Large Language Model guided Protocol Fuzzing

Ruijie Meng*, Martin Mirchev*, Marcel Böhme† and Abhik Roychoudhury*
*National University of Singapore
†MPI-SP and Monash University
{ruijie, mmirchev, abhik}@comp.nus.edu.sg, marcel.boehme@mpi-sp.org

***Abstract*—How to find security flaws in a protocol implementation without a machine-readable specification of the protocol? Facing the internet, protocol implementations are particularly security-critical software systems where inputs must adhere to a specific structure and order that is often informally specified in hundreds of pages in natural language (RFC). Without some machine-readable version of that protocol, it is difficult to automatically generate valid test inputs for its implementation that follow the required structure and order. It is possible to partially alleviate this challenge using mutational fuzzing on a set of recorded message sequences as seed inputs. However, the set of available seeds is often quite limited and will hardly cover the great diversity of protocol states and input structures.**

In this paper, we explore the opportunities of systematic interaction with pre-trained large language models (LLMs), which have ingested millions of pages of human-readable protocol specifications, to draw out machine-readable information about the protocol that can be used during protocol fuzzing. We use the knowledge of the LLMs about protocol message types for well-known protocols. We also checked the LLM's capability in detecting "states" for stateful protocol implementations by generating sequences of messages and predicting response codes. Based on these observations, we have developed an LLM-guided protocol implementation fuzzing engine. Our protocol fuzzer CHATAFL constructs grammars for each message type in a protocol, and then mutates messages or predicts the next messages in a message sequence via interactions with LLMs. Experiments on a wide range of real-world protocols from PROFUZZBENCH show significant efficacy in state and code coverage. Our LLM-guided stateful fuzzer was compared with state-of-the-art fuzzers AFLNET and NSFUZZ. CHATAFL covers 47.60% and 42.69% more state transitions, 29.55% and 25.75% more states, and 5.81% and 6.74% more code, respectively. Apart from enhanced coverage, CHATAFL discovered nine distinct and previously unknown vulnerabilities in widely-used and extensively-tested protocol implementations while AFLNET and NSFUZZ only discovered three and four of them, respectively.

I. INTRODUCTION

The development of an automatic vulnerability discovery tool for protocol implementations is particularly interesting both, from a practical and from a research point of view.

From a practical point of view, protocol implementations are the most exposed components of every software system that is directly or indirectly connected to the internet. Protocol implementations thus constitute a critical attack surface that must be automatically and continuously rid of security flaws. A simple arbitrary code execution vulnerability in a widely-used protocol implementation renders even the most secure software systems vulnerable to malicious remote attacks.

From a research point of view, protocol implementations constitute stateful systems that are difficult to test. The same input executed twice might give different outputs every time. Finding a vulnerability in a specific protocol state requires sending the right inputs in the right order. For instance, some protocols require an initialization or handshake message before other types of messages can be exchanged. For the receiver to properly parse that message and progress to the next state, the message must follow a specific format. However, by default, we can assume *neither* to know the correct structure *nor* the correct order of those messages.

Mutation-based protocol fuzzing reduces the dependence on a machine-readable specification of that required message structure or order by fuzzing *recorded* message sequences [1], [2], [3], [4]. The simple mutations often preserve the required protocol while still corrupting the message sequences enough to expose errors. However, the effectiveness of mutation-based protocol fuzzers is limited by the quality and diversity of the recorded seed message sequences, and the available simple mutations do not help in the effective coverage of the otherwise rich input or state space.

To foster the adoption of a protocol among the participants of the internet, almost all popular, widely-used protocols are specified in publicly available documents, which are often hundreds of pages long and written in natural language. What if we could programmatically *interrogate* the natural language specification of the protocol whose implementation we are testing? How could we use such an opportunity to resolve the challenges of existing approaches to protocol fuzzing?

In this paper, we explore the utility of large language models (LLMs) to guide the protocol fuzzing process. Fed with many terabytes of data from websites and documents on the internet, LLMs have recently been shown to accurately answer specific questions about any topic, at all. An LLM like ChatGPT 4.0 has also consumed natural-language protocol specifications. The recent, tremendous success of LLMs provides us with the opportunity to develop a system that puts a protocol fuzzer into a systematic interaction with the LLM, where the fuzzer

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24556
www.ndss-symposium.org

can issue very specific tasks to the LLM.

We call this approach *LLM-guided protocol fuzzing* and present three concrete components. Firstly, the fuzzer uses the LLM to extract a machine-readable grammar for a protocol that is used for structure-aware mutation. Secondly, the fuzzer uses the LLM to increase the diversity of messages in the recorded message sequences that are used as initial seeds. Lastly, the fuzzer uses the LLM to break out of a coverage plateau, where the LLM is prompted to generate messages to reach new states.

Our results for all text-based protocols in the PROFUZZBENCH protocol fuzzer benchmark [5] demonstrate the effectiveness of the LLM-guided approach: Compared to the baseline (AFLNET [1]) into which our approach was implemented, our tool CHATAFL covers almost 50% more state transitions, 30% more states, and 6% more code. CHATAFL shows similar improvements over the state-of-the-art (NSFUZZ [2]). In our ablation study, starting from the baseline we found that enabling (i) the grammar extraction, (ii) the seed enrichment, and (iii) the saturation handler one by one allows CHATAFL to achieve the same code coverage 2.0, 4.6, and 6.1 times faster, respectively, as the baseline achieves in 24 hours. CHATAFL is highly effective at finding critical security issues in protocol implementations. In our experiments, CHATAFL discovered nine distinct and previously unknown vulnerabilities in widely-used and extensively-tested protocol implementations.

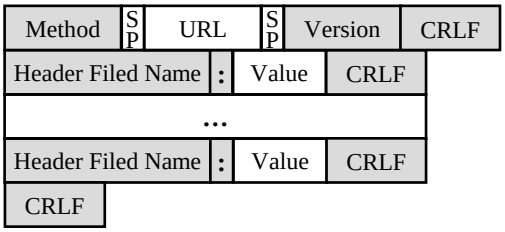
In summary, our paper makes the following contributions:

- We build a large language model (LLM) guided fuzzing engine for protocol implementations to overcome the challenges of existing protocol fuzzers. For deeper behavioral coverage of such protocols, on-the-fly state inference is needed - which is accomplished by interrogating an LLM, like ChatGPT, about the state machine and input structure of a given protocol.
- We present three strategies for integrating an LLM into a mutation-based protocol fuzzer, each of which explicitly addresses an identified challenge of protocol fuzzing. We develop an extended greybox fuzzing algorithm and implement it as a prototype CHATAFL. The tool is publicly available at

  https://github.com/ChatAFLndss/ChatAFL
- We conducted experiments that demonstrate that our LLM-guided stateful fuzzer prototype CHATAFL is substantially more effective than the state-of-the-art AFLNET and NSFUZZ in terms of the coverage of the protocol state space and the protocol implementation code. Apart from enhanced coverage, CHATAFL discovered nine previously unknown vulnerabilities in widely-used protocol implementations, the majority of which could not be found by AFLNET and NSFUZZ.

II. BACKGROUND AND MOTIVATION

We start by introducing the main technical concepts in protocol fuzzing and elucidating the key open challenges that we seek to address in this paper. We then provide some background on large language models and our motivation.



(a) Structure of RTSP client requests.

```
PLAY rtsp://127.0.0.1:8554/aacAudioTest/ RTSP/1.0\r\n
CSeq: 4\r\n
User-Agent: ./testRTSPClient (LIVE555 Streaming Media v2018.08.28)\r\n
Session: 000022B8\r\n
Range: npt=0.000-\r\n
\r\n
```

(b) Example of RTSP PLAY client request from Live555.

Fig. 1. Structure of RTSP client requests in (a), and a PLAY client request from Live555 in (b).

A. Protocol Fuzzing

In order to facilitate the systematic and reliable exchange of information on the Internet, all participants agree to use a common protocol. Many of the most widely-used protocols have been designed by the Internet Engineering Task Force (IETF) and published as Request for Comments (RFC). These RFCs are mostly written in natural language and can be hundreds of pages long. For instance, the Real Time Streaming Protocol (RTSP) 1.0 protocol is published as RFC 2326 and is 92 pages long.[1] As internet-facing software components, protocol implementations are security-critical. Security flaws in protocol implementations have often been exploited to achieve remote code execution (RCE).

A *protocol* specifies the general structure and order of the messages to be exchanged. An example of the *structure* of an RTSP message is shown in Figure 1: Apart from a header specifying message type (PLAY), address, and protocol version, the message consists of key-value pairs (**key: value**) separated by carriage return and line feed characters (CRLF; `\r\n`). The required *order* of RTSP messages is shown in Figure 2: Starting from the INIT state, only a message of type SETUP or ANNOUNCE would lead to a new state (READY). To reach the PLAY state from the INIT state, at least two messages of specific types and structures are required.

A *protocol fuzzer* automatically generates message sequences that ideally follow the required structure and order of that protocol. We can distinguish two types of protocol fuzzers. A *generator-based protocol fuzzer* [6], [7], [8] is given machine-readable information about the protocol to generate random message sequences from scratch. However, a protocol implementation itself, the manually written generator often only covers a small portion of the protocol specification, and its implementation is tedious and error-prone [1].

[1]RFC 2326 (RTSP): https://datatracker.ietf.org/doc/html/rfc2326.

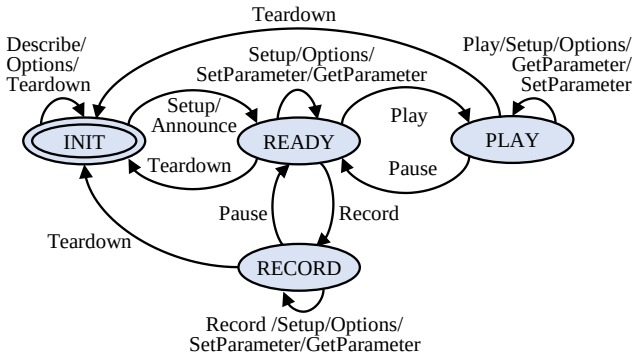


Fig. 2. The state machine for the RTSP protocol from RFC 2326.

A *mutation-based protocol fuzzer* [1], [2] uses a set of pre-recorded message sequences as seed inputs for mutation. The recording ensures that the message structure and order are valid while mutational fuzzing will slightly corrupt both [1]. In fact, all recently proposed protocol fuzzers, such as AFLNET [1] and NSFUZZ [2] follow this approach.

Challenges. However, as a state-of-the-art (SOTA) approach, mutation-based protocol fuzzing still faces several challenges:

- **(C1) Dependence on initial seeds**. The effectiveness of mutation-based protocol fuzzers is severely limited by the provided initial seed inputs. The pre-recorded message sequences will hardly cover the great diversity of protocol states and input structures as discussed in the protocol specification.
- **(C2) Unknown message structure**. Without machine-readable information about the message structure, the fuzzer cannot make structurally interesting changes to the seed messages, e.g., to construct messages of unseen types or to remove, substitute, or add an entire, coherent data structure to a seed message.
- **(C3) Unknown state space**. Without machine-readable information about the state space, the fuzzer cannot identify the current state or be directed to explore previously unseen states.

B. Large Language Models

Emerging pre-trained Large Language Models (LLMs) have demonstrated impressive performance on natural language tasks, such as text generation [9], [10], [11] and conversations [12], [13]. LLMs have also been proven effective in translating natural language specifications and instructions into executable code [14], [15], [16]. These models have been trained on extensive corpora and possess the ability to execute specific tasks without the need for additional training or hard coding [17]. They are invoked and controlled simply by providing a natural language *prompt*. The degree to which LLMs understand the tasks depends largely on the prompts provided by users.

The capabilities of LLMs have various implications for network protocols. Network protocols are implemented in accordance with the RFCs, which are written in natural language and available online. Since LLMs are pre-trained on billions of internet samples, they should be capable of understanding RFCs as well. Additionally, LLMs have already demonstrated strong text-generation capabilities. Considering messages are in text format to be transmitted between servers and clients, generating messages for LLMs should be straightforward. These capabilities of LLMs have the potential to address the open challenges of mutation-based protocol fuzzing. Moreover, the inherently automatic and easy-to-use attributes of LLMs align harmoniously with the design concept of fuzzing.

Motivation. In this paper, we propose to use LLMs to guide the protocol fuzzing. To alleviate the dependence on initial seeds (**C1**), we propose to ask the LLM to add a random message to a given seed message sequence. But does this really increase the diversity and the validity of the messages? To combat the unknown structure of messages (**C2**), we propose to ask the LLM to provide machine-readable information about the message structure (*i.e.,* the *grammar*) for every message type. But how good are those grammars compared to the ground truth and which message types are covered? To navigate the unknown state space (**C3**), we propose to ask the LLM, given the recent message exchange between fuzzer and protocol implementation, to return a message that would lead to a new state. But does this really help us transition to a new state? We will investigate these questions carefully within the following case study.

III. Case Study: Testing the Capabilities of LLMs for Protocol Fuzzing

In our study, we selected the Real Time Streaming Protocol (RTSP), along with its implementation LIVE555[2] from PROFUZZBENCH [5]. RTSP is an application-level protocol for control over the delivery of data with real-time properties. LIVE555 implements RTSP in accordance with RFC 2326, functioning as a streaming server in entertainment and communications systems to manage streaming media servers. It is included in PROFUZZBENCH, a widely-used benchmark for stateful fuzzers of network protocols [1], [3], [2]. PROFUZZBENCH comprises a suite of representative open-source network servers for popular protocols, with LIVE555 being among them. Therefore, the study results on LIVE555 would be a strong indication of whether LLMs can effectively guide protocol fuzzing. Our study was carried out in the state-of-the-art ChatGPT model [3]. In this section, we mainly demonstrate the capabilities of LLMs. Our approach and the corresponding prompts will be discussed more precisely in Section IV.

A. Lifting Message Grammars: Quality and Diversity

We ask the LLM to provide machine-readable information about the message structure (*i.e.,* the grammar), and we evaluate the quality of the generated grammars and the diversity of message types covered *w.r.t.* the ground truth. To establish the *ground-truth* grammar, two authors spent a total of 8 hours in reading the RFC 2326, and manually

[2]LIVE555 available at http://www.live555.com/

[3]Available at https://platform.openai.com/docs/models/gpt-3-5

```
PLAY <Value> RTSP/1.0 \r\n
CSeq: <Value>\r\n
User-Agent: <Value>\r\n
Session: <Value>\r\n
Range: <Value>\r\n
\r\n
```

Fig. 3. Grammar for the RTSP PLAY client request.

and individually extracting the corresponding grammar with the perfect agreement. We finally extracted the ground-truth grammar for 10 types of client requests specific to the RTSP protocol, each consisting of about 2 to 5 header fields. Figure 3 shows the PLAY message grammar, corresponding to the grammar of the PLAY client request shown in Figure 1. The PLAY grammar includes 4 essential header fields: *CSeq*, *User-Agent*, *Session*, and *Range*. Additionally, certain request types have specific header fields. For example, *Transport* is specific to SETUP requests, *Session* applies to all types except SETUP and OPTIONS, and *Range* is specific to PLAY, PAUSE, and RECORD requests.

To obtain the LLM grammar for analysis, we randomly sampled 50 answers from the LLM for the RTSP protocol and consolidated them into one answer set. [4] As shown in Figure 4, the LLM generates grammars for all ten message types that we expected to see appear in over 40 answers from the LLM. Additionally, the LLM occasionally generated 2 random types of client requests, such as "SET_DESCRIPTION"; however, each random type only appeared once in our answer set.

Furthermore, we examined the quality of the LLM-generated grammar. For 9 out of the 10 message types, the LLM produced a grammar that is *identical* to the ground-truth grammar extracted from RFC for all answers. The only exception was the PLAY client request, where the LLM overlooked the (optional) "*Range*" field in some answers. Upon further examination of the PLAY grammar in the entire answer set, we discovered that the LLM accurately generated the PLAY grammar, including the "*Range*" field, in 35 answers but omitted it in 15 answers. These findings demonstrate the LLM's ability to generate highly accurate message grammar, which motivates us to leverage grammar to guide mutation.

> The LLM generates machine-readable information for the structures of all types of RTSP client requests that match the ground truth, although there is some stochasticity.

B. Enriching the Seed Corpus: Diversity and Validity

We ask the LLM to add a random message to a given seed message sequence and evaluate the diversity and validity of the message sequences. In ProFuzzBench, the initial seed corpus of Live555 comprises only 4 types of client requests out of 10 present in the ground truth: DESCRIBE, SETUP, PLAY, and TEARDOWN. The absence of the remaining 6 types of client requests leaves a significant portion of the

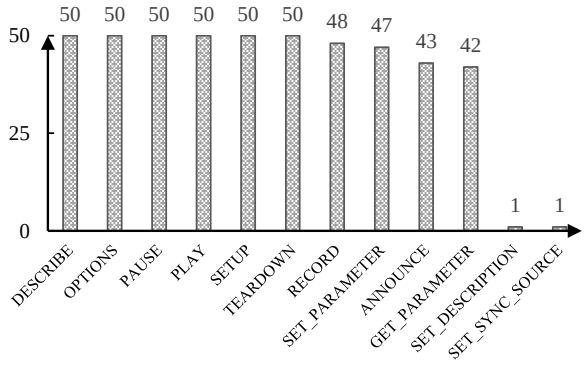


Fig. 4. Types of client requests in the answer set and the corresponding occurrence times for each type.

RTSP state machine unexplored, as shown in Figure 2. While it is possible for the fuzzers to generate the missing six types of client requests, the likelihood is relatively low. To validate this observation, we examined seeds generated by state-of-the-art fuzzers AFLNet and NSfuzz, and none of these missing message types were generated. Therefore, it is crucial to enhance the initial seeds. Can we use the LLM to generate client requests and augment the initial seed corpus?

It would be optimal if the LLM could not only generate accurate message contents but also insert the messages into the appropriate locations of the client-request sequence. It is known that the servers of network protocols are typically stateful reactive systems. This feature determines that for a client request to be accepted by servers, it must satisfy two mandatory conditions: (1) it appears in the appropriate states, and (2) the message contents are accurate.

To investigate this capability of the LLM, we requested it to generate 10 messages for each of the 10 types of client requests, resulting in a total of 100 client requests.[5] Subsequently, we verified whether the client requests were placed in the appropriate locations within a given client-request sequence. For this purpose, we compared them against the RTSP state machine shown in Figure 2, because the message sequences should transit based on the state machine. Once we ensured that a sequence of client requests was accurate based on the state machine, we sent it to the Live555 server. By examining the response code from the server, we could determine if the message content was accurate, thereby double-checking the message order as well.

Our study results demonstrate that LLM is capable of generating accurate messages and enriching the initial seeds. 99% of the collected client requests were placed in the accurate positions. The only exception is that a "DESCRIBE" client request was inserted after the "SETUP" client requests. As only one exception appeared, we consider the LLM performance to be acceptable. We sent the client-request sequences to the Live555 server and the processed results were shown in Table I. Approximately 55% of client requests can be directly

[4] We discuss the prompt engineering in Section IV-A.

[5] We discuss the detailed model prompt in Section IV-B.

Table I. Processed results of client requests after being sent to the server.

Status	Accepted	Unsupported	Session-Mismatch
Ratio	55.1%	20.4%	24.5%

accepted by the server with the successful response code "2xx". However, unsuccessful cases are not due to lacking capability of the LLM. In the unsuccessful set, 20.4% of the messages happened because LIVE555 does not support the functionality for "ANNOUNCE" and "RECORD", despite being included in its RFC. The remaining cases were attributed to incorrect session IDs in the "PLAY", "TEARDOWN", "GET_PARAMETER" and "SET_PARAMETER" requests. A session ID is dynamically assigned by the server and included in the server response. Since the LLM lacks this context information, it is not able to generate a correct session ID. However, when we replaced the session ID with the correct one, all of these messages were accepted by the server.

For our approach, we developed two methods to improve the LLM's capability of incorporating correct session IDs when provided with additional context information. We first included the server's responses in the prompt and then requested the LLM to generate the same types of messages. At this time, the generated client requests were directly accepted by the server. Furthermore, we attempted to include the session IDs into the given client-request sequence, and then the LLM also accurately inserted the same values into these messages and produced correct results.

> The LLM is able to generate accurate messages and has the capability to enrich the initial seeds.

C. Inducing Interesting State Transitions

We give the LLM the message exchange between fuzzer and the protocol implementation and ask it to return a message that would lead to a new state. We evaluate how likely the message induce a transition to a new state. Specifically, we provide the LLM with existing communication history, enabling a server respectively to reach each state (*i.e.,* INIT, READY, PLAY, and RECORD). Afterward, we query the LLM to determine the next client requests that can affect the server's state. To mitigate the influence of the LLM's stochastic behavior, we prompted the LLM 100 times for each state.

Figure 5 shows the results. Each pie chart demonstrates the results for each state. Each segment in each pie chart represents a distinct type of client request. The gray portion represents the percentage of client-request types that can lead to state change. The orange ones represent the message types that appear in the appropriate states but do not trigger any state transition (so there is no state change). The blue ones represent the types that appear in the inappropriate state that would be directly rejected by the server. From Figure 5, we can see that there are 81%, 74%, 89%, and 69% client requests, respectively, that can induce state transitions to different states. Additionally,

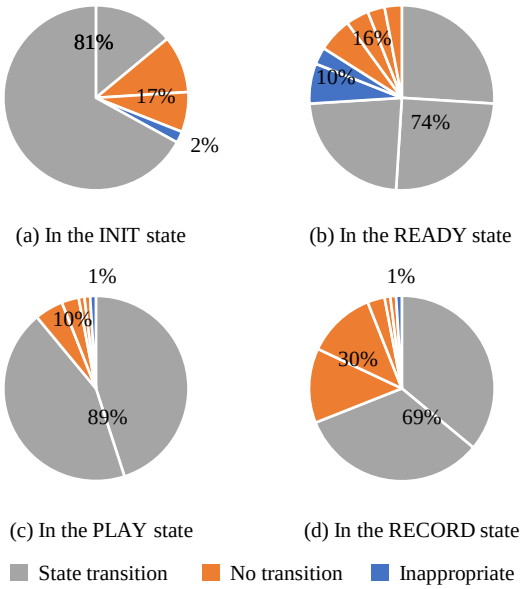


Fig. 5. The next types of client requests generated by the LLM in each state. The types in gray induce state transitions, the ones in orange appear in the suitable state but do not trigger state transitions, and the ones in blue appear in the inappropriate states. Each segment represents one distinct message type.

approximately 17%, 16%, 10%, and 30% client requests can still be accepted and processed by the server although they do not trigger the state change. These messages are still potentially useful to cover more code branches although they failed to cover more states. Besides, there is also a small percentage of inappropriate messages, which account for about 2%, 10%, 1%, and 1% in our case study. These results demonstrate that the LLMs have the capability to infer the protocol states albeit with extremely occasional mistakes.

Moreover, the generated types of client requests exhibit diversity. The LLM successfully generated client requests that encompass all state transitions for each individual state. Besides, the LLM also generated 2 to 5 appropriate types of client requests. These results further demonstrate the potential of the LLM to guide fuzzing, enabling it to surpass the coverage plateau and explore a wide range of state transitions.

> Of the LLM-generated client requests, 69% to 89% induced a transition to a different state, covering all state transitions for each individual state.

IV. LLM-GUIDED PROTOCOL FUZZING

Motivated by the impressive capabilities demonstrated by the LLMs in the case study (Section III), we develop LLM-guided protocol fuzzing (LLMPF) to tackle the challenges of existing mutation-based protocol fuzzing (EMPF).

Algorithm 1 (without the gray-shaded text) specifies the general procedure of the classical EMPF approach. The *input* is the protocol server under test P_0, the corresponding protocol p, the initial seed corpus C, and the total fuzzing time T. The *output* consists of the final seed corpus C and the seeds $C_\times$

Algorithm 1: LLM-guided Protocol Fuzzing

Input: P_0: protocol implementation
Input: p: protocol name
Input: C: initial seed corpus
Input: T: total fuzzing time
Output: C: final seed queue
Output: $C_\times$: crashing seeds

1 $P_f \leftarrow$ INSTRUMENT (P_0)
2 *Grammar* $G \leftarrow$ CHATGRAMMAR (p)
3 $C \leftarrow C \cup$ ENRICHCORPUS (C, p)
4 *PlateauLen* $\leftarrow 0$
5 StateMachine $S \leftarrow \emptyset$
6 **repeat**
7 State $s \leftarrow$ CHOOSESTATE (S)
8 Messages M, response $R \leftarrow$ CHOOSESEQUENCE (C, s)
9 $\langle M_1, M_2, M_3 \rangle \leftarrow M$ *(i.e., split M in subsequences, s.t. M_1 is the message sequence to drive P_f to arrive at state s, and message M_2 is selected to be mutated).*
10 **for** i from 1 to ASSIGNENERGY (M) **do**
11 **if** *PlateauLen* $<$ *MaxPlateau* **then**
12 **if** UNIFORMRANDOM () $< \epsilon$ **then**
13 $M_2' \leftarrow$ GRAMMARMUTATE (M_2, G)
14 $M' \leftarrow \langle M_1, M_2', M_3 \rangle$
15 **else**
16 $M' \leftarrow \langle M_1,$ RANDMUTATE (M_2), $M_3 \rangle$
17 **end**
18 **else**
19 $M_2' \leftarrow$ CHATNEXTMESSAGE (M_1, R)
20 $M' \leftarrow \langle M_1, M_2', M_3 \rangle$
21 *PlateauLen* $\leftarrow 0$
22 **end**
23 $R' \leftarrow$ SENDTOSERVER (P_f, M')
24 **if** ISCRASHES (M', P_f) **then**
25 $C_\times \leftarrow C_\times \cup \{M'\}$
26 *PlateauLen* $\leftarrow 0$
27 **else if** ISINTERESTING (M', P_f, S) **then**
28 $C \leftarrow C \cup \{(M', R')\}$
29 $S \leftarrow$ UPDATESTATEMACHINE (S, R')
30 *PlateauLen* $\leftarrow 0$
31 **else**
32 *PlateauLen* $\leftarrow$ *PlateauLen* + 1
33 **end**
34 **end**
35 **until** timeout T reached or abort-signal

that crash the server. In each fuzzing iteration (lines 7-34), EMPF selects a progressive state *s* (line 7), and the sequence *M* (line 8) that exercises *s* to steer the fuzzer in exploring the larger space. To ensure that the selected state *s* is exercised, *M* is split into three parts (line 9): M_1, the sequence to reach *s*; M_2, the portion selected for mutation; and M_3 is the remaining subsequence. Subsequently, EMPF assigns the energy for *M* (line 10) to determine mutated times and then mutates it into M' with (structure-unaware) mutators (line 16). This mutated sequence is then sent to the server (line 23). EMPF saves M' that lead to crashes (lines 24-25) or increase code or state coverage (lines 27-28). If the latter, it updates the state machine (line 29). This process is repeated until the assigned energy runs out (line 10), at which point the next state is selected.

For our LLMPF approach, we augment the baseline logic of EMPF by incorporating the grayed components: (1) Extract the grammar by prompting the LLM (line 2) and utilize the grammar to guide the fuzzing mutation (lines 12-14) (Section IV-A); (2) query the LLM to enrich the initial seeds (line 3) (Section IV-B); and (3) leverage the LLM's capability to break out of a coverage plateau (lines 4, 19-21, 26, 30 and 32) (Section IV-C). Now we will introduce each component.

A. Grammar-guided Mutation

In this section, we will introduce the approach to extracting grammar from the LLM and then leveraging the grammar to guide the structure-aware mutation.

1) Grammar Extraction: Before the fuzzer can ask the LLM to generate a grammar for structure-aware mutation [18], we encountered one immediate challenge: How to obtain a machine-readable grammar for the fuzzer? The fuzzer operates on a single machine and is restricted to parsing a predetermined format. Unfortunately, the responses generated by the LLM typically are in a natural language structure with considerable flexibility. If the fuzzer is to understand the LLM's responses, the LLM should consistently answer queries from our fuzzer in a predetermined format. An alternative option would involve manually converting the LLM's responses to the desired format. However, this approach would compromise the fuzzer's highly automated nature, which is less desirable. Therefore, the issue at hand is how to make the LLM answer questions in the desired format.

One common paradigm involves fine-tuning models to achieve proficiency in a specific task [19]. Similarly, when it comes to the LLM, fine-tuning the prompt becomes necessary. This is because the LLM can perform specific tasks by simply providing natural language prompts, without the need for additional training or hard coding. Hence, the fuzzer prompts the LLM to generate the message grammar of the protocol under test. However, the scope for prompt fine-tuning is extensive.

To make the LLM generate a machine-readable grammar, we ultimately employ *in-context few-shot learning* [9], [20] within the domain of prompt engineering. With the increasing understanding of LLMs, many prompt engineering approaches have been proposed [9], [21], [22]. In-context learning serves as an effective approach to fine-tuning the model. Few-shot learning is utilized to enhance the context with a few examples of desired inputs and outputs. This enables the LLM to recognize the input prompt syntax and output patterns. With in-context few-shot learning, we prompt the LLM with a few examples to extract the protocol grammar in the desired format.

Figure 6 illustrates the model prompt used to extract the RTSP grammar. In this prompt, the fuzzer provides two grammar examples from two different protocols in the desired format. In this format, we retain the message keywords in the grammar, which we consider to be immutable, and replace the mutable regions with the "⟨Value⟩". Notice that, to guide the

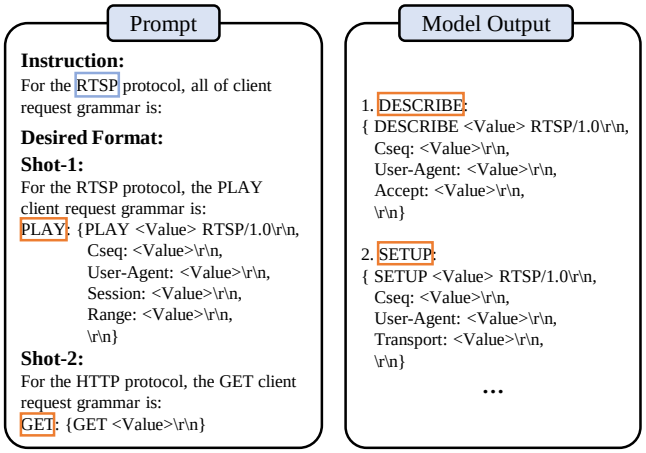


Fig. 6. Example of the model prompt and the responding response for extracting the RTSP grammar.

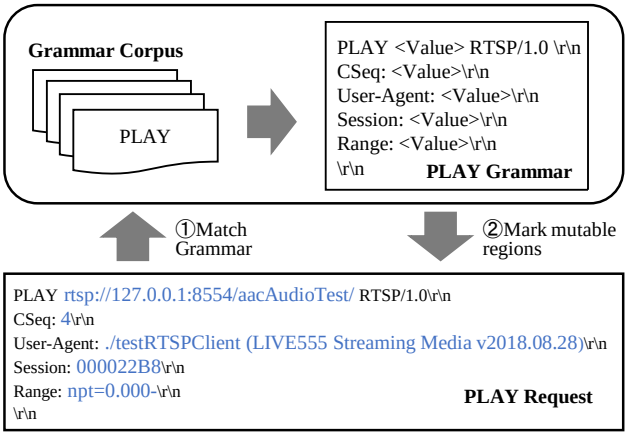


Fig. 7. Workflow of the grammar-based mutation using the PLAY request of the RTSP protocol as the example.

LLM in properly generating grammar, we utilize two shots instead of relying on a single example. This helps prevent the LLM from strictly adhering to the given grammar and potentially overlooking important facts.

In addition, another issue was revealed in our case study: the LLM may occasionally generate stochastic answers, such as "SET_DESCRIPTION". Fortunately, these instances are rare. To address the stochastic nature of the minority-sampled generation, we engage in multiple conversations with the LLM and consider the majority of consistent answers as the final grammar. This approach shares similarities with *self-consistency checks* [21] in the domain of prompt engineering, but it does not occur in chain-of-thought prompting.

Through these approaches, the fuzzer is able to effectively obtain accurate grammar from the LLM across various protocols. The model output shown in Figure 6 demonstrates a portion of the RTSP grammar derived from the LLM. In practice, the LLMs are occasionally not sensitive to the word "all" in this prompt, resulting in them generating only part of grammar types. To resolve this issue, we just simply prompt the LLMs again to ask about the remaining grammar.

Before commencing the fuzzing campaign (see line 2 of Algorithm 1 in the overview), our LLMPF approach engages in a conversation with the LLM to obtain the grammar. Subsequently, this grammar is saved into the grammar corpus G, which is utilized for structure-aware mutation throughout the entire campaign. This design is intended to minimize the overhead of interacting with the LLM while ensuring optimal fuzzing performance. Following that, we elaborate on the approach to provide guidance for structure-aware fuzzing based on the extracted grammar.

2) Mutation based on Grammar: Using the grammar corpus extracted from the LLM, LLMPF conducts structure-aware mutations of the seed message sequences. In previous work [23], researchers employed the LLM to generate variants of given inputs by tapping into their ability to comprehend input grammar. However, the limitation posed by the conversation overhead restricts the frequency of interactions with the LLM. In our approach, we adopt a different strategy. LLMPF utilizes the extracted grammar to guide the mutations. The fuzzer extracts the grammar just once, enabling it to incorporate the grammar throughout the entirety of the fuzzing campaign. We leave opportunities to escape the coverage plateau in Section IV-C. Here, we proceed to introduce the workflow of mutation based on the extracted grammar.

In line 9 of Algorithm 1, the fuzzer chooses the message portion M_2 for mutation as part of the algorithm design. Let us assume M_2 consists of multiple client requests, one of which is the PLAY client request of the RTSP protocol. Our mutation approach guided by grammar is illustrated in Figure 7. It shows the workflow for mutating one single RTSP PLAY client request. Specifically, when presented with the PLAY client request, LLMPF first matches it with the corresponding grammar. To expedite the matching process, we maintain the grammar corpus in the *map* format: $G = \{type \rightarrow grammar\}$. Here, *type* represents the types of client requests. LLMPF uses the first line of each grammar as the label for message types. The *grammar* corresponds to the concrete message grammar. Using the message type, LLMPF retrieves the corresponding grammar. Subsequently, we employ regular expressions (Regex) to match each header field in the message with the grammar, marking regions as mutable falling under "⟨Value⟩". In Figure 7, these mutable regions identified are highlighted in blue. During mutation, LLMPF only selects these regions, ensuring the messages retain valid formats. However, if no grammar match is found, we consider all regions mutable.

To preserve the fuzzer's capability of exploring some corner cases, we continue to employ the structure-*un*aware mutation approach from the classical EMPF, as demonstrated in line 16 of Algorithm 1. Nonetheless, LLMPF conducts structure-aware mutations with a higher likelihood, considering that valid messages hold a greater potential for exploring a larger state space.

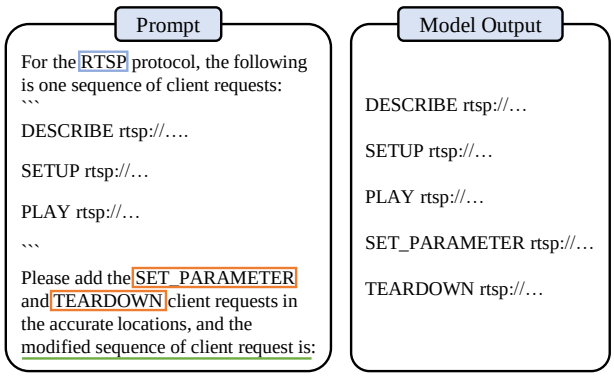


Fig. 8. Example of the model prompt and the responding response for enriching initial seed corpus (we omit the details of messages).

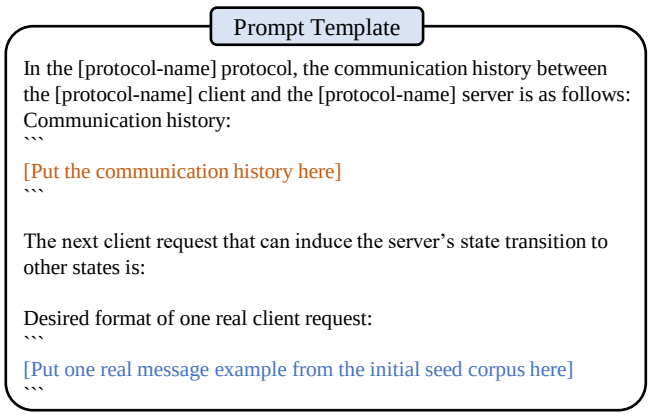


Fig. 9. The prompt template for obtaining the next client request that can induce the server's state transition to other states.

B. Enriching Initial Seeds

Motivated by the ability of the LLM to generate new messages and insert them into the appropriate positions within the provided message sequence (cf. Section III-B), we propose to enrich the initial seed corpus used for fuzzing (line 3 of Algorithm 1). However, there are several *challenges* that our approach must first tackle: (i) How to generate new messages that carry the correct context information (*e.g.,* the correct session ID in the RTSP protocol)? (ii) How to maximize the diversity of the generated sequences? (iii) How to prompt the LLM to generate the entire modified message sequence from the given seed message sequence?

As for Challenge (i), we found that the LLM can automatically learn the required context information from the provided message sequence. For instance, for our experiments, ProFuzzBench already possesses some message sequences as initial seeds (although they lack diversity). The initial seeds of ProFuzzBench are constructed by capturing the network traffic between the tested servers and the clients. Thereby, these initial seeds contain correct and sufficient context information from the servers. Hence, when prompting the LLM, we include the initial seeds from ProFuzzBench to facilitate the acquisition of the necessary context information.

As for Challenge (ii), the fuzzer determines which types of client requests are missing in the initial seeds, *i.e.,* what types of messages should be generated by the LLM to enrich the initial seeds. In Section IV-A, we have obtained the grammar for all types of client requests; thus, identifying the missing types in initial seeds is not a difficult issue. Let us revisit the grammar prompt shown in Figure 6. The prompt includes the names of message types (*i.e.,* PLAY and GET), and correspondingly, the message names are also included in the model output (*e.g.,* DESCRIBE and SETUP). We utilize this information to maintain a set of message types: $AllTypes = \{messageType\}$, and one map from grammars to the corresponding type: $G2T = \{grammar \rightarrow type\}$.

While detecting the missing message types, we first utilize the grammar corpus G obtained in Section IV-A and the grammar-to-type map $G2T$ to obtain existing message types and maintain them into a set (*i.e., ExistingTypes*). Consequently, the missing message types are in the complement: *MissingTypes* = (*AllTypes* - *ExistingTypes*). We then instruct the LLM to generate the missing types of messages and insert them into the initial seeds; thereby, our approach is based on existing initial seeds but enriches them. To avoid excessively long initial seeds, we evenly select and add two missing types at a time in a given message sequence. This allows us to control the length and diversity of the initial messages.

As for Challenge (iii), to ensure the validity of the generated message sequence, we design our prompt in the continuation format (*i.e.,* "the modified sequence of client requests is:"). In practice, the obtained responses can be directly utilized as the seeds, with the exception of removing the newline character (`\n`) at the beginning or adding any missing delimiters (`\r\n`) at the end. An illustrative example is presented in Figure 8. In this case, we instruct the LLM to insert two types of messages, "SET_PARAMETER" and "TEARDOWN", into the given sequence. The modified sequence is shown on the right.

C. Surpassing Coverage Plateau

Exploring unseen states poses a challenge for stateful fuzzers. To better understand this challenge, let us revisit the RTSP state machine illustrated in Figure 2. Assume the server is currently in the READY state after accepting a sequence of client requests. If the server intends to transition to different states (*e.g.,* the PLAY or RECORD state), the client must send corresponding PLAY or RECORD requests. In the context of the fuzzing design, the fuzzer assumes the role of the client. While the fuzzer possesses the capability to generate messages that induce state transitions, it requires the exploration of a considerable number of seeds. There is a high likelihood that the fuzzer may fail to generate suitable message orders to cover the desired state transitions [1], [3]. Consequently, a substantial portion of the code space remains unexplored. Therefore, it is important to explore additional states in order to thoroughly test stateful servers. Unfortunately, accomplishing this task proves challenging for existing stateful fuzzers.

In this paper, when the fuzzer becomes unable to explore new coverage, we refer to this scenario as the fuzzer entering a coverage plateau. Motivated by the study results in Section III-C, we utilize the LLM to assist the fuzzer in surpassing the coverage plateau. This occurs when the fuzzer is unable to generate interesting seeds within a given time period. We quantify this duration based on the number of uninteresting seeds continuously generated by the fuzzer. Specifically, throughout the fuzzing campaign, we maintain a global variable called *PlateauLen* to keep track of the number of uninteresting seeds continuously observed thus far. Before commencing the fuzzing campaign, *PlateauLen* is initialized to 0 (Line 4 of Algorithm 1). During each fuzzing iteration, *PlateauLen* is reset to 0 if we encounter a seed that crashes the program (line 26) or when the coverage increases (line 30). Otherwise, if the seed is deemed uninteresting, *PlateauLen* is incremented by 1 (line 32).

Based on the value of *PlateauLen*, we determine whether the fuzzer has entered the coverage plateau. If *PlateauLen* does not exceed *MaxPlateau*, the predefined maximum length of the coverage plateau (line 11), our LLMPF mutates messages using the strategy introduced earlier. The value of *MaxPlateau* is specified by users and provided to the fuzzer. However, when *PlateauLen* surpasses *MaxPlateau*, we consider the fuzzer to have entered the coverage plateau. In such case, LLMPF will utilize the LLM to overcome the coverage plateau (lines 19-21). To achieve this, we employ the LLM to generate the next suitable client requests that may induce state transitions to other states. The prompt template is shown in Figure 9. We provide the LLM with the communication history between servers and clients; *i.e.,* the client requests and the corresponding server responses. To ensure that the LLM generates an authentic message rather than message types or descriptions, we demonstrate the desired format by extracting any message from the initial seed corpus. Subsequently, the LLM infers the current states and generate the next client request M_2'. This request acts as a mutation of the original M_2 and is inserted into the message sequence M', which is then sent to the server.

Let us reconsider the RTSP example. Initially, the server is in the INIT state. Upon receiving the message sequence M_1 = {SETUP}, it responds with R_1 = {200-OK}, transitioning to the READY state. Subsequently, the fuzzer encounters a coverage plateau, where it fails to generate interesting seeds. Upon noticing this, we stimulate the LLM by presenting the communication history H = {SETUP, 200-OK}. In response, the LLM is highly likely to reply a PLAY or RECORD message, as indicated by the study results in Section III-C. These messages lead the server to transition to a different state, overcoming the coverage plateau.

D. Implementation

We implemented this LLM-guided protocol fuzzing (*cf.* Algorithm 1) into AFLNET [1], called CHATAFL, to test protocols written in C/C++. AFLNET is one of the most popular mutation-based open-source protocol fuzzers[6]. It maintains an inferred state machine and uses state and code feedback to guide the fuzzing campaign. The identification of the current state involves parsing the response codes from servers' response messages. A seed is considered interesting if it increases state or code coverage. CHATAFL continues to utilize this approach while seamlessly integrating the three aforementioned strategies into the AFLNET framework.

V. EXPERIMENTAL DESIGN

To evaluate the utility of Large Language Models (LLMs) for tackling the challenges of mutation-based protocol fuzzing of text-based network protocols, we seek to answer the following questions:

- **RQ.1** **State coverage.** How much more state coverage does CHATAFL achieve compared to baseline?
- **RQ.2** **Code coverage.** How much more code coverage does CHATAFL achieve compared to baseline?
- **RQ.3** **Ablation.** What is the impact of each component on the performance of CHATAFL?
- **RQ.4** **New bugs.** Is CHATAFL useful in discovering previously unknown bugs in widely-used and extensively-tested protocol implementations?

To answer these questions, we follow the recommended experimental design for fuzzing experiments [24], [25].

A. Configuration Parameters

In order to decide saturation, we set the maximum length of the coverage plateau (*MaxPlateau*) to 512 non-coverage-increasing message sequences. This value was determined through a heuristic screening approach. In preliminary experiments, we found 512 to be a reasonable setting for *MaxPlateau*, achieved within approximately 10 minutes. Setting the value too small would cause CHATAFL to overly query the LLM, while setting it too large would lead CHATAFL to remain stuck for too long instead of benefiting from our optimization (*cf.* Section IV-C). Once the coverage plateau is reached, CHATAFL prompts the LLM to generate message sequences that surpass the coverage plateau (Section IV-C). To limit the cost of LLM prompts, we set a quarter of *MaxPlateau* as the maximum number of ineffective prompts.

As a large language model (LLM), we used the *gpt-3.5-turbo* model. In accordance with the recommendation to employ a low temperature for precise and factual responses [26], [21], a temperature of 0.5 was used to extract the grammar and enrich the initial seeds (*cf.* Section IV-A & Section IV-B). To generate new messages, J. Qiang *et al.* [23] found for greybox fuzzing, a temperature of 1.5 is optimal. Hence, we set a temperature of 1.5 to break out of the coverage plateau (*cf.* Section IV-C). When extracting the grammar, for the *self-consistency check* [21], we use five repetitions. As confirmed in our case study (*cf.* Section III-A), we found five repetitions sufficient to filter out incorrect cases.

[6]Available at https://github.com/aflnet/aflnet; 689 stars at the time of writing.

Table II. Detailed information about our subject programs.

Subject	Protocol	#LOC	#Stars	Version
Live555	RTSP	57k	631	31284aa
ProFTPD	FTP	242k	445	61e621e
PureFTPD	FTP	29k	572	10122d9
Kamailio	SIP	939k	1,915	a220901
Exim	SMTP	118k	662	d6a5a05
forked-daapd	DAAP	79k	1,718	2ca10d9

B. Benchmark and Baselines

Table II presents the subject programs that are used in our evaluation. Our benchmark consists of six text-based network protocol implementations, including five widely-used network protocols (*i.e.,* RTSP, FTP, SIP, SMTP, and DAAP). These subject programs cover all text-based network protocols in PROFUZZBENCH, a widely-used benchmark for evaluating stateful protocol fuzzers [1], [4], [2], [27]. The protocols cover a wide range of applications, including streaming, messaging, and file transfer. The implementations are mature and widely used both in enterprises and by individual users. For each protocol, we selected implementations that are popular and suitable for use in real-world applications. Security flaws in these projects can have wide-reaching consequences.

As baseline tools, we selected AFLNET and NSFUZZ-v. Since our tool CHATAFL has been implemented into AFLNET, every observed difference between CHATAFL and AFLNET can be attributed to our changes to implement LLM guidance. AFLNET [1] is a popular open-source, state-of-the-art, mutation-based, code- and state-guided protocol fuzzer. NSFUZZ-v [2] extends AFLNET to get a better handle on the protocol state space. It identifies state variables through static analysis and uses state variable values as fuzzer feedback to maximize the coverage of the state space. The underlying idea is very similar to that of SGFUZZ [3] which was published around the same time but implemented into LibFuzzer [28]. SGFUZZ also uses the sequence of state variable values to implicitly capture the coverage of the protocol state space. Other protocol fuzzers, like STATEAFL [4] and BooFuzz [7] have previously been (unfavourably) compared to AFLNET or NSFUZZ-v, *i.e.,* the tools that we use as baselines.

C. Variables and Measures

In order to evaluate the effectiveness of CHATAFL versus the baseline fuzzers, we measure how well the protocol fuzzers cover the state space of the protocol and the code of the protocol implementation. The key idea is that a protocol fuzzer cannot find bugs in uncovered code or states. However, coverage is only a proxy measure for the bug-finding ability of a fuzzer [24], [25]. Hence, we complement the coverage results with bug-finding results.

Coverage. We report the coverage of both, the code and the state space. To evaluate *code coverage*, we measure the branch coverage achieved using the automated tooling provided by the benchmarking platform PROFUZZBENCH [5]. To evaluate the coverage of the state space, we measure (i) the number of distinct states (*state coverage*) and the number of transitions between these states (*transition coverage*) using automatic tooling provided by the benchmarking platform. Like the authors of AFLNET and PROFUZZBENCH, in the absence of ground truth state machines for the tested protocols, we define distinct states traversed by a message sequence the set of unique response codes that are returned by the server. To mitigate the impact of randomness, we report the average coverage achieved across 10 repetitions of 24 hours.

Bugs. To identify bugs, we execute the tested programs under the Address Sanitizer (ASAN). CHATAFL stores the crashing message sequences, and then we use the AFLNet-replay utility provided by AFLNET to reproduce the crashes and debug the underlying causes. We distinguish different bugs by analyzing stack traces reported by ASAN. Finally, we report these bugs to their respective developers for confirmation.

D. Experimental Infrastructure

All experiments were conducted on a machine equipped with an Intel(R) Xeon(R) Platinum 8468V CPU. This machine has 192 logical cores running at 2.70GHz. It operates on Ubuntu 20.04.2 LTS with 512GB of main memory.

VI. EXPERIMENTAL RESULTS

RQ.1 State Space Coverage

Transitions. Table III shows the average number of state transitions covered by our tool CHATAFL versus the two baselines AFLNET and NSFUZZ-v. To quantify the improvement of CHATAFL over the baselines, we report the percentage improvement in terms of transition coverage achieved in 24 hours (*Improv*), how much faster CHATAFL can achieve the same transition coverage as the baseline in 24 hours (*Speed-up*), and the probability that a random campaign of CHATAFL outperforms a random campaign of the baseline ($\hat{A}_{12}$, Vargha-Delaney measure of effect size [29]).

Compared to both baselines, CHATAFL exercised a greater number of state transitions and significantly sped up the state exploration process. On average, CHATAFL exercised 48% *more* state transitions than AFLNET. Specifically, in the LIVE555 subject, CHATAFL increased the number of state transitions by 91% compared to AFLNET. Furthermore, CHATAFL explored the same number of state transitions 48× faster than AFLNET, on average. In comparison to NSFUZZ, CHATAFL covered 43% more state transitions on average and achieved the same number of state transitions 16× faster. For all subjects, the Vargha-Delaney effect size $\hat{A}_{12} \geq 0.86$ indicates a substantial advantage of CHATAFL over both AFLNET and NSFUZZ in exploring state transitions.

States. Table IV shows the average number of states covered by our tool CHATAFL versus the two baselines AFLNET and NSFUZZ-v and the corresponding percentage improvement. Clearly, CHATAFL outperformed both AFLNET and NSFUZZ. Specifically, CHATAFL covered 30% more states than AFLNET and 26% more states than NSFUZZ, respectively. To put the number of covered states in the context of the total

Table III. Average number of state transitions for our CHATAFL and the baselines AFLNET and NSFUZZ in 10 runs of 24 hours.

Subject	CHATAFL	Transition comparison with AFLNET				Transition comparison with NSFUZZ			
		AFLNET	Improv	Speed-up	$\hat{A}_{12}$	NSFUZZ	Improv	Speed-up	$\hat{A}_{12}$
Live555	160.00	83.80	90.98%	228.62×	1.00	90.20	77.38%	63.09×	1.00
ProFTPD	246.70	172.60	42.91%	7.12×	1.00	181.20	36.11%	4.97×	1.00
PureFTPD	281.80	216.90	29.91%	5.61×	1.00	206.10	36.72%	7.94×	1.00
Kamailio	130.00	99.90	30.14%	5.53×	1.00	105.30	23.42%	4.58×	1.00
Exim	108.40	62.70	72.98%	40.27×	1.00	69.50	55.97%	13.25×	1.00
forked-daapd	25.40	21.40	18.65%	1.58×	1.00	20.10	26.52%	1.79×	0.86
AVG	-	-	47.60%	48.12×	-	-	42.69%	15.94×	-

Table IV. Average number of states and the improvement of CHATAFL compared with AFLNET and NSFUZZ.

Subject	CHATAFL	AFLNET	Improv	NSFUZZ	Improv	Total
Live555	14.20	10.00	41.75%	11.70	21.16%	15
ProFTPD	28.70	22.60	26.84%	24.30	17.81%	30
PureFTPD	27.90	25.50	9.37%	24.00	16.20%	30
Kamailio	17.00	14.00	21.43%	15.10	12.50%	23
Exim	19.50	14.10	38.19%	14.40	35.42%	23
forked-daapd	12.10	8.70	39.74%	8.00	51.39%	13
AVG	-	-	29.55%	-	25.75%	-

number of reachable states, the last column of Table IV shows the total number of states that have been covered by *any* of the three tools in *any* of the ten runs of 24 hours. We can see that the *average* fuzzing campaign of CHATAFL covers almost all reachable states. For instance, in the case of LIVE555, CHATAFL covers an average of 14.2 out of 15 states, while AFLNET and NSFUZZ only manage to cover 10 states and 11.7 states, respectively. Only for Kamailio CHATAFL covers a smaller proportion of the reachable state space (avg. 17; max. 20 of 23 states). Nevertheless, CHATAFL still outperforms the baselines in terms of state coverage.

In terms of state coverage, on average, CHATAFL covers 48% and 43% more state transitions than AFLNET and NSFUZZ, respectively. Compared to the baseline, CHATAFL covers the same number of state transitions 48 and 16 times faster, respectively. In addition, CHATAFL also explores a substantially larger proportion of the reachable state space than both AFLNET and NSFUZZ.

RQ.2 Code Coverage

Table V shows the average branch coverage achieved by CHATAFL and the baselines AFLNET and NSFUZZ across 10 fuzzing campaigns of 24 hours. To quantify the improvement of CHATAFL over the baselines, we report the percentage improvement in terms of branch coverage in 24 hours (*Improv*), how much faster CHATAFL can achieve the same branch coverage as the baseline in 24 hours (*Speed-up*), and the probability that a random campaign of CHATAFL outperforms a random campaign of the baseline ($\hat{A}_{12}$).

As we can see, for all subjects, CHATAFL covers more branches than both baselines. Specifically, CHATAFL covers 5.8% more branches than AFLNET with a range from 2.4% to 8.0%. When compared to NSFUZZ, CHATAFL covers 6.7% more branches. In addition, CHATAFL covers the same number of branches 6× faster than AFLNET and 10× faster than NSFUZZ. For all subjects, the Vargha-Delaney effect size $\hat{A}_{12} \geq 0.70$ demonstrates a substantial advantage of CHATAFL over both baselines in terms of code coverage achieved.

In terms of code coverage, on average, CHATAFL covers 5.8% and 6.7% more branches than AFLNET and NSFUZZ, respectively. In addition, CHATAFL achieves the same number of branches 6 and 10 times faster than AFLNET and NSFUZZ, respectively.

RQ.3 Ablation Studies

CHATAFL implements three strategies to interact with the LLM to overcome the challenges of protocol fuzzing:

- S_A) grammar-guided mutation,
- S_B) enriching initial seeds, and
- S_C) surpassing coverage plateau.

To evaluate the contribution of each strategy towards the increase in coverage, we conducted an ablation study. For this purpose, we developed four tools:

- CL0: AFLNET, *i.e.,* all strategies all are *disabled*,
- CL1: AFLNET plus grammar-guided mutation (S_A),
- CL2: AFLNET plus grammar-guided mutation (S_A) and enriching initial seeds (S_B), and
- CL3: AFLNET plus *all* strategies ($S_A + S_B + S_C$), *i.e.,*CL3 is CHATAFL.

Table VI shows the results in terms of branch coverage in a similar format we have used previously (Improv, Speed-up, and $\hat{A}_{12}$). *However*, compared to previous tables, crucially the results in terms of improvement, speed-up, and $\hat{A}_{12}$ effect size are shown in the *inverse* direction. For instance, for ProFTPD, the configuration CL3 (*i.e.,*CHATAFL) achieves 8% more branch coverage than the baseline configuration CL0 (*i.e.,*AFLNET). The difference in improvement between two neighboring configurations (shown in parenthesis) quantifies the effect of the strategy that is enabled. For instance, for

Table V. Average number of branches covered by our CHATAFL and the baselines AFLNET and NSFUZZ in 10 runs of 24 hours.

Subject	CHATAFL	Branch comparison with AFLNET				Branch comparison with NSFUZZ			
		AFLNET	Improv	Speed-up	$\hat{A}_{12}$	NSFUZZ	Improv	Speed-up	$\hat{A}_{12}$
Live555	2,928.40	2,860.20	2.38%	9.61×	1.00	2,807.60	4.30%	21.60×	1.00
ProFTPD	5,143.30	4,763.00	7.99%	4.04×	1.00	4,421.80	16.32%	21.96×	1.00
PureFTPD	1,134.30	1,056.30	7.39%	1.60×	0.91	1,041.10	8.96%	1.60×	1.00
Kamailio	10,064.00	9,404.10	7.02%	12.69×	1.00	9,758.70	3.13%	2.95×	1.00
Exim	3,789.40	3,647.60	3.89%	4.27×	1.00	3,564.30	6.32%	11.33×	0.77
forked-daapd	2,364.80	2,227.10	6.18%	4.63×	1.00	2,331.30	1.43%	1.66×	0.70
AVG	-	-	5.81%	6.14×	-	-	6.74%	10.18×	-

Table VI. Improvements in terms of branch coverage compared with baseline if we enable each strategy one by one.

Subject	CL0	Enable strategy S_A (CL1)			Enable strategies S_A and S_B (CL2)			Enable all strategies (CL3)		
		Improv	Speed-up	$\hat{A}_{12}$	Improv	Speed-up	$\hat{A}_{12}$	Improv	Speed-up	$\hat{A}_{12}$
Live555	2,860.20	0.28%	1.60×	0.89	1.49% (1.21*pp*)	8.45×	1.00	2.38% (0.89*pp*)	9.61×	1.00
ProFTPD	4,763.00	3.63%	2.45×	0.60	5.27% (1.64*pp*)	3.69×	0.63	7.99% (2.72*pp*)	4.04×	1.00
PureFTPD	1,056.30	6.67%	1.34×	0.61	6.70% (0.03*pp*)	1.36×	0.86	7.39% (0.69*pp*)	1.60×	0.91
Kamailio	9,404.10	0.60%	1.75×	0.96	2.24% (1.64*pp*)	8.92×	1.00	7.02% (4.78*pp*)	12.69×	1.00
Exim	3,647.60	2.36%	2.48×	0.52	2.54% (0.18*pp*)	2.36×	0.58	3.89% (1.35*pp*)	4.27×	1.00
forked-daapd	2,227.10	4.67%	2.48×	0.68	4.93% (0.26*pp*)	2.98×	1.00	6.18% (1.25*pp*)	4.63×	1.00
AVG	-	3.04%	2.02×	-	3.86% (0.82*pp*)	4.63×	-	5.81% (1.95*pp*)	6.14×	-

ProFTPD, the configuration CL2 only achieves a 5.3% improvement, which is 2.7 percentage points (pp) less than CL3, demonstrating the effectiveness of strategy S_C which was enabled from CL2 to CL3.

Overall. All the strategies contributed to the improvement of branch coverage, and none of the strategies had a negative impact on branch coverage. Specifically, CL1 resulted in an average increase of 3.04% in branch coverage compared to CL0. CL2 exhibited an average increase of 3.9%, while CL3 showed the highest average increase of 5.9% in branch coverage. Furthermore, CL1 achieved the same branch coverage 2× faster than CL0, CL2 achieved the same branch coverage with a 5× speed-up, and CL3 demonstrated a 6× faster achievement. Therefore, enabling all three strategies proved to be the most effective approach.

Strategy S_A. We evaluated the impact of strategy S_A (*i.e.,* grammar-based mutation). In ProFTPD, PureFTPD, Exim, and forked-daapd, CL1 increased the branch coverage by 2.4% to 6.7%. However, in the remaining two subjects Live555 and Kamailio, although CL1 also improved the branch coverage, it only increased by 0.28% and 0.60%, respectively. Upon investigating the implementations of these two subjects, we discovered that their implementations do not strictly adhere to the message grammar. The messages with missing or incorrect header fields can still be accepted by their servers.

Strategy S_B. When compared to CL1, which only enabled strategy S_A, we observed the contribution of strategy S_B. On average, enabling the strategy led to 0.82% more branches covered. Strategy S_B significantly increased branch coverage in Live555, ProFTPD, and Kamilio by 1.21% to 1.64%, while it only increased branch coverage by about 0.03% to 0.26% in the other three subjects. For the latter three subjects, PROFUZZBENCH included nearly all types of client requests; therefore, there is not much chance to increase seed diversity.

Strategy S_C. When comparing CL3 to CL2, we can observe that enabling strategy S_C significantly increased the branch coverage by 0.69% to 4.78%. Specifically in ProFTPD and Kamailio, strategy S_C helps increase 2.72% and 4.78% branch coverage, respectively.

> Overall, every strategy contributes to varying degrees of improvement in branch coverage. Enabling strategies S_A, S_B, and S_C one by one allows us to achieve the same branch coverage 2.0, 4.6, and 6.1 times faster, respectively.

RQ.4 Discovering New Bugs

In this experiment, we evaluate the utility of CHATAFL by checking whether it is able to discover zero-day bugs in our subject programs. For this purpose, we utilized CHATAFL on the latest versions of our subjects, running 10 repetitions over 24 hours. In the course of the experiment, CHATAFL produced promising results, as demonstrated in Table VII.

A total of nine (9) unique and previously unknown vulnerabilities were discovered by CHATAFL, despite extensive testing conducted by AFLNET and NSFUZZ. Vulnerabilities were found in three of the six tested implementations and encompass various types of memory vulnerabilities, including use-after-free, buffer overflow, and memory leaks. Moreover, these bugs have potential security implications that can result in remote code execution or memory leakage. We reported these bugs to the respective developers. Out of the 9 bugs, 7 have been confirmed by the developers, and 3 have already

Table VII. Statistics of 9 zero-day vulnerabilities discovered by CHATAFL in widely-used and extensively-tested protocol subjects.

ID	Subject	Version	Bug Description	Potential Security Issue	Status
1	Live555	2023.05.10	Heap use after free in handling PLAY client requests	Remote code execution	CVE-requested, fixed
2	Live555	2023.05.10	Heap use after free in handling SETUP client requests	Remote code execution	CVE-requested, fixed
3	Live555	2023.05.10	Use after return in handling DESCRIBE client requests	Remote code execution	CVE-requested
4	Live555	2023.05.10	Use after return in handling SETUP client requests	Remote code execution	CVE-requested
5	Live555	2023.05.10	Heap buffer overflow in handling stream	Remote code execution	CVE-requested
6	Live555	2023.05.10	Memory leaks after allocating memory for stream parameters	Memory leakage	Reported
7	Live555	2023.05.10	Heap use after free in calling RTPInterface::sendDataOverTCP	Remote code execution	CVE-requested
8	ProFTPD	61e621e	Heap buffer overflow while parsing FTP commands	Remote code execution	CVE-requested, fixed
9	Kamailio	a220901	Memory leaks after allocating memory in parsing config files	Memory leakage	Reported

been fixed by now (the time of paper submission). We have requested CVE IDs for the confirmed bugs.

We utilized AFLNET and NSFUZZ to detect these 9 vulnerabilities. Both AFLNET and NSFUZZ were configured with the same subject versions to run an equal duration (*i.e.,* 10 repetitions over 24 hours) as CHATAFL. However, AFLNET was only able to discover three of them (*i.e.,* bugs #5, #6, and #9), and NSFUZZ was able to discover four of them (*i.e.,* bugs #5, #6, #7, and #9). In addition, AFLNET and NSFUZZ did not find any additional bugs.

To understand the contributions of the LLM guidance, we conducted a more detailed investigation of Bug #1, a heap-use-after-free vulnerability. This bug occurs when the allocated memory for the usage environment of a particular track is deallocated during processing PAUSE client requests. Subsequently, this memory is overwritten upon receiving the PLAY client request, leading to a heap-use-after-free issue.

In order to trigger this bug, it is necessary to involve several types of client requests: SETUP, PLAY, and PAUSE. However, the PAUSE client requests were not included in the initial seeds used in previous works. While it is theoretically possible for fuzzers to generate such client requests, it is unlikely. We examined all the seeds generated by AFLNET and NSFUZZ in our experiments and found that none of them produced the PAUSE client requests in any of the runs. However, CHATAFL prompts the LLM to add the PAUSE client requests during the enrichment of the initial seeds (*cf.* Section IV-B).

Once the required client requests are available, triggering this bug necessitates sending specific messages to the server that cover particular states and state transitions. Specifically, these messages should cover three states as shown in Figure 2: INIT, READY, and PLAY. Additionally, several state transitions need to be covered: INIT $\rightarrow$ READY, READY $\rightarrow$ PLAY, PLAY $\rightarrow$ READY, and then READY $\rightarrow$ PLAY again. The fuzzer itself has the potential to cover these states and state transitions with diverse seeds. Additionally, the LLM can provide guidance to the fuzzer in order to cover them. For instance, during the PLAY states, the LLM can generate the next client request, PAUSE, to execute the PLAY $\rightarrow$ READY transition (*cf.* Section IV-C).

Lastly, we should not ignore the contribution of structure-aware mutation. To trigger this bug, a minimal message sequence is required: SETUP $\rightarrow$ PLAY $\rightarrow$ PAUSE $\rightarrow$ PLAY. Omitting any of these messages will render the bug untriggerable. Existing mutation-based fuzzers, with their structure-unaware mutation approach, have a high likelihood of breaking the message structures and rendering them invalid. In contrast, by utilizing the grammar derived from the LLM, structure-aware mutation efficiently maintains the validity of messages.

> CHATAFL discovered 9 distinct, previously unknown bugs while AFLNET and NSFUZZ only discovered 3 and 4 of those, respectively. AFLNET and NSFUZZ did not find any additional bugs, either. Seven of the nine bugs (7/9) are potentially security-critical.

Experience on Manual Effort

During the CHATAFL's usage, no manual effort was needed to run the experiments for all protocols shown in Table II. Specifically, when extracting grammar from the LLM, we utilize the prompt shown in Figure 6. During protocol testing, only the protocol name (*e.g.,* RTSP) in the Instruction part is changed. Under Desired Format, Shot-1 and Shot-2 serve as examples for the LLM to print the grammar in the given machine-readable structure so that CHATAFL can parse the printed grammar. We spent an hour obtaining these exemplary shots, but this setup is a one-time effort; subsequent testing of other protocols requires no additional manual effort. With the grammar obtained from the LLM, the structure-aware mutations are fully automatic (*cf.* Section IV-A).

To enrich initial seeds, we utilize the prompt template in Figure 8. The entire prompt is automatically generated from this prompt template when utilizing CHATAFL for protocol testing. The protocol name and an existing message sequence are automatically pasted into this template. In addition, the names for the message types under generation are sourced from the model output in Figure 6. In soliciting the LLM's assistance to overcome coverage plateaus, we generate the complete prompt using the template in Figure 9. Therefore, there is no manual effort needed to utilize CHATAFL.

CHATAFL is designed to test text-based protocols with publicly available RFCs. The specifications for most protocols are documented in these publicly available RFCs, which are included as training data for the LLM. However, for certain

proprietary protocols, whose RFCs are not included in the LLM training data, ChatAFL may not perform optimally when testing them.

VII. Related Work

Grammar-based fuzzing: Generation-based fuzzing generates messages from scratch based on manually constructed specifications [30], [8], [6], [7], [31], [32]. These specifications typically include a data model and a state model. The data model describes the message grammar, while the state model specifies the message order between servers and clients. However, constructing these specifications can be a laborious task and requires large human efforts. In contrast, large language models (LLMs) are pre-trained on billions of documents and possess extensive knowledge about protocol specifications. In ChatAFL, we leverage LLMs directly to obtain specification information, eliminating the need for additional manual efforts.

Dynamic Message Inference: To reduce the reliance on prior knowledge and manual work before fuzzing, several existing works have been proposed to dynamically infer message structures, including blackbox fuzzers [33], [34] and whitebox fuzzers [35], [36], [37]. Blackbox fuzzers such as TreeFuzz [34] employ machine learning techniques over the seed corpus to construct probabilistic models that are subsequently used for input generation. Whereas the whitebox fuzzers, such as Polyglot [35], extract the message structure through dynamic analysis techniques over systems under test, such as symbolic execution and taint tracking. However, these approaches can only infer message structures based on the observed messages. As a result, the inferred structure may deviate significantly from the actual message structures.

Dynamic State Inference: Mutation-based fuzzing is one of the primary categories within fuzzing protocol implementations. Mutation-based fuzzers [38], [39], [18], [28], [40], [41], [42] generate new inputs by randomly mutating existing seeds selected from a corpus of seed inputs and utilize coverage information to systematically evolve this corpus. Guided by branch coverage feedback, they have been proven to be effective in fuzzing stateless programs. However, when fuzzing stateful programs, branch coverage alone is a useful but insufficient metric for guiding the fuzzing campaign as elucidated in existing works [43]. Therefore, state coverage feedback is employed to work with branch coverage to guide the fuzzing campaign. However, identifying states presents a significant challenge. A series of works [3], [1], [4], [2] proposes various state representation schemes. AFLNet [1] utilizes the response code as states, constructs a state machine during the fuzzing campaign, and employs it as state-coverage guidance. StateAFL [4], SGFuzz [3], and NSFuzz [2] propose distinct state representation schemes based on program variables. In this paper, we do not attempt to answer what states are. Instead, we delegate this task to the LLM, allowing it to infer states. This approach has proven effective.

Fuzzing based on Large Language Models: Following the remarkable success of pre-trained large language models (LLMs) in various natural language processing tasks, researchers have been exploring their potential in diverse domains, including in fuzzing. For instance, CodaMosa [44] was the first to apply LLMs to fuzzing (*i.e.,* the automatic generation of test cases for Python modules). Later, TitanFuzz [45] and FuzzGPT [46] used an LLM to automatically generate test cases for Deep Learning software libraries, specifically. While these works were taking a generational approach to fuzzing, ChatFuzz [23] takes a mutational one by asking the LLM to modify human-written test cases. Ackerman *et al.* [47] leverages the ambiguity of format specifications and employs the LLM to recursively examine a natural language format specification to generate instances for use as strong seed examples to a mutation fuzzer. In contrast to these techniques, ChatAFL separates the information extraction from the fuzzing. ChatAFL first extracts information about the structure and order of inputs from the LLM in machine-readable format (*i.e.,* via grammars and state machines) before running a highly efficient fuzzer that is fed with this information. For efficiency, ChatAFL uses the LLM for a mutational approach (similar to ChatFuzz) only whenever the coverage saturates during fuzzing.

VIII. Conclusion

Protocol fuzzing is an inherently difficult problem. As compared to file processing applications, where the inputs to be fuzzed are given as file(s), protocols are typically reactive systems that involve sustained interaction between system and environment. This poses two separate but related challenges: a) to explore uncommon deep behaviours leading to crashes, we may need to generate complex sequences of valid events and b) since the protocol is stateful, this also implicitly involves on-the-fly state inference during fuzz campaign (since not all actions may be enabled in a state). Moreover, the effectiveness of fuzzing heavily depends on the quality of the initial seeds, which serve as the foundation for fuzzing generation.

In this work, we have demonstrated that for protocols with publicly available RFCs, LLMs prove to be effective in enriching initial seeds, enabling structure-aware mutation, and aiding in state inference. We evaluated ChatAFL on a wide range of protocols from the widely-used ProFuzzBench suite. The results are highly promising: ChatAFL covered more code and explored larger state space in significantly less time compared to the baseline tools. Furthermore, ChatAFL found 9 zero-day vulnerabilities, while the baseline tools only discovered 3 or 4 of them.

Acknowledgment

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing NRF-NCR25-Fuzz-0001). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

References

[1] V. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation: Testing Tools Track*. New York: IEEE, 2020.

[2] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "Nsfuzz: Towards efficient and state-aware network service fuzzing," *ACM Transactions on Software Engineering and Methodology*, 2023.

[3] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *Proceedings of the 31st USENIX Security Symposium*. USENIX Association, 2022, pp. 3255–3272.

[4] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, p. 191, 2022.

[5] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. ACM, 2021, pp. 662–665.

[6] D. Aitel, "The advantages of block-based protocol analysis for security testing," *Immunity Inc., February*, vol. 105, p. 106, 2002.

[7] Jtpereyda, "Boofuzz: A fork and successor of the sulley fuzzing framework." [Online]. Available: https://github.com/jtpereyda/boofuzz

[8] M. Eddington, "Peach fuzzer platform." [Online]. Available: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce

[9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[11] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," 2022.

[12] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[13] OpenAI, "Gpt-4 technical report," 2023.

[14] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," *arXiv preprint arXiv:2205.10583*, 2022.

[15] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[16] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[17] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with gpt-4," 2023.

[18] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2021.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[20] S. Sun, Y. Liu, D. Iter, C. Zhu, and M. Iyyer, "How does in-context learning help prompt tuning?" *arXiv preprint arXiv:2302.11521*, 2023.

[21] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *Proceedings of the 11th International Conference on Learning Representations*, 2023.

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *arXiv preprint arXiv:2201.11903*, 2022.

[23] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative ai," *arXiv preprint arXiv:2306.06782*, 2023.

[24] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[25] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, pp. 1–13.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[27] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proceedings of the 17th European Conference on Computer Systems*, 2022, pp. 166–180.

[28] "libfuzzer – a library for coverage-guided fuzz testing," LLVM. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[29] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486

[30] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[31] H. J. Abdelnur, R. State, and O. Festor, "Kif: a stateful sip fuzzer," in *Proceedings of the 1st international Conference on Principles, Systems and Applications of IP Telecommunications*, 2007, pp. 47–56.

[32] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: toward a stateful network protocol fuzzer," in *Proceedings of the 9th International conference on information security*, vol. 4176. Springer, 2006, pp. 343–358.

[33] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 445–458.

[34] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[35] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, p. 317–329.

[36] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, p. 391–402.

[37] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *Proceedings of the 16th Annual Network & Distributed System Security Symposium*, vol. 8, 2008, pp. 1–15.

[38] M. Zalewski, "Afl." [Online]. Available: https://lcamtuf.coredump.cx/afl/

[39] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[40] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies*, 2020.

[41] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types." in *Proceedings of the 29th USENIX Security Symposium*, 2021, pp. 2597–2614.

[42] A. Andronidis and C. Cadar, "Snapfuzz: high-throughput fuzzing of network applications," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 340–351.

[43] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1597–1612.
[44] C. Lemieux, J. Priya Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *Proceedings of the 45th International Conference on Software Engineering*, 2023.
[45] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.
[46] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," *arXiv preprint arXiv:2304.02014*, 2023.
[47] J. Ackerman and G. Cybenko, "Large language models for fuzzing parsers (registered report)," in *Proceedings of the 2nd International Fuzzing Workshop*, 2023, pp. 31–38.

APPENDIX A
ARTIFACT APPENDIX

CHATAFL is a protocol fuzzer guided by large language models (LLMs). This artifact contains the source code of CHATAFL and all the subjects utilized in the experimental sections of the paper. This document outlines the steps to retrieve the artifact and provides guidance on using it to reproduce the experiments.

A. Description & Requirements

In this section, we introduce how to obtain the artifact, including fuzzers and benchmarks, along with the software and hardware requirements to run it.

1) How to access: We provide public access to our code and experiment setups through the following Zenodo link:

https://zenodo.org/record/10115151

You can also access it in Github:

https://github.com/ChatAFLndss/ChatAFL

The artifact is licensed under the Apache License 2.0.

2) Hardware dependencies: For a single execution of CHATAFL on a subject, standard commodity machines are sufficient to meet our requirements. These machines should have a minimum of a 1-core CPU, 8GB RAM, and a 32GB hard drive. However, when *simultaneously* running multiple fuzzing sessions, it is necessary to ensure that each fuzzing instance receives similar resource allocations.

3) Software dependencies: For running the artifact, a working `Docker` installation is required. The fuzzers execute within Docker containers, but they are controlled by scripts running outside the container on the host system. All scripts on the host system are tested on `Ubuntu 20.04`. However, they are expected to work on any Linux distribution. To run these scripts successfully, the host machines should have `Python-3` installed along with the `pandas` and `matplotlib` libraries.

4) Benchmarks: All the benchmarks required for evaluation are located within the `benchmark` directory of the Zenodo and Github repository.

B. Artifact Installation & Configuration

We now set up the artifact, and the entire process is estimated to take 40 minutes.

(1) Download the artifact from Github:

```
$ git clone https://github.com/ChatAFLndss/
    ChatAFL.git
```

(2) Set OpenAI API Key:

```
$ export KEY=<OPENAI_API_KEY>
```

We require users to use their own OpenAI API key here.

(3) Install the dependencies `Docker` and `Python-3` along with the `pandas` and `matplotlib` required on the host machine:

```
$ cd ChatAFL && ./deps.sh
```

(4) Set up the docker image for each subject with all fuzzers:

```
$ ./setup.sh
```

After these, no further configuration is needed, and we can proceed with a basic run to verify that everything is functioning correctly. For example, to use CHATAFL for fuzzing `pure-ftpd` for a duration of 5 minutes in a single run, we execute the following command:

```
$ ./run.sh 1 5 pure-ftpd chatafl
```

This command encompasses instructions for running the fuzzer and collecting data. Once this process is completed (approximately 5 minutes later), we will observe the output in the same terminal:

```
$ <FUZZER>: I am done!
```

Then we can locate the `results-pure-ftpd` folder, housing the fuzzing results, in the `benchmark` directory.

C. Experiment Workflow

Our experiments consist of two primary phases: (1) executing fuzzers on subjects to gather data, and (2) analyzing this data to compare the performance of CHATAFL with that of baseline tools.

1) Gather code and state coverage: We leverage the following command to run fuzzers on subjects:

```
$ ./run.sh <container_number> <fuzzed_time>
    <subjects> <fuzzers>
```

Where `container_number` specifies how many containers are created to run a single fuzzer on a particular subject (each container runs one fuzzer on one subject). `fuzzed_time` indicates the fuzzing time in minutes. `subjects` is the list of subjects under test, and `fuzzers` is the list of fuzzers that are utilized to fuzz subjects. For example, the command above (`run.sh 1 5 pure-ftpd chatafl`) would create 1 container for the fuzzer CHATAFL to fuzz the subject pure-ftpd for 5 minutes.

Once the allocated time reaches, the fuzzer is terminated, and the data is subsequently gathered. The data gathered from the fuzzing campaign (*i.e.,* code and state coverage, seed corpus, generated grammar corpus, stall messages, and enriched seeds) are archived and compressed. This archive is then extracted from the container and placed into a host folder `results-<subject>` in the `benchmark` directory.

2) Analyze data: After all data is gathered, the script `analyze.sh` can be employed to construct plots illustrating the average code and state coverage over time for fuzzers on each subject. The script is executed using the following command:

```
$ ./analyze.sh <subjects> <fuzzed_time>
```

The script takes in 2 arguments - the list of subjects under test and the duration of the run to be analyzed. For example, executing the command (`./analyze.sh pure-ftpd 240`) generates plots illustrating state and code coverage over 240 minutes for fuzzers running on pure-ftpd. The command processes the results folders, producing `cov_over_time_<subject>.png` and `state_over_time_<subject>.png` visualizations.

Finally, after completing the evaluation, we can execute the `clean.sh` script to remove all docker containers and images from the system, leaving only the artifact folder.

D. Major Claims

- C1: CHATAFL covers more states and achieves the same state coverage faster than baselines. This is proven by experiment (E1), whose results are reported in [Table III and Table IV].
- C2: CHATAFL covers more code and achieves the same code coverage faster than baselines. This is proven by experiment (E1), whose results are reported in [Table V].
- C3: Each strategy proposed in the paper contributes to varying degrees of improvement in code coverage. This is proven by experiment (E2), whose results are reported in [Table VI].

E. Evaluation

To conduct the experiments outlined in the paper, we utilized a vast amount of resources. We executed a 24-hour fuzzing session using 5 fuzzers on 6 subjects, each iterated 10 times. Consequently, it is impractical to replicate all the experiments within a single day using a standard desktop machine. To facilitate the evaluation of the artifact, we downsized our experiments, employing fewer fuzzers, subjects, and iterations.

1) Experiment (E1): [Improvement of state and code coverage] [5 human-minutes + 180 compute-hours]: CHATAFL outperforms AFLNET in state coverage and code coverage (present results for claims C1 and C2).

[How to] Run two fuzzers, CHATAFL and AFLNET, on the three subjects `kamailio`, `pure-ftpd`, and `live555`, respectively, iterating the process 5 times. Each execution takes place within a container and spans a duration of 360 minutes. Consequently, this experiment involves a total of 30 containers, with each container running fuzzing for 240 minutes and coverage collection for 120 minutes.

[Preparation] Ensure that the artifact installation is complete, meaning `setup.sh` has been executed.

[Execution] Execute the following commands:

```
$ ./run.sh 5 240 kamailio,pure-ftpd,live555
    chatafl,aflnet
$ ./analyze.sh kamailio,pure-ftpd,live555 240
```

[Results] Upon completion of the commands, a folder prefixed with `res_` will be generated. This folder contains PNG files illustrating the state and code covered by two fuzzers over time as well as the output archives from all the runs. It will be placed in the root directory of the artifact.

2) Experiment (E2): [Ablation Study] [5 human-minutes + 180 compute-hours]: Each strategy in CHATAFL contributes to enhancing code coverage (present results for the claim C3).

[How to]: Run the CHATAFL fuzzer and two different ablations - CHATAFL-CL1, CHATAFL-CL2, over the two subjects `proftpd` and `exim`, respectively, iterating the process 5 times. Each execution takes place within a container and spans a duration of 360 minutes. Consequently, this experiment involves a total of 30 containers, with each container running fuzzing for 240 minutes and coverage collection for 120 minutes.

[Preparation] Ensure that the artifact installation is complete, meaning `setup.sh` has been executed.

[Execution] Execute the following commands:

```
$ ./run.sh 5 240 proftpd,exim chatafl,chatafl-
    cl1,chatafl-cl2
$ ./analyze.sh proftpd,exim 240
```

[Results] Upon completion of the commands, a folder prefixed with `res_` will be generated. This folder contains PNG files illustrating the code covered by three fuzzers over time as well as the output archives from all the runs. It will be placed in the root directory of the artifact.

F. Customization

We have the flexibility to choose the fuzzers for comparisons and the subjects to undergo fuzzing. Additionally, we can define the fuzzing duration and extend our benchmarks by incorporating new subjects. For instance, we included a new subject, `Lighttpd1`, in our benchmarks.