(s) = \begin{cases} E_{ori}(s), & \text{if } rareRatio \le 50\% \\ 2E_{ori}(s), & \text{if } rareRatio > 50\% \end{cases} \tag{4}$$

where $E_{ori}(s)$ is the original energy assigned to the seed $s$ by the original fuzzer AFL [2].

Finally, to avoid the energy beyond the maximum limit supported by the fuzzer, we constrain the energy $E_e(s)$.

$$E_e(s) = MIN\left(E_e(s), E_{\text{MAX}}\right) \tag{5}$$

where $E_{MAX}$ is the max energy supported by AFL.

## 5. Implementation

CSFuzzer is implemented on AFL [2] and LLVM, and has two major components, namely state variable identification, and context-aware state feedback.

**State Variable Identification**. We implement an LLVM pass to identify state variables and use GLLVM [16] to build the entire IR. We then use the GDB script [6] to set breakpoints automatically to the system call that receives data and then dump the call stacks while receiving messages as auxiliary information to identify the entry function of message processing functions. We also use the GDB script to calculate the depth of the call stack to identify pstate-variables. The identification process is semi-automated, as manual involvement is required for starting protocol programs, and does not require the involvement of expert knowledge.

**Context-aware State Feedback**. We define a series of trace functions for different data types of state variables and use the same LLVM pass to insert the trace functions into IR and then link the IR to a new binary for fuzzing. We then implement the feedback on top of AFL and extend a new bitmap of size $2^{14}$ to collect state coverage to avoid conflicts between code and state coverage.

| Subject | Protocol | Version | Language | Transport Layer |
|---|---|---|---|---|
| PureFTPD | FTP | c21b45f | C | TCP |
| BFTPD | FTP | v5.7 | C | TCP |
| ippsample | IPP | 1ee7bcd | C | TCP |
| CUPS | IPP | 07ec506 | C | TCP |
| OpenSSL | TLS | 12ad22d | C | TCP |
| Live555 | RTSP | ceeb4f4 | C++ | TCP |
| DCMTK | DICOM | 7f8564c | C++ | TCP |
| Exim | SMTP | 38903fb | C | TCP |
| Dnsmasq | DNS | v2.73rc6 | C | UDP |
| Curl | Custom | aab3a7 | C | TCP/UDP |
| MbedTLS | DTLS | e483a7 | C | UDP |
| TinyDTLS | DTLS | 06995d4 | C | UDP |

**Table 2**
Target protocol implementations.

## 6. Evaluation

We evaluate CSFuzzer by answering the following research questions:

- **RQ1.** How many state variables can CSFuzzer identify, and what is the proportion of false positives and false negatives?
- **RQ2.** Can CSFuzzer achieve higher branch coverage than other existing fuzzers?
- **RQ3.** How does the state-space exploration capability of CSFuzzer compared to existing fuzzers?
- **RQ4.** How efficiently does CSFuzzer discover protocol state vulnerabilities, and can it discover new vulnerabilities?
- **RQ5.** How each component affects the results of CSFuzzer?

### 6.1. Experiment Setup

**Target Programs**. We selected 12 widely used open-source protocol implementations for evaluating CSFuzzer, as shown in Table 2, with more than 10 different protocols selected (Curl [5] implements multiple protocols). All these protocol implementations have been fuzzed in the evaluation of existing fuzzing works [41, 21, 28].

**Baselines**. Since our goal is to evaluate the impact of different state feedback strategies on protocol fuzzing, we selected eight state-of-the-art (SOTA) fuzzers to make the comparison, including two CCG fuzzers like AFL [2] and AFL++[29], and five SCG fuzzers, AFLNet [41], IJON [20], StateAFL [38], SGFUZZ [21], ChatAFL [36], and NSFuzz [42].

Apart from AFLNet, StateAFL, ChatAFL, and NSFuzz, other fuzzers need additional support to fuzz protocol programs. For example, for the AFL-based fuzzers, such as AFL, AFL++, IJON, and CSFuzzer, we modify the source code to enable the program to read the input from files or the standard input (*stdin*) instead of using *desock* [12] due to compatibility issue [47], and our modifications do not affect the normal logic of the protocol. Since IJON requires manual annotation of state variables, we use the `ijon_push_state` function to annotate the same state variables as CSFuzzer identified for fairness. We also set all the AFL-based fuzzers to *skip deterministic* for better performance [29]. For SGFUZZ, we use *netdriver* to support protocol fuzzing as suggested.

**Environment**. We ran all experiments on two local machines, each of which has two Intel(R) Xeon(R) Gold 6226R CPUs with 32 logical cores, 256 GB RAM, and an Ubuntu 20.04.1 LTS system. To mitigate randomness, we repeated the fuzzing campaign 10 times.

**Evaluation Metrics**. We conducted all evaluations on the Docker container, using the same building processes to provide the same independent environment, including the same initial seeds and dictionaries. We use the *gcov* tool to measure the branch coverage and count the number of unique crashes by running all seeds on recompiled programs with *AddressSanitizer* and *stack hash* [33].

**Threshold Setting in State Variable Identification**. We set three thresholds in our evaluation for the code feature analysis in Section 4.2.3. First, we set the minimum number of branch statements with the same condition, including the *case* and *if* statements, and the length of the global array to three, i.e., *threshold 1*, since it can reflect the number of protocol states. Since there is a gap between protocol design and implementation, e.g., the RFC [13] for RTSP protocol defines 12 message types, but Live555 only uses seven branches to implement the state machine, we do not identify state variables as described in protocol specifications. Instead, we set the value to three to ensure that CSFuzzer can maximize match different protocol implementations and filter out codes that are not used to dispatch protocol important behaviors. We then set the percentage of branch statements and global arrays that call functions with the same prefix or suffix name to 70% to represent the majority of cases, i.e., *threshold 2*, to avoid the static analysis being too restrictive or too broad. Finally, we set the code complexity detection threshold for branch statements to 30, i.e., *threshold 3*, which is used in the software engineering field to indicate that the current code belongs to a highly complex module [3, 25]. All threshold settings are determined based on the investigation and analysis of the 40 protocol programs in Table 1.

## 6.2. State Variable Identification Effectiveness (RQ1)

To evaluate the effectiveness of state variable identification, we manually analyzed how many of the state variables recognized by CSFuzzer were correct and how many state variables were missed by CSFuzzer, and compared them to SGFuzz.

First, for the state variables that have been identified by CSFuzzer and SGFUZZ, we manually analyzed each state variable and its code comment, comparing them to the descriptions in RFC documents and program documentation, to determine the percentage of false positives (FP) in the results. Second, for state variables that are not identified in programs, we construct two ground truths and then determine

| Subject | SGFUZZ | | | CSFuzzer | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Total Num | FP | RFN | Total Num | PState-Variable | | | SState-Variable | | |
| | | | | | Num | FP | RFN | Num | FP | RFN |
| **PureFTPD** | 6 | 5 | 11 | **12** | 1 | 0 | 1 | 11 | 2 | 1 |
| **BFTPD** | 0 | 0 | 0 | **12** | 1 | 0 | 2 | 11 | 5 | 0 |
| **ippsample** | 11 | 1 | 61 | **72** | 1 | 0 | 8 | 71 | 8 | 7 |
| **CUPS** | 36 | 8 | 118 | **130** | 1 | 0 | 7 | 129 | 11 | 23 |
| **OpenSSL** | 239 | 169 | 237 | **680** | 3 | 0 | 2 | 677 | 404 | 29 |
| **Live555** | 4 | 0 | 122 | **132** | 1 | 0 | 0 | 131 | 6 | 0 |
| **DCMTK** | 66 | 7 | 62 | **102** | 1 | 0 | 5 | 101 | 11 | 30 |
| **Exim** | 30 | 23 | 115 | **201** | 1 | 0 | 0 | 200 | 85 | 6 |
| **Dnsmasq** | 0 | 0 | 72 | **89** | 0 | 0 | 0 | 89 | 17 | 0 |
| **Curl** | 303 | 145 | 227 | **531** | 8 | 0 | 6 | 523 | 239 | 93 |
| **Mbedtls** | 51 | 23 | 53 | **122** | 1 | 0 | 7 | 121 | 55 | 14 |
| **TinyDTLS** | 4 | 0 | 15 | **17** | 1 | 0 | 6 | 16 | 0 | 2 |

**Table 3**
Results of state variable identification for SGFUZZ and CSFuzzer.

| Subject | Total Num | SGFUZZ | | | CSFuzzer | | |
|---|---|---|---|---|---|---|---|
| | | Total | FP | CFN | Total | FP | CFN |
| **PureFTPD** | 24 | 6 | 5 | 23 | 12 | 2 | 14 |
| **TinyDTLS** | 53 | 4 | 0 | 49 | 17 | 0 | 35 |
| **ippsample** | 119 | 11 | 1 | 109 | 72 | 8 | 47 |

**Table 4**
Complete state variable analysis for SGFUZZ and CSFuzzer.

the false negative results of CSFuzzer and SGFUZZ based on RFC documents and other documentation. (1) Relative ground truth. Since thousands of variables are declared in the program, e.g., we found 30k variable-defining statements in OpenSSL through CodeQL [4], it is almost impossible to analyze each of them, thus we first merge the identification results of SGFUZZ and CSFuzzer and then remove the false positive results as a set of valid state variables to analyze the relative false negatives (RFN) in results. (2) Complete ground truth. We chose three programs, PureFTPD, TinyDTLS, and ippsample, for constructing a complete ground truth to analyze the complete false negatives (CFN) because they have a small code base to facilitate manual analysis. To reduce the impact of false positives from human analysis, we had two researchers perform the analysis independently and then confirm and correct the results against each of their analysis results.

Table 3 shows the overall results of the state variables that SGFUZZ and CSFuzzer identified in each program. On average, CSFuzzer identified 175 state variables and found 177.8% more state variables than SGFuzz. Then, there are 40.1% FP and 13.7% RFN results in the state variable identification results of CSFuzzer compared to SGFUZZ, which has 50.8% FP and 67.2% RFN, respectively. In addition, we divided the experimental programs into two categories: one was included in the initial study shown in Table 1 and the other was not, and then assessed whether there was a significant difference between the two FP/RFN ratios based on the Mann Whitney U-test [31]. The p-value result was 0.68, which indicates no significant differences, proving the generality of our identification method.

Table 4 shows the CFN results for SGFUZZ and CSFuzzer in PureFTPD, TinyDTLS, and ippsample where *Total Num* denotes the total number of state variables in programs. Besides, in the complete manual analysis, CSFuzzer identified on average 51.5% of the state variables, which is five times more than SGFUZZ, with 9.8% FP and 49% CFN results, compared to 28.6% FP and 92.3% CFN results for SGFUZZ. Therefore, compared to existing work, CSFuzzer has lower FP and CFN rates for state variable identification. This avoids saving redundant test cases due to protocol-independent state variable changes during fuzzing and improves the efficiency of state space exploration.

With further investigation of FNs, we found two possible causes of FNs in pstate-variable identification: (1) The program may use different state variables to represent the same protocol state. We treat only one of the variables located at the dispatch codes responsible for protocol state as the pstate-variable, and the others as sstate-variables, leading to FNs. However, this does not affect the correct construction of the state context, since such dispatch code is usually the entry for message processing, the pstate-variable at this location can accurately mark the state context of all sstate-variables during subsequent processing. (2) Some program supports two or more protocols, e.g., IPP protocol [14] is based on HTTP, which may cause CSFuzzer to treat the pstate-variable from only one of the protocols as the pstate-variable and others as sstate-variables. Regarding the FNs in sstate-variable identification, we summarized two reasons. (1) Most of the sstate-variables in programs are not directly used in branch statements, so they are excluded by CSFuzzer. (2) The semantics of some sstate-variable names do not match with our dictionary since only high-frequency and important protocol attributes are considered to be identified when constructing the dictionary.

We also investigate the FPs in state variable identification and conclude the following three possible reasons: (1) We use the complete program rather than a single function for evaluation, which caused CSFuzzer to identify variables in code that current fuzzing will never execute into as state variables. (2) Since CSFuzzer does not restrict the identification range of sstate-variables through semantic analysis, it may misidentify variables in other non-protocol modules as state variables, such as the cryptography module in OpenSSL and Curl. (3) The variables in the remaining results are mainly used to store configuration information or logs, and the values of these variables are derived from configurations or options, thus the state transitions hardly change once they are covered. Overall, 40.1% of FP results are tolerable for fuzzing because most (more than 62%) of them are located in code paths that are never reached during fuzzing, and the value of 20% of the variable values do not change frequently, they do not have a large impact on state feedback.

## 6.3. Branch Coverage Results (RQ2)

We evaluate the impact of the state feedback of CSFuzzer on code exploration and analyze the branch coverage since code coverage is also an important metric for quantifying fuzzers' capabilities [33]. To evaluate branch coverage, we first adopted the standard 24-hour fuzzing configuration. Additionally, we set evaluation configuration with the same number of fuzzing iterations (i.e., 1 million) for all fuzzers as mentioned in [46, 19]. This setup creates a significant advantage for slower fuzzers, such as AFLNET and StateAFL, ensuring a fair assessment of the state feedback strategies for each fuzzer.

Table 5 shows all campaigns' average branch coverage results after 24 hours and 1 million fuzzing iterations. Overall, CSFuzzer achieves more branch coverage compared to other fuzzers in most cases. Specifically, at the 24-hour setting, CSFuzzer achieves on average 3.7% more branch coverage than AFL, 3.3% than AFL++, 37% than AFLNet, 5.8% than IJON, 25.3% than StateAFL, 40.2% than SGFUZZ, 13.4% than ChatAFL, and 11.9% than NSFuzz. Furthermore, in the setting of 1 million iterations, CSFuzzer also outperforms others, and could achieve 2.8% more branch coverage than AFL, 2.2% than AFL++, 7.0% than AFLNet, 3.3% than IJON, 6.4% than StateAFL, 18.3% than SGFUZZ, 4.8% than ChatAFL, and 4.0% than NSFuzz. We also use the Mann-Whitney U-test [31] to calculate the p-value, and most companies have a p-value less than 0.05, indicating a significant statistical difference between CSFuzzer and others. Due to the space constraint, we provided it in Appendix A.

We further investigate why CSFuzzer performs better compared to other fuzzers. The first reason is better state guidance capabilities of CSFuzzer. Specifically, CSFuzzer performs better than other SOTA state-guided fuzzers under the same limit of iterations, proving the effectiveness of CAST-Coverage that is constructed with more categories of state variables. It is worth noting that CSFuzzer and IJON track the same state variables but apply different state guidance, which demonstrates that CAST-Coverage can more effectively facilitate code exploration, and therefore perform better in both short-term (1 million iterations) and long-term (24 hours) test cycles. Similarly, AFLNet and NSFuzz utilize only response codes, so the state feedback granularity is coarser than CSFuzzer. Second, CSFuzzer effectively balances feedback sensitivity and fuzzing efficiency by categorizing variables and tracking them differently, whereas IJON and SGFUZZ face the problem of seed explosion. Specifically, IJON tracks four consecutive state values and thus may have seed explosion problems when the number of states is large, leading to unstable performances. SGFUZZ has a more severe seed explosion potential because it tracks state variables on all paths, such as saving 90k and 80k seeds in OpenSSL and Curl, respectively. Furthermore, since the state feedback in CSFuzzer allows for the execution of less-likely reached code that can be executed under certain states, CSFuzzer outperforms AFL and AFL++. Since MbedTLS and TinyDTLS have a smaller code base, CSFuzzer achieves almost identical branch coverage as AFL and AFL++. The *netdriver* used by SGFUZZ may prevent fuzzing from starting properly on some programs. For example, it does not support the *fork* system call, thus SGFUZZ could not track coverage and only preserve the initial seeds in PureFTPD and Exim [21]. On the other hand, it prompts that connections were terminated by peers when testing CUPS, resulting in most input not being received by the program. Lastly, since

| Subject | Branch Coverage for 24H | | | | | | | | | Branch Coverage for 1M | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AFL | AFL++ | AFLNet | IJON | StateAFL | SGFUZZ | ChatAFL | NSFuzz | CSFuzzer | AFL | AFL++ | AFLNet | IJON | StateAFL | SGFUZZ | ChatAFL | NSFuzz | CSFuzzer |
| PureFTPD | 1346 | 1373 | 1121 | 1390 | 989 | 633 | 1138 | 1336 | **1404** | 1280 | 1272 | 1090 | 1176 | 1066 | 633 | 1122 | 1281 | **1298** |
| BFTPD | 452 | 496 | 468 | 488 | 424 | 368 | 467 | 464 | **512** | 453 | 424 | 487 | 476 | 467 | 368 | 477 | 483 | **492** |
| ippsample | 3219 | 3204 | 2894 | 3220 | - | 2914 | 2938 | - | **3273** | 2886 | **2965** | 2786 | 2904 | - | 2941 | 2863 | - | 2951 |
| CUPS | 3878 | 3986 | 3340 | 4032 | - | 3046 | 3736 | - | **4121** | 3606 | 3607 | 3348 | 3596 | - | 3046 | **3877** | - | 3629 |
| OpenSSL | 12311 | 12054 | 11352 | 11996 | 11064 | 11549 | - | 11857 | **12659** | 11592 | 11647 | 11677 | 11427 | 11606 | 11702 | - | 11680 | **11714** |
| Live555 | 2761 | 2775 | 2102 | 2367 | 2127 | 2230 | 2236 | 2153 | **2798** | 2112 | 2185 | 1886 | 2032 | 2142 | 2034 | 1977 | 2197 | **2227** |
| DCMTK | 8369 | 7826 | 3062 | **8728** | 6560 | 6896 | - | 6499 | 8615 | 7407 | 7825 | 7720 | 7833 | 7623 | 7137 | - | 6418 | **7846** |
| Exim | 3255 | 2970 | 3116 | **3347** | 3253 | 2714 | 3258 | 3213 | 3282 | 2703 | **2804** | 2741 | 2743 | 2756 | 2714 | 2762 | 2736 | 2784 |
| Dnsmasq | 1173 | 1157 | 804 | 1064 | 838 | - | - | 1144 | **1184** | 1099 | 1089 | 1094 | 1045 | 1037 | - | - | 1094 | **1104** |
| Curl | 8755 | 9617 | - | 7627 | - | 4542 | - | - | **9641** | 5162 | **5332** | - | 5171 | - | 5099 | - | - | 5209 |
| MbedTLS | 1562 | **1565** | - | 1556 | - | 1554 | - | - | 1563 | **1545** | 1544 | - | 1539 | - | **1545** | - | - | **1545** |
| TinyDTLS | **620** | **620** | 517 | 611 | 515 | - | - | 591 | **620** | 578 | 557 | 531 | 580 | 548 | - | - | 585 | **597** |
| **Improvement** | 3.7% | 3.3% | 37.0% | 5.8% | 25.3% | 40.2% | 13.4% | 11.9% | | 2.8% | 2.2% | 7.0% | 3.3% | 6.4% | 18.3% | 4.8% | 4.0% | |

**Table 5**
Average branch coverage after 24 hours (left), and branch coverage after 1 million fuzzing iterations (right). The *Improvement* indicates the percentage improvement of CSFuzzer over each corresponding fuzzer, and the symbol - indicates that the fuzzer does not support that subject.

| Subject | State Coverage for 24H | | | | | | | | | State Coverage for 1M | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | AFL | AFL++ | AFLNet | IJON | StateAFL | SGFUZZ | ChatAFL | NSFuzz | CSFuzzer | AFL | AFL++ | AFLNet | IJON | StateAFL | SGFUZZ | ChatAFL | NSFuzz | CSFuzzer |
| PureFTPD | 151 | 161 | 123 | 297 | 117 | 8 | 136 | 134 | **327** | 143 | 128 | 138 | 285 | 127 | 8 | 132 | 101 | **292** |
| BFTPD | 715 | 681 | 760 | 1338 | 431 | 106 | 636 | 784 | **2508** | 421 | 456 | 1002 | 719 | 519 | 106 | 893 | 996 | **2506** |
| ippsample | 471 | 445 | 189 | 666 | - | 223 | 235 | - | **712** | 271 | 367 | 194 | 300 | - | 270 | 238 | - | **371** |
| CUPS | 1884 | 2068 | 449 | 2257 | - | 469 | 530 | - | **2541** | 648 | 670 | 456 | 994 | - | 269 | 540 | - | **1543** |
| OpenSSL | 743 | 739 | 661 | 704 | 626 | 604 | - | 636 | **766** | 687 | 667 | 694 | 678 | 646 | 653 | - | 637 | **696** |
| Live555 | 253 | 254 | 169 | 270 | 232 | 223 | 197 | 245 | **553** | 302 | 294 | 232 | 309 | 361 | 314 | 210 | 286 | **527** |
| DCMTK | 239 | 139 | 140 | 275 | 183 | 161 | - | 172 | **334** | 184 | 109 | 199 | 219 | 183 | 161 | - | 164 | **269** |
| Exim | 190 | 205 | 174 | **530** | 109 | 80 | 247 | 180 | 526 | 168 | 213 | 108 | 320 | 109 | 80 | 255 | 106 | **511** |
| Dnsmasq | 283 | 301 | 231 | 423 | 180 | - | - | 250 | **663** | 238 | 315 | 238 | 247 | 192 | - | - | 230 | **420** |
| Curl | 1908 | 1843 | - | 981 | - | 503 | - | - | **2112** | 702 | **1089** | - | 930 | - | 751 | - | - | 928 |
| MbedTLS | 259 | 257 | - | 255 | - | 224 | - | - | **261** | 259 | 269 | - | 196 | - | 271 | - | - | **274** |
| TinyDTLS | 238 | 246 | 183 | 330 | 91 | - | - | 209 | **446** | 226 | 234 | 183 | 176 | 162 | - | - | 187 | **347** |
| **Improvement** | 85.4% | 90.9% | 205.3% | 38.4% | 243.2% | 809.1% | 218.5% | 134.4% | | 105.7% | 95.7% | 129.3% | 57.7% | 151.9% | 703.0% | 132.5% | 131.1% | |

**Table 6**
Average CAST-Coverage after 24 hours (left), and CAST-Coverage after 1 million fuzzing iterations (right). The *Improvement* indicates the percentage improvement of CSFuzzer over each corresponding fuzzer, and the symbol - indicates that fuzzer does not support that subject.

ChatAFL currently only supports text-based protocols [36], it only supports six protocol programs as shown in Table 5.

We further analyzed the reasons why CSFuzzer underperforms compared to some fuzzers in certain scenarios. First, CAST-Coverage is less sensitive in terms of feedback compared to IJON and SGFuzz. Moreover, when the number of observed pstate-variable values is limited, CAST-Coverage degrades to ST-Coverage, further reducing coverage exploration. Thus, in scenarios where the number of state values is small, such as in DCMTK and Exim, CSFuzzer performs worse than IJON. Second, CSFuzzer is implemented based on AFL, while AFL++ incorporates a range of advanced fuzzing strategies beyond those in AFL. Consequently, CSFuzzer may be outperformed by AFL++ on certain targets. Similarly, ChatAFL leverages large language models to generate more diverse and high-quality test cases, which are difficult to generate through traditional mutation strategies within a limited time budget. Therefore, in such cases, CSFuzzer also lags behind ChatAFL. Nonetheless, we believe that CSFuzzer is complementary to AFL++ and ChatAFL, which can also be easily integrated into these advanced fuzzers to further enhance state exploration performance.

## 6.4. State Coverage Results (RQ3)

To evaluate the ability of CSFuzzer to explore more state spaces, we compare the state coverage of CSFuzzer with other fuzzers. We measured the state coverage of a fuzzer as the number of paths in the state bitmap that is constructed across the execution of all seeds generated throughout the campaign and from the initial corpus [21]. To ensure fairness, we also set up a 24-hour and 1 million evaluation for state coverage as mentioned in Section 6.3.

Table 6 shows the average CAST-Coverage results after 24 hours and 1 million fuzzing iterations for all campaigns. Overall, CSFuzzer achieves more state coverage than other fuzzers in most programs. On average, CSFuzzer achieves 85.4% more state coverage than AFL, 90.9% than AFL++, 205.3% than AFLNet, 38.4% than IJON, 243.2% than StateAFL, 809.1% than SGFUZZ, 218.5% than ChatAFL, and 134.4% than NSFuzz, respectively, in the setting of 24 hours. Similarly, CSFuzzer can achieve higher state coverage for the same number of iterations. On average, CSFuzzer achieves 105.7% more state coverage than AFL, 95.7% than AFL++, 129.3% than AFLNet, 57.7% than IJON, 151.9% than StateAFL, 703% than SGFUZZ, 132.5% than ChatAFL, and 134.4% than NSFuzz, respectively. We provide the p-value of these results in Appendix A.

Through further investigation, we found that CSFuzzer improves feedback sensitivity by maintaining state context

| Subject | Bug | AFL | AFL++ | AFLNET | IJON | StateAFL | SGFUZZ | ChatAFL | NSFuzz | CSFuzzer |
|---|---|---|---|---|---|---|---|---|---|---|
| | 942 | - | - | - | - | - | - | - | - | **5.41h** |
| | LEAK | - | | - | - | - | - | - | - | **0.49h** |
| DCMTK | SEGV | 2.76h | 1.97h | 7.84h | 0.29h | 6.51h | - | - | 6.12h | **0.07h** |
| | CVE | - | - | - | **22.31h** | - | - | - | - | 23.17h |
| | STACK | **0.15h** | **0.15h** | - | 0.17h | 0.16h | - | - | 0.24h | **0.15h** |
| | CVE1 | 1.05h | 0.96h | - | 1.05h | - | 1.47h | - | - | **0.85h** |
| Live555 | CVE2 | 0.43h | 0.37h | - | 7.10h | - | 0.31h | - | - | **0.29h** |
| | CVE3 | **0.06h** | 0.07h | 4.03h | 7.11h | 3.86h | 0.71h | 0.42h | 3.54h | 0.39h |
| | CVE4 | 0.05h | 0.05h | 5.88h | 2.95h | 6.17h | 0.36h | 2.14h | 5.51h | **0.04h** |
| | CVE | 0.61h | 0.69h | 11.49h | 1.86h | 10.53h | - | - | 10.81h | **0.51h** |
| | STACK1 | 0.65h | **0.01h** | 4.14h | 4.14h | 4.12h | - | - | 4.14h | 0.13h |
| | STACK2 | 0.61h | 0.74h | 11.49h | 1.87h | 5.46h | - | - | 5.34h | **0.51h** |
| Dnsmasq | HEAP1 | **0.01h** | **0.01h** | 0.67h | 0.04h | 0.73h | - | - | 0.65h | **0.01h** |
| | HEAP2 | 0.84h | 0.93h | - | 4.56h | - | - | - | - | **0.61h** |
| | HEAP3 | 0.09h | 0.42h | 3.91 | 0.08h | 2.62h | - | - | 2.45h | **0.08h** |
| | 544819 | 1.76h | 0.86h | - | 3.63h | - | - | - | 4.52h | **0.37h** |
| TinyDTLS | STACK1 | **0.01h** | **0.01h** | 0.07 | 0.01h | 0.09h | - | - | 0.01h | **0.01h** |
| | STACK2 | **0.01h** | **0.01h** | 0.47 | 0.08h | 0.57h | - | - | 0.01h | **0.01h** |
| | STACK3 | **0.01h** | **0.10h** | 0.11 | 0.02h | 0.15h | - | - | 0.01h | **0.01h** |
| ippsample | CNVD | 6.20h | 3.22h | 16.47 | **1.22h** | - | 6.92h | - | - | 2.03h |
| CUPS | CVE | - | - | - | - | - | - | - | - | **6.22h** |
| **Average Factor** | | 3.9x | 3.5x | 44.6x | 10.9x | 42.3x | 3.4x | 27.3x | 31.6x | |

**Table 7**
Time to exposure for all vulnerabilities found in experiments. The *Average Factor* indicates the factor improvement of CSFuzzer compared to each corresponding fuzzer. The symbol - means that the fuzzer cannot expose the bug within 24 hours or does not support that subject.

for each sstate-variable, which strikes a balance between retaining fuzzing efficiency and feedback sensitivity, and so exhibits the best state exploration capability under most protocol programs. IJON slightly outperforms CSFuzzer when fuzzing PureFTPD and Exim, as Table 6 shows. This is because IJON tracks the four consecutive state variables, which have higher feedback sensitivity. Meanwhile, IJON can avoid seed explosion problems since these two programs have a limited number of states. However, based on the state and branch coverage results, we believe that CSFuzzer performs more efficiently and consistently than IJON in most protocol implementations. It is necessary to clarify that since SGFUZZ does not execute successfully in some programs, it has the same results in the 24 hours and 1 million setups, e.g. it does not support the fork system call [21] without being able to keep track of the coverage, which results in not being able to save new seeds in PureFTPD.

## 6.5. Vulnerability Detection (RQ4)

To evaluate the vulnerability detection capability of CSFuzzer, we analyze it from two perspectives: the efficiency of vulnerability discovery in the benchmark program and the discovery of new vulnerabilities. We responsibly disclosed all vulnerabilities in our evaluation to the stakeholders, and at the time of paper writing, all of them have been fixed.

Table 7 shows the average time to exposure for all vulnerabilities found in the benchmark program. The *Bug* indicates the simplified vulnerability identifier, and due to space constraints, we provide the full description for bugs in the supplemental file. CSFuzzer can discover all these 21 protocol vulnerabilities in evaluation, while AFL, AFL++, AFLNET, IJON, StateAFL, SGFUZZ, ChatAFL, and NSFuzz were able to discover 17, 17, 12, 18, 12, 5, 2, and 13 of these vulnerabilities, respectively. On average, CSFuzzer triggers these vulnerabilities 3.9x faster than AFL, 3.5x than AFL++, 44.6x than AFLNET, 10.9x than IJON, 42.3x than StateAFL, 3.4x than SGFUZZ, 27.3x than ChatAFL, and 31.6x than NSFuzz.

Table 8 shows the basic information about 10 new vulnerabilities discovered by CSFuzzer, and 9 CVE/CNVD IDs have been assigned, where *CVE-2022-43272* and *CVE-2023-50656* in DCMTK correspond to *LEAK* and *SEGV* of DCMTK in Table 7, and the two new bugs of ippsample and CUPS also correspond to Table 7, respectively. CSFuzzer could find them more efficiently than all other fuzzers except for ippsample. On the other hand, we also evaluated CSFuzzer's vulnerability detection capabilities in a well-known MQTT protocol program, NanoMQ [15] for generalizability, where CSFuzzer could discover five new vulnerabilities. We evaluated whether other baseline fuzzers can discover them as well; specifically, AFL, AFL++, and IJON could discover three bugs, e.g., *CVE-2023-33658*, *CVE-2023-33659*, and *CVE-2023-33660*. AFLNet, StateAFL, and ChatAFL could only find *CVE-2023-33658* in 24 hours, while SGFUZZ does not support fuzzing this project, thus, it cannot find all of them.

Overall, CSFuzzer has good vulnerability detection capabilities in different types of protocol implementations, which is attributed to the fact that *context-aware state feedback* not only facilitates the exploration of protocol states while maintaining high fuzzing efficiency, which enhances protocol vulnerability discovery than other fuzzers. It is worth noting that CSFuzzer may perform worse than AFL and AFL++ in detecting certain easily triggered vulnerabilities (e.g., CVE-3 in Live555 and STACK1 in Dnsmasq). This is because CSFuzzer saves messages that cover new states as seeds, which increases the length of the seed queue. Compared to AFL, CSFuzzer may delay the selection of subsequent seeds, thereby reducing its efficiency in triggering such vulnerabilities within a short time.

| Subject | Version | Bug Type | CVE/CNVD | CVSS |
|---|---|---|---|---|
| DCMTK | v3.6.7 | Memory leak | CVE-2022-43272 | 7.5 |
| DCMTK | v3.6.7 | Memory leak | CVE-2022-43272 | 7.5 |
| DCMTK | v3.6.7 | Segmentation fault | CVE-2023-50656 | 7.5 |
| ippsample | #baf8b77 | Segmentation fault | CNVD-2022-44199 | 4.6 |
| CUPS | v2.4.2 | Heap-buffer overflow | CVE-2023-32324 | 7.5 |
| NanoMQ | v0.17.2 | Heap-buffer overflow | CVE-2023-33658 | 7.5 |
| NanoMQ | v0.17.2 | Heap-buffer overflow | CVE-2023-33659 | 7.5 |
| NanoMQ | v0.17.2 | Heap-buffer overflow | CVE-2023-33660 | 7.5 |
| NanoMQ | v0.17.2 | Heap use after free | CVE-2023-33657 | 7.5 |
| NanoMQ | v0.17.2 | Memory leak | CVE-2023-33656 | 5.5 |

**Table 8**
New vulnerabilities found by CSFuzzer.

| Subject | Average Branch Coverage | | | Average State Coverage | | |
|---|---|---|---|---|---|---|
| | CSFuzzer$_x$ | CSFuzzer | Improvement | CSFuzzer$_x$ | CSFuzzer | Improvement |
| PureFTPD | 1364 | **1404** | 2.9% | 254 | **327** | 22.3% |
| BFTPD | 507 | **512** | 1.0% | **2260** | 2058 | -9.8% |
| ippsample | 3238 | **3273** | 1.1% | 488 | **712** | 31.5% |
| CUPS | 3985 | **4121** | 3.3% | 1851 | **2541** | 27.2% |
| OpenSSL | 12295 | **12659** | 2.9% | 755 | **766** | 1.4% |
| Live555 | 2658 | **2798** | 5.0% | 283 | **553** | 48.8% |
| DCMTK | 8371 | **8615** | 2.8% | 244 | **334** | 27.0% |
| Exim | **3289** | 3282 | -0.2% | 353 | **526** | 32.9% |
| Dnsmasq | 1137 | **1184** | 4.0% | 654 | **663** | 1.4% |
| Curl | 8360 | **9641** | 13.3% | 1357 | **2112** | 35.8% |
| MbedTLS | 1561 | **1563** | 0.1% | 259 | **261** | 0.8% |
| TinyDTLS | **620** | **620** | 0.0% | 362 | **446** | 18.8% |
| | | | **Avg: 3.0%** | | | **Avg: 19.8%** |

**Table 9**
Average branch and state coverage results for ablation study in CSFuzzer$_x$ and CSFuzzer over 24 hours.

## 6.6. Ablation Study (RQ5)

To understand whether classifying state variables into pstate-variables and sstate-variables to construct CAST-Coverage has an impact on the performance of CSFuzzer, we implemented another fuzzer named CSFuzzer$_x$, which identified and tracked same state variables as CSFuzzer but does not classify state variables and uses ST-Coverage to construct state feedback. Table 9 shows the average branch and state coverage results after 24 hours for CSFuzzer and CSFuzzer$_x$. Overall, CSFuzzer achieves 3.0% more branch coverage and 19.8% more state coverage than CSFuzzer$_x$, demonstrating that categorizing state variables into two categories for constructing CAST-Coverage as state feedback is more effective than directly using all state variables to guide the exploration of the code and state space of protocol implementations. This is because it effectively distinguishes the same sub-state transitions in different protocol states and improves the sensitivity of state feedback.

On the other hand, we also analyzed in detail the impact of each heuristic approach introduced in Section 4.2 on the identification of the state variable results. Table 10 shows the statistical results of the number of state variables identified in each step, where *Step1* corresponds to Section 4.2.2, *Step2* corresponds to Section 4.2.3, including three approaches, e.g., *Indirect* indicates the code features in the indirect call statement, *Function* indicates the function name characteristic of branch statements, *Complexity* indicates the code complexity of branch statements. *Step3* and *Step4* corresponds to Section 4.2.4 and Section 4.2.5, respectively. *Step5* corresponds to Section 4.2.6 including two approaches, e.g., *Substring* indicates the sub-string matching algorithm, and *Word2vec* indicates the cosine similarity algorithm. Overall,

| Subject | Step1 | Step2 | | | Step3 | Step4 | Step5 | | Total |
|---|---|---|---|---|---|---|---|---|---|
| | | Indirect | Function | Complexity | | | Substring | Word2vec | |
| PureFTPD | 131 | 0 | 2 | 0 | 1 | 1 | 10 | 0 | 12 |
| BFTPD | 88 | 4 | 1 | 0 | 1 | 1 | 7 | 0 | 12 |
| ippsample | 365 | 0 | 11 | 3 | 8 | 1 | 56 | 2 | 72 |
| CUPS | 596 | 0 | 27 | 3 | 12 | 1 | 96 | 4 | 130 |
| OpenSSL | 5898 | 60 | 144 | 1 | 162 | 3 | 470 | 5 | 680 |
| Live555 | 467 | 0 | 5 | 2 | 3 | 1 | 122 | 3 | 132 |
| DCMTK | 1849 | 0 | 22 | 1 | 5 | 1 | 77 | 2 | 102 |
| Exim | 705 | 1 | 28 | 2 | 25 | 1 | 166 | 4 | 201 |
| Dnsmasq | 244 | 0 | 0 | 2 | 2 | 0 | 87 | 0 | 89 |
| Curl | 4135 | 4 | 80 | 2 | 21 | 8 | 538 | 7 | 631 |
| MbedTLS | 832 | 0 | 13 | 2 | 4 | 1 | 109 | 0 | 122 |
| TinyDTLS | 86 | 0 | 3 | 0 | 3 | 1 | 14 | 0 | 17 |

**Table 10**
Results of the state variables identified by each step in CSFuzzer.

each heuristic approach contributes to the identification of state variables to varying degrees, where *Step1* identifies the range of potential state variables from a huge number of variables, *Step2* and *Step3* then identifies a few candidate pstate-variables, which are then confirmed by *Step4*. Lastly, *Step5* identifies the remaining sstate-variables, proving the effectiveness of our approach.

## 7. Threats to Validity

The first concern is external validity, i.e., the generality. CSFuzzer is based on the investigation and analysis of the 40 protocol implementations presented in Table 1. It is necessary to mention that seven of the protocol implementations used for evaluation overlap with the 40 implementations investigated. In addition, due to software implementations' variability, CSFuzzer may not apply to subjects we have not tested. By analyzing existing work [21, 36], our evaluation has maximally covered more types of key protocol implementations, and in particular, we have additionally added five protocol implementations that were not in the scope of the initial study in Table 1 and found nine new vulnerabilities, further validating the effectiveness of CSFuzzer.

The second concern is internal validity, i.e., how systematic errors can be minimized in the study. One of the internal threats of fuzzing experiments is the selection of initial test cases [23]. To ensure the validity, we used off-the-shelf corpora, such as the seeds provided by default by AFLNet and other fuzzers. In addition, considering that the performance differences of different fuzzers may affect the fair evaluation of different state-guided strategies, we added experiments with 1 million fuzzing iterations to ensure the fairness and reliability of the evaluation results.

## 8. Related Work

Recent years have witnessed significant advances in protocol fuzzing, which can be broadly classified into two categories: black-box and grey-box approaches.

**Black-box Protocol Fuzzing.** Black-box fuzzing is particularly well-suited for protocol testing, as it does not require source code access or instrumentation, making it applicable to both closed-source implementations and open-source implementations written in different languages. Peach [11] is a comprehensive black-box fuzzing framework that supports testing of different types of network protocols. Bleem [35] performs mutations on packet sequences exchanged between protocol clients and servers and introduces state feedback mechanisms for black-box fuzzing. RESOLVERFUZZ [55] uses differential testing to test DNS resolvers to find non-crash vulnerabilities in them. LLMIF [48] incorporates large language models into IoT protocol fuzzing to extract Zigbee protocol information and infer device response states. SP-Fuzz [53] uses protocol state and data models to generate stateful paths and then divide them into discrete tasks for assignment to improve fuzzing performance. DY Fuzzing [18] employs formal models to explore the logic of cryptographic protocols and detect design-level flaws. While black-box fuzzing has shown encouraging results, its lack of internal program insight limits its ability to effectively guide exploration through program logic and protocol states.

**Grey-box Protocol Fuzzing.** In contrast, grey-box fuzzing leverages internal program feedback (e.g., code coverage) to guide input generation and mutation. Coverage-guided fuzzers such as AFL [2], libFuzzer [9], and Enfuzz [22] can be applied to protocol fuzzing, but their lack of protocol state awareness limits their effectiveness. To address this, several works have introduced protocol-specific state feedback. AFLNet and its variants [41, 42, 34] regard the response codes in packets to construct state feedback, which suffers from the problem of uninformative response codes [21]. StateAFL [38] infers the protocol state by analyzing memory contents, which ignores state changes during packet processing. IJON [20] proposes to annotate state variables manually. SGFUZZ [21] recognizes enum-type variables and tracks state values along the state transition path, which does not consider other non-enum-type variables and may cause seed explosion. NSFuzz [42] only focuses on variables that represent the state of network services (usually a very small number) to increase fuzzing performance, but does not utilize these state variables to guide state space exploration. Logos [51] collects log information about the protocol implementation and embeds it into a high-level vector space for semantic representation, dynamically maintaining semantic coverage to guide the drawing of the exploration space. CSFuzzer extends this line of research by identifying two categories of state variables and introducing CAST-Coverage, which provides more sensitive feedback while mitigating seed explosion.

Additionally, many works focus on improving the overall performance of grey-box fuzzing by optimizing I/O performance. MultiFuzz [54] enhances *desock* for improved network fuzzing. SNPSFuzzer [32] and Nyx-Net [47] use snapshot mechanisms for state preservation and recovery. NSFuzz [42] modifies the *forkserver* in AFLNet to better synchronize with signal-based feedback. SnapFuzz [19] optimizes network communication and file I/O with an in-memory file system, while HNPFuzzer [30] further reduces message-passing overhead by leveraging shared memory.

## 9. Conclusion

In this paper, we analyzed the composition of the protocol state space and proposed a novel context-aware state-guided fuzzing approach CSFuzzer. CSFuzzer classifies state variables into two categories through state contexts and automatically identifies them through static and dynamic analysis. Then CSFuzzer uses a new coverage metric named *context-aware state transition coverage* to track state values and applies a rarity-preferred energy scheduling strategy to facilitate state exploration. CSFuzzer has higher coverage accuracy and can mitigate seed explosion. We implemented an CSFuzzer prototype, and experiments show that it could achieve higher code and state coverage than existing approaches and find most vulnerabilities faster than others. Besides, CSFuzzer found 10 zero-day vulnerabilities.

## CRedit authorship contribution statement

**Xiangpu Song**: Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Yingpei Zeng**: Methodology, Validation, Writing – review & editing, Supervision. **Jianliang Wu**: Methodology, Writing – review & editing. **Hao Li**: Writing – review & editing. **Chaoshun Zuo**: Writing – review & editing. **Qingchuan Zhao**: Writing – review & editing. **Shanqing Guo**: Methodology, Validation, Supervision, Writing – review & editing.

## Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62372268), the Major Scientific and Technological Innovation Projects of Shandong Province, China (No. 2024CXGC010114), and the Shandong Provincial Natural Science Foundation, China (No. ZR2022LZH013, No. ZR2021LZH007). It is also supported in part by the Zhejiang Provincial Natural Science Foundation of China (No. LY22F020022) and the National Natural Science Foundation of China (No. 61902098).

## Appendix.A Statistical difference in coverage

Table A.1 and Table A.2 present the p-values obtained from the Mann-Whitney U-test, comparing the average branch and state coverage of each fuzzer after 24 hours and 1 million fuzzing iterations, respectively. The terms *p1*, *p2*, *p3*, *p4*, *p5*, *p6*, and *p7* represent the differences between CSFuzzer and AFL, AFL++, AFLNet, IJON, StateAFL, SGFUZZ, and ChatAFL, respectively. Overall, CSFuzzer performs better than the other fuzzers in most subjects, and the corresponding p-values are mostly less than 0.05, with the results possessing significant differences.

## Appendix.B Detailed bug identifier in evaluation

Table B.1 shows the detailed bug descriptions corresponding to the vulnerability identifiers and types in our

| Subject | Branch Coverage for 24 hours | | | | | | | | Branch Coverage for 1 million iterations | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 |
| PureFTPD | $2.4 * 10^{-4}$ | $4.4 * 10^{-4}$ | $1.8 * 10^{-4}$ | $4.9 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.3 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $3.1 * 10^{-2}$ | $4.6 * 10^{-3}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.3 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.7 * 10^{-2}$ |
| BFTPD | $1.8 * 10^{-4}$ | $2.3 * 10^{-2}$ | $2.4 * 10^{-4}$ | $1.1 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.3 * 10^{-4}$ | $1.7 * 10^{-3}$ | $3.3 * 10^{-4}$ | $2.5 * 10^{-3}$ | $3.2 * 10^{-4}$ | $4.7 * 10^{-2}$ | $3.1 * 10^{-2}$ | $4.6 * 10^{-3}$ | $1.3 * 10^{-4}$ | $4.9 * 10^{-2}$ | $1.5 * 10^{-1}$ |
| ippsample | $7.3 * 10^{-3}$ | $1.3 * 10^{-2}$ | $1.8 * 10^{-4}$ | $3.1 * 10^{-2}$ | - | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $2.8 * 10^{-3}$ | $2.3 * 10^{-1}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - |
| CUPS | $1.3 * 10^{-3}$ | $6.4 * 10^{-2}$ | $1.8 * 10^{-4}$ | $4.6 * 10^{-3}$ | - | $8.7 * 10^{-5}$ | $1.8 * 10^{-4}$ | - | $4.6 * 10^{-1}$ | $4.1 * 10^{-2}$ | $1.8 * 10^{-4}$ | $3.2 * 10^{-4}$ | - | $1.6 * 10^{-4}$ | $1.8 * 10^{-4}$ | - |
| OpenSSL | $3.3 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | $1.5 * 10^{-3}$ | $2.8 * 10^{-2}$ | $2.7 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.7 * 10^{-3}$ | $6.5 * 10^{-1}$ | - | $3.8 * 10^{-1}$ |
| Live555 | $1.0 * 10^{-3}$ | $3.2 * 10^{-3}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $2.2 * 10^{-3}$ | $2.4 * 10^{-2}$ | $1.8 * 10^{-4}$ | $2.5 * 10^{-4}$ | $2.6 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $4.6 * 10^{-2}$ |
| DCMTK | $3.8 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $2.6 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $5.4 * 10^{-2}$ | $1.8 * 10^{-4}$ | $6.4 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.1 * 10^{-4}$ | - | $1.8 * 10^{-4}$ |
| Exim | $9.6 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $2.4 * 10^{-4}$ | $1.4 * 10^{-2}$ | $1.1 * 10^{-4}$ | $1.0 * 10^{-2}$ | $1.8 * 10^{-4}$ | $4.4 * 10^{-4}$ | $1.5 * 10^{-3}$ | $5.8 * 10^{-4}$ | $2.4 * 10^{-4}$ | $3.6 * 10^{-3}$ | $1.1 * 10^{-4}$ | $7.2 * 10^{-3}$ | $1.8 * 10^{-4}$ |
| Dnsmasq | $2.1 * 10^{-2}$ | $7.7 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | - | $3.3 * 10^{-4}$ | $6.2 * 10^{-1}$ | $2.4 * 10^{-2}$ | $1.7 * 10^{-1}$ | $4.4 * 10^{-4}$ | $2.4 * 10^{-4}$ | - | - | $1.3 * 10^{-1}$ |
| Curl | $1.8 * 10^{-4}$ | $1.2 * 10^{-1}$ | - | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | - | - | $3.8 * 10^{-4}$ | $3.3 * 10^{-4}$ | - | $5.0 * 10^{-4}$ | - | $1.5 * 10^{-4}$ | - | - |
| MbedTLS | $8.8 * 10^{-1}$ | $2.4 * 10^{-2}$ | - | $6.4 * 10^{-3}$ | - | $2.6 * 10^{-2}$ | - | - | $8.2 * 10^{-1}$ | $1.4 * 10^{-1}$ | - | $2.8 * 10^{-2}$ | - | $5.7 * 10^{-1}$ | - | - |
| TinyDTLS | $8.8 * 10^{-1}$ | $8.5 * 10^{-1}$ | $1.8 * 10^{-4}$ | $3.4 * 10^{-2}$ | $1.8 * 10^{-4}$ | - | - | $1.8 * 10^{-4}$ | $2.1 * 10^{-3}$ | $1.1 * 10^{-3}$ | $1.8 * 10^{-4}$ | $1.1 * 10^{-3}$ | $1.8 * 10^{-4}$ | - | - | $2.8 * 10^{-2}$ |

**Table A.1**
The p-value of branch coverage for 24 hours and 1 million iterations of fuzzing campaigns.

| Subject | State Coverage for 24 hours | | | | | | | | State Coverage for 1 million iterations | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 |
| PureFTPD | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.7 * 10^{-3}$ | $1.8 * 10^{-4}$ | $8.7 * 10^{-5}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.1 * 10^{-1}$ | $1.8 * 10^{-4}$ | $1.3 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ |
| BFTPD | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.1 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ |
| ippsample | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.5 * 10^{-3}$ | - | $1.3 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | $8.5 * 10^{-1}$ | $1.8 * 10^{-4}$ | $3.3 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - |
| CUPS | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - |
| OpenSSL | $4.5 * 10^{-2}$ | $6.9 * 10^{-2}$ | $9.9 * 10^{-4}$ | $4.9 * 10^{-2}$ | $4.4 * 10^{-4}$ | $3.3 * 10^{-4}$ | - | $3.3 * 10^{-4}$ | $4.3 * 10^{-2}$ | $1.1 * 10^{-3}$ | $8.2 * 10^{-1}$ | $2.3 * 10^{-2}$ | $1.3 * 10^{-3}$ | $5.7 * 10^{-4}$ | - | $4.4 * 10^{-4}$ |
| Live555 | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $4.4 * 10^{-4}$ | $3.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.0 * 10^{-3}$ | $4.4 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ |
| DCMTK | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $2.0 * 10^{-3}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $3.6 * 10^{-3}$ | $1.8 * 10^{-4}$ | $2.0 * 10^{-3}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ |
| Exim | $1.3 * 10^{-2}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $4.6 * 10^{-4}$ | $4.4 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.2 * 10^{-3}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ |
| Dnsmasq | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | - | $1.8 * 10^{-4}$ | $1.4 * 10^{-3}$ | $2.5 * 10^{-4}$ | $1.8 * 10^{-4}$ | $4.4 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | - | $1.8 * 10^{-4}$ |
| Curl | $1.8 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | - | - | $2.3 * 10^{-4}$ | $1.8 * 10^{-4}$ | - | $4.6 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | - | - |
| MbedTLS | $2.2 * 10^{-1}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | - | - | $2.1 * 10^{-1}$ | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | - | $1.8 * 10^{-4}$ | - | - |
| TinyDTLS | $8.7 * 10^{-1}$ | $7.2 * 10^{-1}$ | $1.8 * 10^{-4}$ | $2.6 * 10^{-2}$ | $1.8 * 10^{-4}$ | - | - | $1.8 * 10^{-4}$ | $8.6 * 10^{-1}$ | $6.7 * 10^{-1}$ | $1.8 * 10^{-4}$ | $1.3 * 10^{-2}$ | $1.8 * 10^{-4}$ | - | - | $1.8 * 10^{-4}$ |

**Table A.2**
The p-value of state coverage for 24 hours and 1 million iterations of fuzzing campaigns.

vulnerability detection efficiency experiments in Section 6.5. Since some vulnerabilities cannot determine their vulnerability identifiers, we use the top three layers of the function stack provided by ASAN as the description followed by [33].

## Appendix.C Threshold Sensitivity Analysis

We set three thresholds for the code feature analysis approach as mentioned in Section 4.2.3 and Section 6.1. The *threshold 2* and *threshold 3* have a greater impact on the identification results, indicating the number of most branches and code complexity, respectively. Thus, we evaluated the impact of the setting of these two thresholds on the state variable identification results.

Table C.1 shows the results of the sensitivity analysis. *P1* and *P2* indicate the branches and code complexity respectively. ○ indicates that the results are normal, ◐ indicates that it affects the number of variables to be analyzed in subsequent dynamic analysis, and ● indicates that it will lead to incorrect final results. We chose 40% and 100% as a comparison for *P1* because these three values incrementally represent a few, most, and all scenarios. We then chose 10 and 50 as a comparison for *P2* since they represent the two code complexity scenarios that are ideal and untestable in practice [3], respectively.

Overall, for *P1*, 70% can cover all the code features of most protocol programs with the best generalization ability and does not cause the results to be too broad or too limited to affect the recognition accuracy, as 40% or 100% shows. For *P2*, although the results are the same for the value 30 and below, we still choose 30 instead of 10 to avoid causing potential incorrect results and increasing the cost of dynamic analysis, and 30 is also a public threshold in software analysis indicating high code complexity [3, 25].

## References

[1] Advanced coverage metrics for object-oriented software. https://citeseerx.ist.psu.edu/document?doi=ae7e36d6bcaf877c35870a85262e280127db6188, Accessed on 2024-10-5.

[2] american fuzzy lop. https://lcamtuf.coredump.cx/afl/. Accessed on 2022-11-3.

[3] Avoiding spaghetti code. https://course.ece.cmu.edu/~ece642/lectures/09_SpaghettiCode.pdf, Accessed on 2023-9-14.

[4] Codeql. https://codeql.github.com/. Accessed on 2023-5-19.

[5] curl. https://curl.se/, Accessed on 2022-12-31.

[6] Gdb: The gnu project debugger. https://www.sourceware.org/gdb/, Accessed on 2022-12-31.

[7] Google word2vec. https://code.google.com/archive/p/word2vec/. Accessed on 2022-10-8.

[8] Introducing webm, an open web media project, webm project. https://webcitation.org/67CpWxl6p?url=http://blog.webmproject.org/2010/05/introducing-webm-open-web-media-project.html. Accessed on 2023-12-9.

[9] libfuzzer – a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html. Accessed on 2022-11-3.

[10] Mpeg program stream. https://en.wikipedia.org/wiki/MPEG_program_stream. Accessed on 2023-12-9.

[11] Peach fuzzer. https://github.com/MozillaSecurity/peach. Accessed on 2022-11-3.

[12] Preeny. https://github.com/zardus/preeny. Accessed on 2022-11-3.

[13] Rfc 2326: Real time streaming protocol (rtsp). https://www.rfc-editor.org/rfc/rfc2326.html, Accessed on 2022-12-31.

[14] Rfc 2565 - internet printing protocol/1.0: Encoding and transport. https://datatracker.ietf.org/doc/html/rfc2565. Accessed on 2023-12-9.

[15] An ultra-lightweight and blazing-fast messaging broker/bus for iot edge & sdv. https://github.com/nanomq/nanomq. Accessed on 2023-12-9.

[16] Whole program llvm: wllvm ported to go. https://github.com/SRI-CSL/gllvm. Accessed on 2022-7-25.

| Subject | Bug | Complete Bug ID or Stack Information |
|---|---|---|
| DCMTK | 942 | DCMTK Bug Tracker#942 |
| | LEAK | ['operator new', 'newValueField', 'loadValue'] |
| | SEGV | ['parseUserInfo', 'parseAssociate', 'AE_6_ExamineAssociateRequest'] |
| | CVE | CVE-2023-50656 |
| | STACK | ['OFStandard::my_strlcpy', 'OFStandard::strlcpy', 'DU_getStringDOElement'] |
| Live555 | CVE1 | CVE-2021-38382 |
| | CVE2 | CVE-2021-39282 |
| | CVE3 | CVE-2018-4013 |
| | CVE4 | CVE-2021-38381 |
| Dnsmasq | CVE | CVE-2017-14491 |
| | STACK1 | ['add_resource_record', 'answer_request', 'receive_query'] |
| | STACK2 | ['questions_crc', 'forward_query', 'receive_query'] |
| | HEAP1 | ['extract_name', 'questions_crc', 'forward_query'] |
| | HEAP2 | ['__interceptor_vsprintf', '__interceptor_sprintf', 'extract_name'] |
| | HEAP3 | ['extract_name', 'extract_request', 'receive_query'] |
| TinyDTLS | 544819 | Eclipse Bug Tracker#544819 |
| | STACK1 | ['dtls_sha256_transform', 'dtls_sha256_update', 'dtls_hash_update'] |
| | STACK2 | ['dtls_uint16_to_int', 'dtls_update_parameters', 'handle_handshake_msg'] |
| | STACK3 | ['__interceptor_memcpy', 'dtls_sha256_update', 'dtls_hash_update'] |
| ippsample | CNVD | CNVD-2022-44199 |
| CUPS | CVE | CVE-2023-32324 |

**Table B.1**
Detailed information about bugs found by fuzzers in evaluation.

| Subject | P1: 40% | P1: 70% | P1: 100% | P2: 10 | P2: 30 | P2: 50 |
|---|---|---|---|---|---|---|
| PureFTPD | ○ | ○ | ● | ○ | ○ | ○ |
| ippsample | ◐ | ○ | ● | ○ | ○ | ◐ |
| CUPS | ◐ | ○ | ● | ○ | ○ | ◐ |
| OpenSSL | ◐ | ○ | ● | ○ | ○ | ○ |
| Live555 | ○ | ○ | ◐ | ○ | ○ | ○ |
| DCMTK | ○ | ○ | ○ | ○ | ○ | ○ |
| Exim | ○ | ○ | ○ | ○ | ○ | ○ |
| Dnsmasq | ○ | ○ | ○ | ○ | ○ | ○ |
| Curl | ● | ○ | ● | ○ | ○ | ○ |
| MbedTLS | ○ | ○ | ● | ○ | ○ | ○ |
| TinyDTLS | ○ | ○ | ○ | ○ | ○ | ○ |

**Table C.1**
Sentivity analysis for code feature analysis.

[17] Bernhard K. Aichernig, Edi Muškardin, and Andrea Pferscher. Learning-based fuzzing of iot message brokers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 47–58, 2021.

[18] Max Ammann, Lucca Hirschi, and Steve Kremer. Dy fuzzing: formal dolev-yao models meet cryptographic protocol fuzz testing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1481–1499. IEEE, 2024.

[19] Anastasios Andronidis and Cristian Cadar. Snapfuzz: An efficient fuzzing framework for network applications. *arXiv preprint arXiv:2201.04048*, 2022.

[20] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

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

[22] 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.

[23] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[24] Dustin Boswell and Trevor Foucher. *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code.* " O'Reilly Media, Inc.", 2011.

[25] Bill Curtis, Jay Sappidi, and Jitendra Subramanyam. An evaluation of the internal quality of business applications: Does size matter? New York, NY, USA, 2011. Association for Computing Machinery.

[26] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for stateful systems: Survey and research directions. *ACM Computing Surveys*, 56(9):1–23, 2024.

[27] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846, 2021.

[28] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. Weizz: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 1–13, 2020.

[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[30] Junsong Fu, Shuai Xiong, Na Wang, Ruiping Ren, Ang Zhou, and Bharat K Bhargava. A framework of high-speed network protocol fuzzing based on shared memory. *IEEE Transactions on Dependable and Secure Computing*, 21(4):2779–2798, 2023.

[31] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[32] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. *arXiv preprint arXiv:2202.03643*, 2022.

[33] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[34] Dongge Liu, Van-Thuan Pham, Gidon Ernst, Toby Murray, and Benjamin I.P. Rubinstein. State selection algorithms and their impact on the performance of stateful network protocol fuzzing. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 720–730, 2022.

[35] 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 *32st USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023.

[36] 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.

[37] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[38] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering*, 27(7):1–31, 2022.

[39] Maria Leonor Pacheco, Max von Hippel, Ben Weintraub, Dan Goldwasser, and Cristina Nita-Rotaru. Automated attack synthesis by extracting finite state machines from protocol specification documents. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 51–68. IEEE, 2022.

[40] Yan Pan, Wei Lin, Liang Jiao, and Yuefei Zhu. Model-based grey-box fuzzing of network protocols. *Security and Communication Networks*, 2022, 2022.

[41] Van-Thuan Pham, Marcel Bohme, 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, Porto, Portugal, October 2020. IEEE.

[42] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology*, 2023.

[43] Krishan Sabnani and Anton Dahbura. A new technique for generating protocol test. *ACM SIGCOMM Computer Communication Review*, 15(4):36–43, 1985.

[44] Felice Salviulo and Giuseppe Scanniello. Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[45] Miro Samek. Practical statecharts in c/c++: Quantum programming for embedded systems with cdrom. pages 57–59. CRC Press, 2002.

[46] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993. IEEE, 2024.

[47] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 166–180, 2022.

[48] Jincheng Wang, Le Yu, and Xiapu Luo. Llmif: Augmented large language model for fuzzing iot devices. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 881–896. IEEE, 2024.

[49] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.

[50] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. 500(235), 1996.

[51] Feifan Wu, Zhengxiong Luo, Yanyang Zhao, Qingpeng Du, Junze Yu, Ruikang Peng, Heyuan Shi, and Yu Jiang. Logos: Log guided fuzzing for protocol implementations. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1720–1732, 2024.

[52] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. Pathafl: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 598–609, 2020.

[53] Junze Yu, Zhengxiong Luo, Fangshangyuan Xia, Yanyang Zhao, Heyuan Shi, and Yu Jiang. Spfuzz: Stateful path based parallel fuzzing for protocols in autonomous vehicles. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024.

[54] Yingpei Zeng, Mingmin Lin, Shanqing Guo, Yanzhao Shen, Tingting Cui, Ting Wu, Qiuhua Zheng, and Qiuhua Wang. Multifuzz: a coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols. *Sensors*, 20(18):5194, 2020.

[55] 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 *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4729–4746, 2024.

[56] Xiaohan Zhang, Cen Zhang, Xinghua Li, Zhengjie Du, Bing Mao, Yuekang Li, Yaowen Zheng, Yeting Li, Li Pan, Yang Liu, et al. A survey of protocol fuzzing. *ACM Computing Surveys*, 57(2):1–36, 2024.

[57] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. Statefuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, 2022.

[58] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.