REACTIVE SYSTEM FUZZING

RUIJIE MENG

NATIONAL UNIVERSITY OF SINGAPORE

2025

REACTIVE SYSTEM FUZZING

RUIJIE MENG

(M.S., University of Chinese Academy of Sciences)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2025

Thesis Advisor:

Professor Abhik Roychoudhury

Examiners:

Associate Professor Zhenkai Liang
Assistant Professor Manuel Rigger

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Ruijie Meng

Ruijie Meng

July 2025

Acknowledgments

My greatest thanks go to my advisor, Prof. Abhik Roychoudhury, for his invaluable advice and constant support throughout my Ph.D. journey. He taught me how to approach research problems systematically, think critically, and strive for work with lasting impact. I will always value the many insightful discussions we had, which encouraged me to think independently and cultivate my own perspectives. His passion for research and imaginative thinking has profoundly shaped my approach and will continue to influence my future career.

I would also like to thank Zhenkai Liang and Manuel Rigger for serving on my thesis committee and for their valuable feedback, which helped improve this thesis. My heartfelt thanks also go to my collaborators: Zhen Dong, Gregory J. Duck, Jialin Li, Marcel Böhme, Martin Mirchev, Van-Thuan Pham, George Pîrlea, and Ilya Sergey. Their contributions have been instrumental to our joint work, and I have learned so much from them, not only about research but far beyond it.

It has been my great honor to have the support from the "Roy" family: Marcel Böhme, Xiang Gao, Van-Thuan Pham, and Shin Hwei Tan, whose generous and constructive guidance has helped me a lot in research, life, and also career development. I am especially grateful to Zhiyu Fan, my closest colleague, with whom I shared countless hours in the lab as we explored, learned, and grew together.

I thank my lab mates and friends at NUS, including Umair Z. Ahmed, Zhendong Ang, Leonhard Applis, Jinsheng Ba, Andreea Costea, Ali El-Husseini, Jingyan Huang, Peisen Lin, Qiuxia Lin, Jiahao Liu, Yu Liu, Zhengxiong Luo, Martin Mirchev, Chee Yong Neo, Yannic Noller, Suyi Ong, Haifeng Ruan, Ridwan Shariffdeen, Yahui Song, Jiawei Wang, Dylan Wolff, Yuanyuan Wu, Jiacheng Xu, Ziyi Yang, Xiao Liang Yu, Mehtab Zafar, Yuntong Zhang, and Huan Zhao. I will always cherish the great times we spent at NUS and in Singapore. I would also like to thank NUS professors Umang Mathur and Prateek Saxena, for their advice and support.

Last but not least, I would like to express my heartfelt gratitude to my family for their unshakable love. In every difficult time, they have always been there, offering strength and courage. I am immensely thankful to my husband, whose unconditional support and constant encouragement have helped me navigate every step of this journey and grow into a better version of myself.

Contents

Abstract

Reactive systems, such as communication protocols, that continuously interact with execution environments, form the foundation of critical software infrastructure. Any bugs or vulnerabilities within these systems may lead to severe consequences. Reactive systems, therefore, must be continuously and automatically secured against vulnerabilities. Unlike transformational systems such as compilers and parsers, which have been the subject of extensive research, suitable techniques for automated testing of reactive systems are still severely lacking. Developers of reactive systems traditionally rely on model checking for validation. Although model checking is a verification technique, it is primarily used for bug finding in practice. While effective, its scalability and usability problems limit its widespread adoption.

In this thesis, we develop automated testing techniques that are effective, scalable, and usable for finding bugs in reactive systems. Greybox fuzzing, known for its low barrier to entry, is a promising technique in this regard. Unfortunately, greybox fuzzing is traditionally designed to test transformational systems automatically and is less effective for reactive systems. Reactive systems are difficult to test due to their inherent statefulness, where sending the same input messages twice might yield a different response every time. This thesis is built to address this core challenge. To achieve this, we develop a new technical method that synergizes the concepts of model checking with advances in greybox fuzzing. Our technical method seeks to achieve bug-finding capabilities close to model checking while still maintaining the usability and scalability of greybox fuzzing.

We have used this method to address the statefulness problem in different contexts. Specifically, we advance greybox fuzzing to detect violations of stateful properties, those expressed in Linear-time Temporal Logic (LTL), going far beyond simple oracles such as crashes and memory safety errors. Furthermore, we leverage the capability of Large Language Models (LLMs) to address the long-standing challenge of state identification in single-node systems, especially protocol implementations. To address distributed systems with distributed states, we introduce a greybox-fuzzing technique, where we leverage Lamport timelines to visualize the states of distributed systems and build a reactive fuzzer to adaptively inject inputs based on observed states. To capture states in complex environments, we develop an automated

method using greybox fuzzing to generate the effect of different environmental states, providing a practical alternative to traditional environment modeling.

The body of work represents a practical advance in automated validation of reactive systems and a conceptual advance in greybox fuzzing. To date, our released open-sourced tools have discovered over 100 vulnerabilities in widely-used reactive systems, more than 50 of which are security-critical vulnerabilities registered as CVEs at the US National Vulnerability Database. These make a step forward toward building more secure and reliable reactive systems. Beyond the reactive systems investigated in this thesis, an increasing number of stateful reactive systems are emerging. Moving forward, we would like to see a much broader type of stateful reactive systems being checked routinely using effective, scalable, and usable techniques.

List of Figures

List of Tables

Chapter 1

Introduction

Reactive systems represent a fundamental category of software systems, distinct from transformational systems in both behavior and purpose. Broadly speaking, software systems can be categorized as transformational or reactive [67]. At an abstract level, transformational systems can be viewed as mapping functions that take inputs, apply predefined transformations to them, and produce corresponding outputs. Typical examples of transformational systems include compilers and parsers. In contrast, many software applications in our daily lives cannot be characterized as simple, one-shot transformations. Instead, they continuously interact with complex execution environments: receiving input messages from environments, updating their internal states, and sending back responses. These systems are inherently non-terminating. Such systems are known as reactive systems.

Reactive systems are ubiquitous in the interconnected world today, where interaction and communication among software components are paramount. Servers, distributed systems, control systems, and autonomous systems (e.g., self-driving cars) are all instances of reactive systems. In addition, reactive systems form the backbone of critical infrastructure. For example, communication protocols such as OpenSSL are the most exposed components of every software system that is directly or indirectly connected to the Internet. A single bug in such a system can have catastrophic consequences. The infamous Heartbleed vulnerability [120] in OpenSSL, for example, led to severe data leakage and huge financial loss. Therefore, it is important to find bugs early before they lead to any consequences.

However, finding bugs in reactive systems is difficult. For software systems, one of the most practical and widely-used methods is automated testing, but automated testing of reactive systems presents significant challenges. The key challenge stems

from the inherent statefulness of reactive systems: sending the same message twice might yield a different response every time, depending on their states. Any effective testing techniques for reactive systems must, therefore, validate whether each state behaves as expected. Yet identifying states in real-world system implementations is hard. Due to large codebases and complex program behaviors, it is not trivial to determine which variables carry states.

Fuzz testing (i.e., fuzzing) has emerged as one of the most promising automated testing techniques [20], which continuously generates inputs and reports those that crash the program. Greybox fuzzing has gained significant attention from both industry and academia due to its scalability and low barrier to entry. Greybox fuzzing leverages lightweight code-coverage feedback (e.g., branch coverage [167, 57]) from the program to select interesting inputs, thereby steering the search. This technique is particularly effective for testing stateless transformational systems, where the same input almost produces the same output. However, for stateful reactive systems, code-coverage feedback becomes less effective for input selection, as even the same input may lead to different outputs depending on the internal states of the systems.

Traditionally, stateful reactive systems are validated using model checking [60, 162]. Although model checking is a verification technique, its common usage in practice is for bug finding. In model checking, reactive systems are modeled as finite-state machines and then checked against the desired properties, where the states are explicitly identified. This approach is effective in checking stateful reactive systems, especially safety-critical software systems. Unfortunately, while using it to check today's software applications, it suffers from the well-known state-explosion problem. In addition, modeling is not trivial, which usually requires much manual effort and expertise. As noted in the prior works [33, 19], building a model often takes multiple person-years and also demands years of PhD-level expertise. Such requirements are rarely feasible in the current testing environment.

In this thesis, we develop automated testing techniques that are effective, scalable, and usable for stateful reactive systems. To this end, at the technical layer, we synergize the concepts of model checking with recent advances in greybox fuzzing and bring statefulness to the greybox fuzzing world. Our goal is to approach the bug-finding capabilities of model checking without providing formal verification

guarantees, while still maintaining the scalability and usability of greybox fuzzing. Along this line, we have addressed the statefulness problem of reactive systems in various dimensions: in temporal logic properties [105], in single-node systems (especially network protocols) [107], in distributed systems [109], and also in complex execution environments [106]. We also reflect on the stateful fuzzing of reactive systems and discuss open problems in this regard [108].

Research Overview. In the following, we give an overview of our approaches.

- **How can we find violations of stateful properties in practice?** Reactive systems are often expected to satisfy complex security and reliability properties that span sequences of states. For example, in the FTP protocol, a user is allowed to copy files into their directory only after successfully logging in. Violations of such stateful properties can lead to severe security issues, such as authorization bypasses. Finding such violations traditionally relies on verification techniques like model checking, which is usually considered beyond the reach of testing techniques. To address this problem, we transport the concept of model checking into greybox fuzzing without increasing the barrier to entry. We introduce LTL-Fuzzer [105], which uses greybox fuzzing to find violations of linear-time temporal logic (LTL) properties. Unlike model checking, which requires constructing a model of the system, LTL-Fuzzer uses greybox fuzzing to directly conduct the search on the system implementations for finding violating traces, thereby eliminating the need for manual modeling. We evaluated LTL-Fuzzer on the real-world implementations of network protocols. Among 50 properties extracted from network Request for Comments (RFCs), LTL-Fuzzer found 15 previously unknown violations for both liveness and safety properties.

- **How can we effectively explore the state space of stateful network protocols?** Protocol implementations are essential components of internet-facing servers, enabling communication between servers and clients, and also forming a critical attack surface. However, their inherently stateful and reactive nature makes them difficult to test. Finding a vulnerability in a specific state requires sending the right input messages in the right order. Without knowledge

of the message formats and state machines, it is difficult for a testing tool to explore the state space. While such information for widely-used protocols is publicly available in documents (e.g., RFCs), these documents are often hundreds of pages long and written in natural language, making them difficult for automated tools to interpret and utilize. The recent advances in large language models (LLMs) have demonstrated the ability to answer specific questions about diverse documents. This opens up the opportunity to develop a protocol fuzzer that interacts with LLMs to retrieve relevant information from RFCs. Based on this, we developed ChatAFL, an LLM-guided protocol fuzzer that uses the LLM to bring the state-exploration capability of model checking into greybox fuzzing. When applied to widely-used protocol implementations, ChatAFL improves state exploration by six times faster and covers about 30% of states compared to state-of-the-art approaches. ChatAFL is also among the first to demonstrate the utility of LLMs in addressing security challenges.

- **How can we effectively explore the state space of stateful distributed systems?** Beyond single-node systems, stateful reactive systems can also operate in distributed settings, such as distributed databases. It is more challenging to test stateful distributed systems. Distributed systems are usually large-scale with multiple nodes, follow complex communication protocols such as Raft and Paxos, and exhibit more complex program behaviors—each node has its own state, while the nodes also communicate with each other. How could we effectively explore the state space in distributed systems? We develop Mallory, a lightweight way to test distributed systems. At its core, Mallory employs Lamport diagrams to visualize states of distributed systems, which form the state feedback function to guide the fuzzing search. Moreover, Mallory itself is a reactive system, dynamically deciding inputs to inject based on observed states to maximize state exploration. Mallory provides a third solution to validating distributed systems alongside software model checking and stress testing, and takes a trade-off between usability and effectiveness. In widely-used and rigorously-tested distributed systems (e.g., MongoDB and ScyllaDB), it achieved 54.27% more states than the

state-of-the-art technique.

- **How can we capture different environmental states in practice?** States also exist in complex program execution environments. Reactive systems are not executed in isolation but rather interact with complex execution environments that drive program behaviors. To comprehensively explore diverse program states, it is critical to account for the influence of execution environments. However, capturing environmental states presents a significant challenge for any testing, analysis, and verification techniques. Environment modeling is the dominant approach in this regard; however, it requires significant effort and expertise from developers. We provided an alternative solution to environmental modeling, reducing the reliance on manual intervention. We developed $\mathcal{E}$FUZZ, which advances the search of greybox fuzzing over the full execution environment and applies selective mutations to produce the effect of different environments. All these are done automatically without resorting to modeling and manual effort. In our evaluation, $\mathcal{E}$FUZZ could explore over 30% more program behaviors than existing techniques, and it quickly discovered a bunch of environment-inducing vulnerabilities in commonly used programs and libraries.

Contributions. We summarize the contributions of this thesis in the following three key perspectives.

- From a problem perspective, to the best of our knowledge, we provide the first effective, scalable, and usable method to the long-standing problem: automated validation of stateful reactive systems. We identify statefulness as the core difficulty and then develop a technical method that synergizes model checking and greybox fuzzing to address it. This technical method has demonstrated effectiveness across a range of stateful contexts, including temporal properties, system implementations, and execution environments. Our reflection on AFLNET, as a representative practical approach for reactive systems, further shows the impact of a practical way to validate reactive systems in industry, academia, and even education.

- From a practical perspective, we have open-sourced all developed tools to foster further research and adoption. To date, these tools have found over 200 bugs in widely-used stateful reactive systems, with more than 50 of which are security-critical vulnerabilities registered as CVEs at the US National Vulnerability Database, making a step forward toward more secure and reliable reactive systems. The practical utility of these tools has garnered substantial interest from industries that seek to integrate them into their daily workflows.
- From the technical perspective, we extend the scope of traditional greybox fuzzing, which has largely focused on stateless transformational systems, into the domain of stateful reactive systems. Furthermore, we transport the key capabilities of model checking, such as property violation detection and systematic state exploration, into a more scalable and usable greybox fuzzing framework. In this way, we can enjoy the practical benefits of model checking, although there are no formal verification guarantees.

Thesis Organization. The remainder of this thesis is organized as follows. Chapter 2 provides the background on reactive systems and commonly used bug-finding techniques. Chapter 3 introduces LTL-Fuzzer, a testing method designed to find violations of stateful properties. In Chapter 4, we detail our approach ChatAFL for automatically identifying states in network protocols. Chapter 5 presents a greybox fuzzing technique Mallory for exploring the state space in distributed systems. Chapter 6 explores an alternative to environment modeling—$\mathcal{E}$fuzz, demonstrating how greybox fuzzing can capture the effects of different environmental states. Chapter 7 reviews related work, and Chapter 8 concludes the thesis with a summary of these techniques, reflections on the impact of practical validation approaches for reactive systems, and discussions of future research directions.

Chapter 2

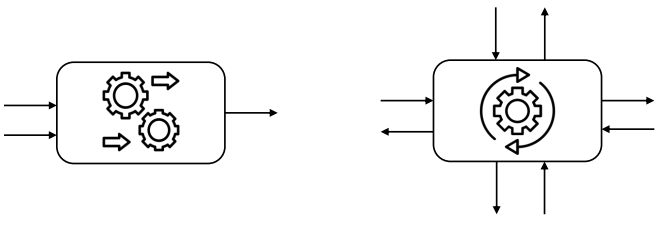
Background

In this chapter, we introduce the background on reactive systems and related bug-finding techniques.

2.1 Reactive Systems

The term reactive systems was first introduced by David Harel and Amir Pnueli in the 1980s [67] to describe a class of systems fundamentally different from transformational systems. At a high level of abstraction, *transformational systems* (as shown in Figure 2.1 (a)) can be understood as computational entities that process a set of inputs, apply predefined transformations to them, and produce the corresponding outputs. Transformational systems are usually static and self-contained, with operations concluded once the input-output transformation is complete. Typical examples of transformational systems include compilers, which transform source code into executable machine code, and parsers, which analyze structured input formats like XML files to generate syntactic or semantic analysis results.

In contrast, *reactive systems* are characterized by their continuous and dynamic



(a) Transformational System (b) Reactive System

Figure 2.1: High-level abstractions of transformational systems (a) and reactive systems (b).

interactions with their environment (as shown in Figure 2.1 (b)). Instead of performing a single, finite transformation, they continuously respond to external stimuli or events by transitioning between states and producing outputs that, in turn, influence their surroundings. Reactive systems maintain ongoing, real-time interactions with their environment. By design, reactive systems are non-terminating. Given this definition, we can see that reactive systems are pervasive in our daily lives, from the smartphone in our pockets to large-scale could applications.

Reactive systems are typically implemented as *stateful*, to achieve continuous and dynamic interactions with environments. Reactive systems maintain internal states to record past communication history and system configurations. In practice, the state can be carried through program variables or other structures, depending on the specific system implementation. When a reactive system receives new environmental input, it generates a response based on its internal state. As a result, in different states, even the same input executed twice might yield a different response each time. This is known as the *statefulness* problem of reactive systems. For example, in the Real-Time Streaming Protocol (RTSP) illustrated in Figure 4.2, a Play command successfully initiates media playback only if the system is in the READY state; otherwise, the operation would fail.

This statefulness problem presents a unique challenge when testing reactive systems. Specifically, the reliability of reactive systems depends not only on a single state, but on a sequence of states that evolve over time—often described using temporal-logic properties. Validating state-dependent properties is inherently difficult for traditional testing techniques. We address this challenge in Chapter 3. Furthermore, effective testing of reactive systems requires validation that each state behaves as expected, making state identification critical but also challenging. In certain domains, such as communication protocols, states are explicitly defined within formal or semi-formal specifications like state machines (as shown in Chapter 4). However, in distributed systems, state identification becomes significantly more complex due to the immense complexity of states and the fact that states are located across multiple components (as shown in Chapter 5). In addition, the internal states of reactive systems are tightly coupled with the states of their execution environment, including configuration files, databases, user interactions, and other environmental resources. Capturing such vast and complex execution environments

poses yet another significant challenge, which is addressed in Chapter 6.

2.2 Formal Verification

Finding bugs in software systems, including reactive systems, is a long-standing problem in both industry and academia. The approaches to tackling this problem have been forked into two pathways: formal verification and software testing. Formal verification systematically explores all program paths to prove that a program satisfies a given specification, whereas software testing searches specific execution paths determined by test inputs. In this section, we introduce and discuss the principles and techniques of formal verification.

Formal verification [103] aims to prove or disprove the correctness of a system with respect to a given formal specification or property. Among the various formal verification techniques, model checking [39] is one of the most widely studied and applied. Model checking verifies whether a finite-state model of a system satisfies a specified property. Although model checking is fundamentally a verification technique, it is extensively used in practice for bug finding, which is widely adopted by the developers of reactive systems. Formally, model checking can be expressed as follows: $M \models \phi$, where M is a finite-state model for the system, and ϕ is the desired property. This process involves a systematic and exhaustive exploration of the model M to find violations of the property ϕ, which are also bugs in the system.

2.2.1 System Modeling

Model checking requires constructing a model of the system implementation, typically represented as a finite-state machine, to enable systematic and exhaustive exploration. However, reactive systems are naturally non-terminating and have an enormous number of states, which exacerbates the state explosion problem. For such infinite models, counterexample-guided abstraction refinement (CEGAR) [36, 37] is introduced to address it. CEGAR addresses this problem in two stages: abstraction, which simplifies a model by grouping states into a finite representation, and refinement, which increases the precision of the abstraction whenever a counterexample is found to check whether the counterexample is spurious.

In addition, reactive systems are not executed in isolation, but rather interact with

a complex execution environment such as libraries, kernel, and drivers, which drives the program behaviors. A dominant approach for handling different environments is environment modeling, which is used by verification and analysis methods. To verify an open software system that interacts with the environment, model-checking methods typically describe the environment as a separate process. This process captures an over-approximation of possible behaviors that could be exhibited by real, concrete environments. The environment process is then composed of the open software system, forming a closed system which can then be subjected to search. Environment synthesis for model checking has been studied in works such as [147]. These approaches depend on user-provided specifications to implement a safe approximation of the environment, and do not use concrete environments to demonstrate program errors.

Despite its theoretical rigor, building these system models often requires much manual effort and expertise. Prior studies have reported that writing abstract models can take multiple person-years of effort [33]. In addition, it requires years of PhD-level expertise [19]. In some cases, the specification is even larger than the code being verified. This usability barrier severely limits the use of model checking in practice, although model checking can provide the highest level of security guarantees.

2.2.2 Temporal Logic Properties

The properties to be verified are often described in temporal logics, such as linear-time temporal logics (LTL) [76, 53], which we adopt in this thesis to formalize the properties of systems. LTL provides a formal specification mechanism to quantitatively describe the desired behaviors of a system over time. It is a modal temporal logic in which modalities express how properties hold across sequences of states along a linear progression of time.

LTL Syntax. The set of LTL formulas is defined as follows, where AP is a set of atomic propositions and $p \in AP$:

$$\psi, \varphi ::= \text{true} \mid \text{false} \mid p \mid \neg\psi \mid \psi \vee \varphi \mid \psi \wedge \varphi \mid X\psi \mid F\psi \mid G\psi \mid \psi U \varphi \mid \psi R \varphi$$

Here, ψ and φ represent formulas built from atomic propositions. The temporal operator $X\psi$ (next) asserts that ψ holds in the next state. The operator $F\psi$ (finally) means that ψ holds at some point in the future, while $G\psi$ (globally) requires that

ψ holds in all future states. The formula $\psi U \varphi$ (until) means that ψ must hold continuously until φ eventually holds. In contrast, $\psi R \varphi$ (release) states that φ must hold until (and including the point at which) ψ becomes true; if ψ never becomes true, φ must hold indefinitely. Common formula transformations are as follows:

$$\psi \rightarrow \varphi := \neg\psi \vee \varphi$$
$$F\psi := true\ U\ \psi$$
$$G\psi := \neg F \neg\psi$$
$$\psi R \varphi := \neg(\neg\varphi U \neg\psi)$$

LTL Semantics. LTL formulas are interpreted over infinite sequences of sets of atomic propositions, known as traces. A trace $\pi = p_0, p_1, p_2, p_3, ...$, where $p_i \in AP$ is a mapping $\pi : \mathbb{N} \rightarrow 2^{AP}$. We denote by π^i the suffix of π starting from the position i, i.e., $\pi^i = p_i, p_{i+1}, p_{i+2}, ...$

For a trace $\pi = p_0, p_1, p_2, p_3, ...$ and $i \in \mathbb{N}$, the satisfaction relation for LTL formulas is defined inductively over the formula structure:

- $\pi \models\ p\ \ if\ p \in p_i$
- $\pi \models\ \neg\psi\ \ if\ \pi \not\models\ \neg\psi$
- $\pi \models\ \psi \vee \varphi\ \ if\ \pi \models\ \psi\ or\ \pi \models \varphi$
- $\pi \models\ X\psi\ \ if\ \pi^1 \models \psi$
- $\pi \models\ \psi U \varphi\ \ if\ there\ is\ i \geq 0\ such\ that\ \pi^i \models \varphi\ and\ \pi^k \models \psi\ for\ all\ 0 \leq k < i$

A trace π satisfies an LTL formula ϕ if $\pi \models \phi$.

Safety and Liveness Properties. LTL formulas can express two key classes of properties: safety and liveness. *Safety properties* assert that "something bad never happens". Intuitively, a safety property is violated by a finite prefix of a trace. Formally, a property ϕ is a safety property if every violation of ϕ has a finite bad prefix—i.e., a finite trace that cannot be extended to any trace satisfying ϕ. *Liveness properties*, in contrast, assert that "something good eventually happens". They cannot be violated by any finite prefix, meaning that any finite trace can potentially be extended to satisfy the property.

More formally, every LTL formula can be decomposed into a conjunction of safety and liveness components. A commonly used syntactic characterization is that:

- A formula $\phi = Gp$ is a safety property if p is a formula referring only to the current and past states (i.e., a pure past formula).
- A liveness property can often be expressed in the form $F(p_t \wedge f_t)$ for $\forall t \in T$ where T is a finite index set, the formula p_t is a pure past formula, and f_t is a satisfiable pure future formula.

2.2.3 Automata-Theoretic Model Checking

The automata-theoretic approach provides a uniform algorithm framework for model checking linear-time properties [149]. It is widely adopted in existing model checkers such as SPIN [72] and SMV [35]. In this approach, when checking a system against an LTL formula ϕ, the standard method is to translate the negation of ϕ into Büchi automata $\mathcal{A}_{\neg\phi}$. Any trace accepted by $\mathcal{A}_{\neg\phi}$ represents a counterexample that violates the original LTL property ϕ. Given a system model M, the LTL property ϕ is satisfied by M (denoted $M \models \phi$), if and only every trace π of M satisfies $\phi(\pi \models \phi)$. In practice, to verify whether M satisfies ϕ, we check the emptiness of the language of the product automata $M \times \mathcal{A}_{\neg\phi}$. If the language is non-empty, it indicates that M violates the LTL property ϕ. In the subsequent step, the system model is further refined to determine whether the violation is real or spurious.

2.3 Software Testing

Software testing is a practical and cost-effective approach to finding bugs by executing programs with various inputs. Among modern testing techniques, fuzzing [20] is one of the most effective techniques to automatically find vulnerabilities in large-scale software systems. It works by continuously generating inputs and reporting those that trigger crashes. Broadly, fuzzing techniques can be classified into three categories: blackbox fuzzing, greybox fuzzing, and whitebox fuzzing.

2.3.1 Blackbox Fuzzing

Blackbox fuzzing generates new inputs without leveraging any internal knowledge of the program under test. Basically, it includes two variants: *mutation-based* and *generation-based* fuzzing. Mutation-based fuzzing starts with a set of initial inputs,

known as seeds, and applies random mutation operators to generate new inputs. This process is repeated iteratively until a specified timeout is reached.

Generation-based fuzzing is more commonly used for testing reactive systems than mutation-based fuzzing. Several generation-based fuzzers have been developed in both academic and industrial settings, including BooFuzz [81] in academia and tools like Peach [52] for network protocols and Jepsen [88] for distributed systems in industry. These fuzzers generate message sequences by traversing a predefined state model, often represented as a finite state machine or a graph. For each message, they leverage data models or grammars to construct syntactically valid inputs, which are then used to stress-test the system under test.

The effectiveness of generation-based fuzzers largely depends on the completeness and accuracy of the provided state and data models. These models are normally written manually, based on the developers' understanding of the protocol specification and sample network traces between the client and the server. However, such manually crafted models may fail to accurately reflect the actual system implementation. Protocol specifications are often hundreds of pages long and written in natural languages. Developers of implementations may misinterpret the specifications, such as adding new states and transitions.

Algorithm 1: Greybox Fuzzing

Input: Seed Input S, Program P_0
Output: Crashing Inputs $S_{\text{✗}}$

```
1:  P_f ← Instrument (P_0)
2:  repeat
3:      s ← ChooseNext (S)
4:      p ← AssignEnergy (s)
5:      for i from 1 to p do
6:          s' ← MutateInput (s)
7:          if IsCrash (s', P_f) then
8:              add s' to S_✗
9:          else if IsInteresting (s', P_f) then
10:             add s' to S
11: until timeout reached or abort-signal
```

2.3.2 Greybox Fuzzing

Greybox fuzzing leverages lightweight feedback (e.g., branch coverage) to select the interesting inputs and then steer the fuzzing search. Greybox fuzzing takes a balance between blackbox fuzzing and whitebox fuzzing. The core algorithm is shown in Algorithm 1. At compile time, greybox fuzzing instruments specific control locations in the program (line 1) to collect feedback on code execution during runtime. The process begins with an initial queue of seed inputs provided by the user. The greybox fuzzer iteratively selects inputs from the queue for the mutation (line 3), with the time spent on mutating each input determined by an assigned energy value (line 4).

The selected seed inputs are then mutated using various mutation operators (e.g., bit flipping or byte insertion) to generate new inputs (line 6). Each generated input is executed to determine whether it causes the program to crash; Any crashing inputs are saved as outputs (lines 7–8). If a newly generated input exercises previously unseen program behaviors based on runtime feedback, it is considered "interesting" and added to the seed queue for further exploration (lines 9–10). This iterative process allows the fuzzer to use code coverage feedback to progressively explore deeper and more complex execution paths.

Popular greybox fuzzers include AFL [167] and libFuzzer [98]. The main advantages of this technique lie in its scalability, which scales to very large-scale programs, and its usability, which requires almost no manual effort and expertise during setup. However, most existing greybox fuzzers are primarily designed for testing transformational programs, such as compilers and parsers. In these programs, the same input almost produces the same output, making code coverage a reliable metric for selecting interesting seeds and steering the search.

2.3.3 Whitebox Fuzzing

Whitebox fuzzing relies on symbolic execution to systematically generate inputs, where each input exercises a different program path. Symbolic execution is a program analysis technique. In symbolic execution, program inputs are labeled as symbolic variables instead of concrete values. During execution, program variables are represented as symbolic expressions over these input symbols, and the analysis

tracks how these expressions evolve throughout the execution.

As the program executes symbolically, it accumulates a set of logical constraints (i.e., path constraints) that describe the requirements for reaching a particular execution path. Each path condition is a conjunction of constraints that must hold for the corresponding path to be feasible. These path conditions are then solved using Satisfiability modulo theories (SMT) solvers to produce concrete program inputs that drive execution along specific paths.

Popular symbolic execution engines include KLEE [27] and SAGE [64]. Symbolic execution is particularly effective for generating high-coverage test cases, uncovering deep bugs, and verifying program properties. However, it faces challenges such as path explosion and the computational cost of constraint solving, which limit its scalability on large or complex programs.

Chapter 3

Finding Stateful Property Violations

In this chapter, we introduce a greybox fuzzing framework to find violations of Linear-time Temporal Logic (LTL) properties. Our framework takes as input a sequential program written in C/C++ and an LTL property. It finds violations, or counterexample traces, of the LTL property in stateful software systems; however, it does not achieve verification. Our work substantially extends directed greybox fuzzing to witness arbitrarily complex event orderings. We note that existing directed greybox fuzzing approaches are limited to witnessing reaching a location or witnessing simple event orderings like use-after-free. At the same time, compared to model checkers, our approach finds the counterexamples faster, thereby finding more counterexamples within a given time budget.

Our LTL-Fuzzer tool, built on top of the AFL fuzzer, has been shown to be effective in detecting bugs in well-known protocol implementations, such as OpenSSL and Telnet. We use LTL-Fuzzer to reproduce known vulnerabilities (CVEs), to find 15 zero-day bugs by checking properties extracted from RFCs (for which 12 CVEs have been assigned), and to find violations of both safety as well as liveness properties in real-world protocol implementations. Our work represents a practical advance over software model checkers—while simultaneously representing a conceptual advance over existing greybox fuzzers. Our work thus provides a starting point for understanding the unexplored synergies among software model checking, runtime verification, and greybox fuzzing.

3.1 Introduction

Software model checking is a popular validation and verification method for reactive stateful software systems. It is an automated technique to check temporal

logic properties (constraining event orderings in program execution) against a finite state transition system. Model checking usually suffers from the state space explosion problem; this is exacerbated in software systems, which are naturally infinite-state. To cope with infinitely many states, the research community has looked into automatically deriving a hierarchy of finite state abstractions via predicate abstractions and abstraction refinement of the program's data memory (e.g., see [12]). Whenever a counterexample trace is found in such model checking runs, the trace can be analyzed to find (a) whether it is a spurious counterexample introduced due to abstractions, or (b) the root cause/bug causing the counterexample. This has rendered model checking a useful automated bug-finding method for software systems.

Runtime verification is a lightweight and yet rigorous verification method, which complements model checking [16, 96, 94]. In runtime verification, a single execution of a system is dynamically checked against formally specified properties (e.g., temporal logic properties). Specifically, formal properties specify the correct behaviors of a system. Then the system is instrumented to capture events that are related to the properties being checked. During runtime, a monitor collects the events to generate execution traces and checks whether the traces conform to the specified properties. When the properties are violated, it reports violations. Runtime verification aims to achieve a lightweight but not full-fledged verification method. It verifies software systems at runtime without the need to construct models about software systems and execution environments. However, to generate effective execution traces, software systems are required to be fed many inputs. These inputs are usually obtained manually or via random generation [94]; therefore, runtime verification may take much manual effort and explore many useless inputs in the process of exposing property violations.

Parallel to the works in software model checking and runtime verification, greybox fuzzing methods [167, 98] have seen substantial recent advances. These methods conduct a biased random search over the domain of program inputs, to find bugs or vulnerabilities. The main advantage of greybox fuzzing lies in its scalability to large software systems. However, greybox fuzzing is only a testing (not verification) method, and it is mostly useful for finding witnesses to simple oracles such as crashes or overflows. Recently, there have been some extensions of greybox fuzzing methods

towards generating witnesses of more complex oracles, such as tests reaching a location [21]. However, generating inputs and traces satisfying a complex temporal property remains beyond the reach of current greybox fuzzing tools. Thus, today's greybox fuzzing technology cannot replace the bug-finding abilities of software model checking and runtime verification.

In this work, we take a step forward in understanding the synergies between software model checking, runtime verification, and greybox fuzzing. Given a sequential program and a Linear-time Temporal Logic (LTL) property ϕ, we construct the Büchi automata $\mathcal{A}_{\neg\phi}$ accepting $\neg\phi$, and use this automata to guide the fuzzing campaign. Thus, given a random input that exercises an execution trace π, we can check the "progress" of π in reaching the accepting states of $\mathcal{A}_{\neg\phi}$, and derive from $\mathcal{A}_{\neg\phi}$, the events that are needed to make further progress in the automata. Furthermore, in general, traces accepted by $\mathcal{A}_{\neg\phi}$ are infinite in length and visit an accepting state infinitely often. To accomplish the generation of such infinite-length traces in the course of a fuzzing campaign, we can take snapshots of the application states (at selected program locations) and detect whether an accepting state of $\mathcal{A}_{\neg\phi}$ is being visited with the same program state. The application state snapshot can also involve a state abstraction if needed, in which case the counterexample trace can be subsequently validated via concrete execution.

We present a fuzzing-based technique that directs fuzzing to find violations of *arbitrary LTL properties*. To the best of our knowledge, no existing fuzzing technique is capable of finding violations of complex constraints on event orderings such as LTL properties. Existing works on greybox fuzzing are limited to finding witnesses of simple properties, such as crashes or use-after-free. This is the main contribution of our work: algorithms and implementation of our ideas in a tool that can validate *any LTL property*, thereby covering a much more expressive class of properties than crashes or use-after-free. Our work adapts directed greybox fuzzing (which directs the search towards specific program locations) to find violations of temporal logic formulae. We realize our approach for detecting violations of LTL properties in a new greybox fuzzer tool called LTL-Fuzzer. LTL-Fuzzer is built on top of the AFL fuzzer [167] and involves additional program instrumentation to check if a particular execution trace is accepted by the Büchi automaton representing the negation of the given LTL property.

We evaluated LTL-Fuzzer on well-known and large-scale protocol implementations such as OpenSSL, OpenSSH, and Telnet. We show that it efficiently finds bugs that are violations of both safety and liveness properties. We use LTL-Fuzzer to reproduce known bugs/violations in the protocol implementations. More importantly, for 50 LTL properties that we manually extracted from Request-for-Comments (RFCs), LTL-Fuzzer found 15 new bugs (representing the violation of these properties), out of which 12 CVEs have been assigned. These are zero-day bugs that have previously not been found. We make the dataset of properties and bugs found available with this work. We expect that in the future, other researchers will take forward the direction in this work to detect temporal property violations via greybox fuzzing. The dataset of bugs found by LTL-Fuzzer can thus form a baseline standard for future research efforts. The dataset and tool are available at https://github.com/ltlfuzzer/LTL-Fuzzer.

3.2 Approach Overview

At a high level, our approach takes a sequential program P and a Linear-time Temporal Logic (LTL) property ϕ as inputs. The atomic propositions in ϕ refer to predicates over the program variables that can be evaluated to true or false. An example is a predicate $x > y$ in which x and y are program variables. Our approach identifies program locations at which the atomic propositions in the LTL property may be affected. For this, we find program locations at which the values of variables in the atomic proposition and their aliases may change.[1] Our technique outputs a *counterexample*, i.e., a concrete program input that leads to a violation of the specification. Counterexample generation proceeds in two phases. In the first phase, the program P is transformed into P'. For this, we use code instrumentation to monitor program behaviors and state transitions during program execution. We check these against the provided LTL property. In the second phase, a fuzzing campaign is launched for the program P' to find a counterexample through directed fuzzing.

We illustrate our technique with an FTP implementation called Pure-FTPd.[2]

[1] In general, our approach requires an alias analysis to map the atomic propositions to program locations.

[2] https://www.pureftpd.org/project/pure-ftpd/

Table 3.1: Mapping between atomic propositions and program locations ("..." indicates omitted loop entries).

Predicate	Atomic Prop.	Locations
$quota\_activated = true$	a	$\langle ftpd.c, 6072\rangle$
$user\_dir\_size > user\_quota$	o	$\langle safe\_rw.c, 12\rangle$ $\langle safe\_rw.c, 43\rangle$
$msg\_quota\_exceeded = true$	n	$\langle ftpd.c, 4444\rangle$ $\langle ftpd.c, 3481\rangle$
$loop\_entry = true$	l	$\langle ftpd.c, 4067\rangle$ ...

Pure-FTPd is a widely-used open-source FTP server that complies with the FTP RFC.[3] Here is a property described in the RFC that an FTP implementation must satisfy. The FTP server must stop receiving data from a client and reply with code 552 when user quota is exceeded while receiving data. Code 552 indicates that the allocated storage is exceeded. Throughout this work, we will use this FTP property—as represented by ϕ—to illustrate how our technique finds property violations in Pure-FTPd.

3.2.1 LTL Property Construction

We start by manually translating the informal property in the RFC into an LTL property ϕ. For this, we search the Pure-FTPd source code using keywords `APPE` and `552`. Source code analysis reveals that (1) Pure-FTPd implements a quota-based mechanism to manage user storage space, and it works only when activated, and (2) the command `APPE` is handled by the function `dostor()`, in which `user_quota_size` is checked when receiving data. When the quota is exceeded, the server replies with code 552 (`MSG_QUOTA_EXCEEDED`) via the function `addreply()`. We therefore construct the property ϕ as

$$\neg F(a \wedge F(o \wedge G\neg n)) \tag{3.1}$$

The negation of ϕ is thus

$$F(a \wedge F(o \wedge G\neg n))$$

where definition of atomic propositions a, o, n appear in Table 3.1.

Next, we identify program locations where the values of variables in atomic propositions in ϕ may change at runtime. A simple example is the proposi-

[3] https://www.w3.org/Protocols/rfc959/

tion $quota\_activated = true$, which corresponds to the program location where quota checking is enabled in Pure-FTPd. In another statement, $user\_dir\_size > user\_quota$, we consider the first statement of functions that are used to store data in user directories. As a result, whenever data is written to user directories, those functions will be invoked, and this proposition will be evaluated, i.e., all cases where the user quota is exceeded will be captured in an execution. For $msg\_quota\_exceeded = true$, we identify function invocations of `addreply(552, MSG_QUOTA_EXCEEDED...)` which are a reply to clients when the quota is exceeded. Specific program locations for each atomic proposition are listed in Table 3.1. Their corresponding code snippets are shown in Code 3.1, Code 3.2, Code 3.3, and Code 3.4. Here, we show one code snippet per atomic proposition. For convenience, we use a tuple $\langle l, p, c_p \rangle$ in which l denotes a program location, p is an atomic proposition, and c_p represents the predicate for the atomic proposition p. At the end of our manual LTL property generation process, we output a list L comprising such tuples. For the example property, the manual process of writing down the predicates and the accompanying tuples was completed by one of the authors in 20 minutes.

```
6063   #ifdef QUOTAS
6064   case 'n': {
...
6072     user_quota_size *= (1024ULL * 1024ULL);
6073  +  if(1){
6074  +    generate_event("a");
6075  +    if(liveness) record_state();
6076  +  }
```

Code 3.1: Enabling the user quota option:<ftpd.c, 6072>.

```
12   safe_write(const int fd, const void * const buf_,
13       size_t count, const int timeout)
14   {
15  +  if(user_dir_size > user_quota){
16  +    generate_event("o");
17  +    if(liveness) record_state();
18  +  }
```

Code 3.2: Writing to user directories:<safe_rw.c, 12>.

3.2.2 Program Transformation

After deriving the property ϕ and the list of tuples L, we transform program P into P', which can report a failure at runtime whenever ϕ is violated. We perform this program transformation using two instrumentation modules: (1) *Event generator,* which generates an event when a proposition in ϕ is evaluated to true at runtime; (2) *Monitor,* which collects the generated events into an execution trace and evaluates if the trace violates ϕ. If a violation is found, the monitor reports a failure.

```
4442   afterquota:
4443     if (overflow > 0) {
4444       addreply(552, MSG_QUOTA_EXCEEDED, name);
4445  +    if(1){
4446  +      generate_event("n");
4447  +      if(liveness) record_state();
4448  +  }
```

Code 3.3: Replying msg_quota_exceeded:<ftpd.c, 4444>.

```
4066   for (;;) {
4067  +  if(1){
4068  +      generate_event("l");
4069  +      if(liveness) record_state();
4070  +  }
```

Code 3.4: Entry of a loop statement:<ftpd.c. 4067>.

Event Generator. To detect changes in ϕ's proposition values during program execution, the event generator injects event generation statements at specific program locations. To do so, the generator takes the list L produced in the previous step as input. For each tuple $\langle l, p, c_p \rangle \in L$, the generator injects a statement `if(`c_p`) generate_event("`p`");` at the program location l, such that an event associated with p can be generated when condition c_p is satisfied. For instance, the program location $\langle ftpd.c, 6072 \rangle$ corresponds to the proposition variable a ($quota\_activated = true$) and the enabling condition is `true`. The generator then inserts a statement `if(1) generate_event("a");` at line 6072 in ftpd.c (see Code 3.1). Consequently, whenever $\langle ftpd.c, 6072 \rangle$ is reached, an event associated with a is generated and recorded at runtime. Instrumentation for the other tuples appears in Code 3.2, Code 3.3, and Code 3.4.

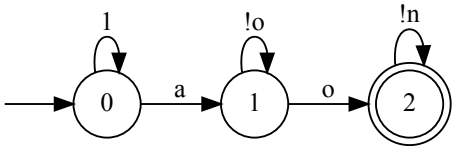


Figure 3.1: Büchi automata accepting traces satisfying $\neg\phi$.

Monitor. The monitor module inserts a *monitor* into program P to verify if the program behavior conforms to property ϕ at runtime. Specifically, the monitor produces a trace τ by collecting events that are generated during execution (by the instrumented code). It then converts the negation of ϕ to a Büchi automata $\mathcal{A}_{\neg\phi}$, and checks whether $\mathcal{A}_{\neg\phi}$ accepts τ. If the trace is accepted, the monitor reports a failure, i.e., ϕ does not hold in P. In our Pure-FTPd example, the negation of ϕ is $F(a \wedge F(o \wedge G\neg n))$, and the converted Büchi automata $\mathcal{A}_{\neg\phi}$ is illustrated in Figure 3.1.

Checking Safety Properties. A Büchi automata accepts a trace τ if and only if τ visits an accepting state of the automata "infinitely often" (e.g., state 2 in Figure 3.1). For the negation of a *safety* property ($\neg\phi$), the Büchi automata $\mathcal{A}_{\neg\phi}$ accepts all traces which reach an accepting state, since all traces reaching an accepting state will loop there infinitely often. Since only a finite prefix of the trace is relevant for obtaining the counter-example of a safety property, the monitor thus outputs a counterexample if it witnesses a trace that leads to an accepting state in the Büchi automata $\mathcal{A}_{\neg\phi}$.

Checking Liveness Properties. The Büchi automata of the negation of ϕ accepts a trace τ if and only if τ visits an accepting state of $\mathcal{A}_{\neg\phi}$ "infinitely often" (e.g., state 2 in Figure 3.1). For instance, an infinite trace $a, o, (v)^\omega$ in which $v \neq n$ will be accepted by $\mathcal{A}_{\neg\phi}$. Formally, such a trace has the form $\tau = \tau_1(\tau_2)^\omega$ ($|\tau_2| \neq 0$), where τ_1 starts in an initial state of the Büchi automata $\mathcal{A}_{\neg\phi}$ and runs until an accepting state s of $\mathcal{A}_{\neg\phi}$, and τ_2 runs from the accepting state s back to itself. Witnessing a trace $\tau = \tau_1(\tau_2)^\omega$ in which τ_2 occurs "infinitely many times" is difficult in practice, since a fuzz campaign visits program executions which are necessarily of finite length. A straightforward approach to tackle this difficulty is to detect a loop in the trace and terminate execution when witnessing the loop occurs m times, e.g., $\tau = \tau_1, \overbrace{\tau_2\tau_2\cdots}^{m}$. This approach is insufficient because witnessing τ_2 for m times does not guarantee

τ_2 occurs infinitely often, for instance `for (i=0; i<m+2; i++){... $\tau_2$...}` may generate τ_2 for m times but stops generating τ_2 after `i==m+1`.

In this work, we record program states when events associated with atomic propositions occur in the execution and detect a state loop in the witnessed trace. If the execution of the state loop produces τ_2, that means, trace τ_2 can be generated infinitely many times by repeatedly going through the state loop. As a result, we assume that the witnessed trace can be extended to an infinite $\tau_1(\tau_2)^\omega$ shaped trace. Consider the following two sequences witnessed in the execution

$$\tau_e = e_0 e_1 \cdots e_i e_{i+1} \cdots e_{i+h} e_i \cdots e_{i+h}$$
$$\tau_s = s_0 s_1 \cdots \overbrace{s_i s_{i+1} \cdots s_{i+h}}^{loop\,body} s_{i+h+1} \cdots s_{i+2h}$$

where τ_e is a sequence of events associated with atomic propositions that occur in the execution and τ_s is a sequence of program states that are recorded when events occur, for instance, s_i indicates the program state that is recorded when the event e_i occurs. Suppose s_i is identical to s_{i+h+1}, then $s_i \cdots s_{i+h+1}$ is a state loop and its loop body is $s_i \cdots s_{i+h}$. Whenever s_i takes input $I_{s_i \cdots s_{i+h+1}}$ that leads to s_i from s_{i+h+1}, s_i will transition to s_i itself. We assume that the system under test is a reactive system taking a sequence of inputs and it is deterministic, that is, the same input always leads to the same program behavior in the execution. Thus, $e_i e_{i+1} \cdots e_{i+h}$ can be generated infinitely many times by repeatedly executing input $I_{s_i \cdots s_{i+h+1}}$ on state s_i. Trace $\tau_e = e_0 \cdots e_{i-1}(e_i \cdots e_{i+h})^\omega$ can be generated by running input $I_{s_0 \cdots s_i}(I_{s_i \cdots s_{i+h+1}})^\omega$, where $I_{s_0 \cdots s_i}$ is an input that leads to state s_i from s_0 and $I_{s_i \cdots s_{i+h+1}}$ is an input that leads to s_i from s_{i+h+1}.

As explained, the occurrence of a state loop in the execution is evidence that the witnessed trace can be extended to an infinite $\tau_1(\tau_2)^\omega$ shaped trace. We leverage this idea to find a violation of a liveness property. When witnessing a trace in the execution that can be extended to a $\tau_1(\tau_2)^\omega$ shaped trace that is accepted by Büchi automata $\mathcal{A}_{\neg\phi}$, we consider a violation of the liveness property has been found. Hence, for liveness property guided fuzzing, we enrich the program transformation of P to P' as follows: (1) instrumenting a function call that records the current program state when an event appears in a transition label of $\mathcal{A}_{\neg\phi}$ occurs in the execution (shown in Code 3.1–Code 3.4)—specifically, function call `record_state()` takes the current program state and generates a hash code for the state at runtime;

(2) instrumenting event-generating and state-recording statements at the entries of `for` and `while` loop statements in the program to observe possible loops in fuzzing. Code 3.4 shows the instrumentation of a `for` loop statement in Pure-FTPd. More detailed and specific optimizations about state saving for checking liveness properties appear in Section 3.5.

3.2.3 Witnessing Event Sequences

Since program P', generated in the previous step, reports a failure when ϕ is violated, we can find a counterexample for ϕ by fuzzing P'. An input that leads to such a failure is a counterexample. However, finding an input of this kind is challenging because it has to generate an execution in which certain events occur in a specific order. In our running example of Pure-FTPd, the quota mechanism must be activated first in the execution, then *user_quota* must be exceeded, and finally the execution must enter a loop in which no *msg_quota_exceeded* is sent back to the client. Existing directed fuzzing approaches like AFLGo [21] aim to direct fuzzing towards a particular program location and cannot drive execution through multiple program locations in a specific order. We now discuss our Büchi automata guided fuzzing in the next section.

3.3 Büchi Automata Guided Fuzzing

Given an LTL property ϕ to be checked, automata-theoretic model checking of LTL properties [150] constructs the Büchi automata $\mathcal{A}_{\neg\phi}$ accepting all traces satisfying $\neg\phi$. In this section, we will discuss how $\mathcal{A}_{\neg\phi}$ can be used to guide fuzzing. First, we design a mechanism to generate an input whose execution passes through multiple program locations in a specific order. We design this mechanism by augmenting a greybox fuzzer in two ways.

- Power scheduling. During fuzzing, the power scheduling component tends to select seeds *closer* to the target on the pre-built inter-procedural control flow graph. Thus, the target can be reached efficiently. To achieve this, we use the fuzzing algorithm of AFLGo [21].

- Input prefix saving. This component observes execution and records input elements that have been consumed when reaching a target.

As mentioned, we focus on fuzzing reactive systems that take a sequence of inputs. The mechanism we follow involves directing fuzzing toward multiple program locations in a specific order. Consider a sequence of program locations $l_1, l_2 \cdots l_m$. Our approach works as follows: first, it takes l_1 as the first target and focuses on generating an input that leads to l_1. Meanwhile, it observes execution and records the prefix i_1 that leads to l_1. Next, it takes l_2 as the target and focuses on exploring the space of inputs starting with prefix i_1, i.e., keeping generating inputs starting with i_1. As a result, an input that reaches l_2 via l_1 can be generated.

Based on the above mechanism of visiting a sequence of program locations, we develop an automata-guided fuzzing approach. The approach uses the Büchi automata $\mathcal{A}_{\neg\phi}$ instrumented in program P' and observes the progress that each trace makes on $\mathcal{A}_{\neg\phi}$ at runtime, e.g., how many state transitions are made towards the accepting state. To guide fuzzing, the approach saves the progress each input achieves on $\mathcal{A}_{\neg\phi}$ and uses it to generate inputs that make further progress. Specifically, it saves the progress for each input by recording state transitions that are executed on $\mathcal{A}_{\neg\phi}$ and the input prefix that leads to those transitions. Consider input i_0 and its trace τ_0 goes from initial state s_0 to state s_m on automata $\mathcal{A}_{\neg\phi}$. The achieved progress is represented as a tuple $\langle x_0^i, x_0^s \rangle$, where x_0^i is the shortest prefix of i_0 whose execution trace goes from s_0 to s_m and x_0^s is the state transition sequence $s_0 \cdots s_m$ visited. Such *progress* tuples are stored in a set $\mathcal{X}$ and are used to guide fuzzing.

For input generation, the approach takes a tuple from $\mathcal{X}$ and uses it to generate inputs that make further progress. Consider a tuple $\langle x^i, x^s \rangle$: x^s records state transitions on automata $\mathcal{A}_{\neg\phi}$ which input prefix x^i has led to. Thus, we can query $\mathcal{A}_{\neg\phi}$ with x^s to find a transition that makes further progress, i.e., a state transition that gets closer to an accepting state of $\mathcal{A}_{\neg\phi}$. In the example, assuming x^s is state 0 in Figure 3.1, then the transition from state 0 to state 1 will be identified since state 1 is closer to the accepting state 2. Suppose t is the next progressive state transition of x^s, then we can further query $\mathcal{A}_{\neg\phi}$ to obtain atomic propositions that trigger transition t. Then, by querying the map between atomic propositions and program locations, we can identify program locations for those atomic propositions.

In the example, an atomic proposition a triggers the transition from state 0 to state 1 and its corresponding program location is $\langle ftpd.c, 6072 \rangle$, as shown in Table 3.1.

From the above we can define criteria for input to make further progress: (1) its execution has to follow the path that an input prefix x^i has gone through such that the generated trace can go through state transitions x^s; and, (2) subsequently the execution reaches one of program locations that are identified above to ensure the generated trace takes a step further in $\mathcal{A}_{\neg\phi}$.

To generate inputs of this kind, our mechanism for generating inputs that traverse a sequence of program locations in a specific order comes into play. Assume l_i is one of the program locations identified above, for making further "progress" in $\mathcal{A}_{\neg\phi}$. The mechanism takes l_i as the target and keeps generating inputs that start with the prefix x^i until generating an input that starts with the prefix x^i and subsequently visits location l_i. This is how our approach uses tuples in $\mathcal{X}$ to generate inputs that make further "progress" towards an accepting state in the Büchi automata $\mathcal{A}_{\neg\phi}$. The detailed fuzzing algorithm is now presented.

3.4 Fuzzing Algorithm

Algorithm 2 shows the workflow of our counterexample-guided fuzzing. To find a counterexample, the algorithm guides fuzzing in two dimensions. First, it prioritizes the exploration of inputs whose execution traces are more likely to be accepted by $\mathcal{A}_{\neg\phi}$. Specifically, if the trace of the prefix of an input reaches a state that is closer to an accepting state on $\mathcal{A}_{\neg\phi}$, then its trace is more likely to be accepted. The algorithm selects input prefixes whose traces have been witnessed to get close to an accepting state and keeps generating inputs starting with them (shown in line 5 and line 10). Secondly, the algorithm focuses on generating inputs whose execution makes further progress on $\mathcal{A}_{\neg\phi}$. Given an input prefix, the algorithm finds a state transition t that helps us get closer to an accepting state in $\mathcal{A}_{\neg\phi}$, and finds the atomic propositions which enable t to be taken (line 6). For the atomic propositions enabling transition t, we identify the corresponding program locations (line 7). Then we attempt to generate inputs that reach the program location in the execution and trigger the program behavior associated with the atomic proposition. As a result, the generated trace can make further progress in $\mathcal{A}_{\neg\phi}$. To generate inputs that reach

Algorithm 2: Counterexample-Guided Fuzzing

Input: P': The transformation of program under test
Input: $\mathcal{A}_{\neg\phi}$: Automata of negation of property under test
Input: *map*: Map between propositions and program locations
Input: $flag$: True for liveness properties
Input: *total_time*: Time budget for fuzzing
Input: *target_time*: Time budget for reaching a program location

1: **Procedure** `Fuzz`(P', $\mathcal{A}_{\neg\phi}$, *map*, $flag$, *total_time*, *target_time*)
2: $s_0 \leftarrow$ `getInitState`($\mathcal{A}_{\neg\phi}$) ;
3: $\mathcal{X} \leftarrow \{\langle\emptyset, s_0\rangle\}$ ▷ Starting with init state of $\mathcal{A}_{\neg\phi}$
4: **for** $time < total\_time$ **do**
5: $\langle x_t^i, x_t^s\rangle \leftarrow$ `selectPrefix`($\mathcal{X}$) ;
6: $p \leftarrow$ `selectTargetAtomicProposition`($\mathcal{A}_{\neg\phi}, x_t^s$) ;
7: $l \leftarrow$ `selectProgramLocationTarget`(map, p) ;
8: **for** $time' < target\_time$ **do**
// $\mathcal{D}$: Feedback of CFG distance
// S_{power}: Power schedule algorithm
9: $I \leftarrow$ `generateInput`($\mathcal{D}, S_{power}$) ;
10: $I' \leftarrow$ `replacePrefix`(I, x_t^i) ;
11: $d, \langle x^i, x^s\rangle \leftarrow$ `evaluate`($P', I', flag$);
12: $\mathcal{D} \leftarrow \mathcal{D} \cup \{d\}$;
13: $\mathcal{X} \leftarrow \mathcal{X} \cup \{\langle x^i, x^s\rangle\}$

a particular program location, we leverage the algorithm proposed in AFLGo (lines 8–14). Its idea is to assign more power to seeds that are *closer* to the target on a pre-built control flow graph such that the generated inputs are more likely to reach the target. The time budget for reaching a target is configurable, via parameter *target_time*.

For prefix selection (line 5), the algorithm defines a fitness function to compute a fitness value for each prefix tuple. Given a tuple $\langle x_t^i, x_t^s\rangle$, its fitness value is

$$f_t = \frac{l_s}{l_s + l_a} + \frac{1}{l_i}$$

where l_s is the length of x_t^s and l_a is the length of the shortest path from the last state of x_t^s to an accepting state on $\mathcal{A}_{\neg\phi}$ and l_i is the length of input prefix x_t^i. As shown in the formula, a prefix tuple has a higher fitness value if the last state of x_t^s is closer to an accepting state on $\mathcal{A}_{\neg\phi}$ and the input prefix is shorter. Heuristically, by extending such a prefix, our fuzzing algorithm is more likely to generate an input

whose execution trace is accepted by $\mathcal{A}_{\neg\phi}$. Prefix tuples with higher fitness values are prioritized for selection.

For atomic proposition selection (line 6), we adopt a random selection strategy. Consider tuple $\langle x_t^i, x_t^s \rangle$ and the last state of x_t^s is s_t, the algorithm identifies atomic propositions that make a progressive transition from s_t on $\mathcal{A}_{\neg\phi}$ as follows: if state s_t is not an accepting state of $\mathcal{A}_{\neg\phi}$, any atomic proposition that triggers a transition from s_t towards an accepting state is selected. If state s_t is an accepting state, any atomic proposition that triggers a transition from s_t back to itself is selected. For simplicity, the algorithm randomly selects one from the identified atomic propositions. When the selected proposition p has multiple associated program locations, we randomly select one of them as a target. The main consideration for adopting a random strategy is to keep our technique as simple as possible. Moreover, these strategies can be configured in our tool.

3.5 State Saving

```
4315  if(...(max_filesize >= (off_t) 0 &&
            (max_filesize=user_quota_size - quota.size)
            < (off_t) 0 )){
         ...
4322     goto afterquota;
4323  }
```

Code 3.5: Quota checking:<ftpd.c. 4315>.

In liveness property verification, LTL-Fuzzer detects a state loop in the witnessed trace. If a state loop is detected, LTL-Fuzzer assumes the current trace can be extended to a lasso-shaped trace $\tau_1(\tau_2)^\omega$. This works with a *concrete* representation of program states, however in reality state representations of software implementations are always abstracted. State representations that are too abstract may miss capturing variable states that are relevant to the loop, which leads to false positives. State representations that are too concrete may contain variable states that are irrelevant to the loop, such as a variable for the system clock, which leads to false negatives. To be practical, LTL-Fuzzer takes a snapshot of the application's registers and *addressable* memory and hashes it into a 32-bit integer,

which is recorded as a state. Addressable memory indicates two kinds of objects: (1) global variables (2) objects that are explicitly allocated with functions `malloc()` and `alloca()`. Such a convention was also adopted in previous works on infinite loop detection [28, 138].

Furthermore, LTL-Fuzzer only records a program state for selected program locations, not for all program locations. Specifically, we only save states for the program locations associated with the transition labels of the automata $\mathcal{A}_{\neg\phi}$ where ϕ is the liveness property being checked. Note that a transition label in $\mathcal{A}_{\neg\phi}$ is a subset of atomic propositions [151, 150]. The full set of atomic propositions is constructed by taking the atomic propositions appearing in ϕ and embellishing this set with atomic propositions that we introduce for the occurrence of each program loop header (such as l in Table 3.1). If the transition label involves a set L of atomic propositions, we track states for only those atomic propositions in L which correspond to loop header occurrences. The goal here is to quickly find possible infinite loops by looking for a loop header being visited with the same program state. Hence for the transition label $!n$ in our running example, we only store states for the program locations corresponding atomic proposition l in Table 3.1.

In the example shown in Section 3.2, LTL-Fuzzer witnesses a state generated at program location $\langle ftpd.c, 4067\rangle$ (shown in Code 3.4) that has been observed before and at the same time the witnessed trace is accepted by $\mathcal{A}_{\neg\phi}$. In this case, LTL-Fuzzer reports a violation of the LTL property ϕ shown in Page 20. To validate if the violation is spurious, we check if the observed state loop can be repeated in the execution. Our analysis shows that a chunk of data was read during the execution of the state loop and the chunk of data was from a file uploaded by the client. We duplicated the chunk of data in the uploaded file and reran the experiment, and found the state loop was repeated. That means the witnessed trace can be extended to a $\tau_1(\tau_2)^\omega$ shaped trace, which visits the accepting state of the automata accepting $\neg\phi$ (Figure 3.1) infinitely many times. Thus, the reported violation is not spurious.

We further analyzed the root cause of the violation. It shows there was a logical bug in the quota-checking module. As shown in Code 3.5, the assignment of `max_filesize` occurs in a conditional statement and is never executed because `max_filesize`'s initial value is -1. To fix the bug, we created a patch and submitted

a pull request on the Github repo of Pure-FTPd, which has been confirmed and verified.

3.6 LTL-Fuzzer Implementation

We implement LTL-Fuzzer as an open-source tool built on top of AFL, which comprises two main components: instrumentor and fuzzer. In the following, we explain these components.

3.6.1 Instrumentation Module

AFL comes with a special compiler pass for `clang` that instruments every branch instruction to enable coverage feedback. By extending this compiler, we instrument a program under test at three levels: specific locations, basic blocks, and the application.

Specific locations. LTL-Fuzzer takes a list of program locations at which program behaviors associated with a property under test might occur. At each of the given program locations, the instrumentation module injects two components: *event generator* and *state recorder*. Event generator is a piece of code that generates an event when the provided condition is satisfied at run-time. The state recorder is a component that takes a snapshot of program states and generates a hash code for the state when the given program location is reached in the execution.

Basic blocks. LTL-Fuzzer guides fuzzing to a target using the feedback on how close to the target input is, as explained in Section 3.3. At runtime, LTL-Fuzzer requires the distance from each basic block to the target on the CFG (control flow graph). The instrumentor instruments a function call in each basic block at runtime. The function call will query a table that stores distances from each block to program locations associated with the given property (i.e., targets). The distance from a basic block to each program location is computed offline with the distance calculator component that is borrowed from AFLGo [21].

Applications. For a program under test, the instrumentation module injects a *monitor* into the program. During fuzzing, the monitor collects events generated by instrumented event generators and produces execution traces. For property checking, the monitor leverages Spot libraries [143] to generate a Büchi automata from the

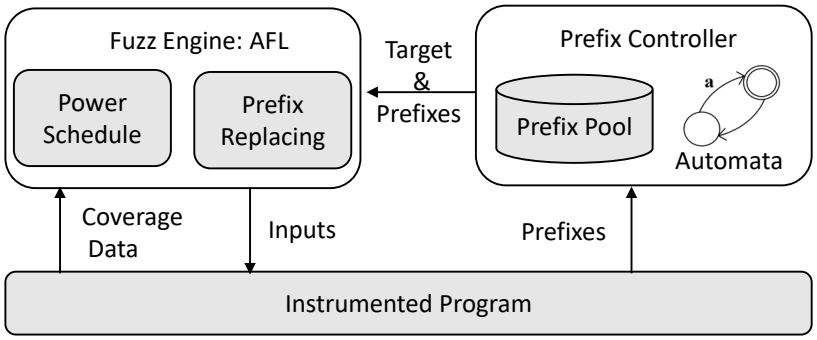


Figure 3.2: The architecture of LTL-Fuzzer.

negation of an LTL property and validates these traces. The instrumentation module also instruments an *observer* in the program that monitors the execution of inputs; it maps a given suitable execution trace prefix to the input event sequence producing it so that the occurrence of the prefix can be detected by the observer, during fuzzing. The fuzzing process then seeks to further extend this prefix with "suitable" events as described in the following.

3.6.2 Fuzzer

Figure 3.2 shows the Fuzzer component's architecture. It mainly comprises two modules: *prefix controller* and *fuzz engine*. LTL-Fuzzer saves input prefixes whose execution traces make transitions on the automata and reuse them for further exploration (Section 3.3). At runtime, the prefix controller conducts three tasks: (1) collecting prefixes reported by the monitor instrumented in the program under test and storing them into a pool; (2) selecting a prefix from the pool for further exploration according to Algorithm 2; (3) identifying the target program location based on the selected prefix. The fuzz engine is obtained by modifying AFL [167]. It generates inputs starting with a given input prefix. To reach a target, our fuzzer integrates the power scheduling component developed in AFLGo [21] to direct fuzzing. In LTL-Fuzzer, we direct execution to reach a target after the execution of an input prefix. Thus, the fuzz engine collects no feedback, such as coverage data, during the execution of the input prefix and only collects feedback data after the execution of the input prefix is completed.

3.7 Evaluation

In our experiments, we seek to answer the following questions:

RQ.1 Effectiveness: How effective is LTL-Fuzzer at finding LTL property violations?

RQ.2 Comparison: How does LTL-Fuzzer compare to the state-of-the-art validation tools in terms of finding LTL property violations?

RQ.3 Usefulness: How useful is LTL-Fuzzer in revealing LTL property violations in real-world systems?

3.7.1 Subject Programs

Table 3.2 lists the subject programs used in our evaluation. This includes 7 open-source software projects that implement 6 widely-used network protocols. We selected these projects because they (1) are reactive software systems that LTL-Fuzzer is designed for, (2) include appropriate specification documents from which LTL properties can be generated, and (3) are widely used and have been studied. Finding bugs in such real-world systems is thus valuable.

Table 3.2: Detailed information about our subject programs.

Project	Protocol	#SLOC	InPreviousWork	GithubStars
ProFTPD	FTP	210.8k	[115]	339
Pure-FTPd	FTP	52.9k	[115]	435
Live555	RTSP	52.5k	[126] [115]	526
OpenSSL	TLS	286.7k	[73] [115] [43]	16.3K
OpenSSH	SSH	98.3k	[60] [115]	1.5K
TinyDTLS	DTLS	63.2k	[58] [115]	43
Contiki-Telnet	TELNET	353.4k	[73]	3.4K

3.7.2 Experimental Setup

To answer the research questions, we conducted three empirical studies on the subject programs.

Table 3.3: Statistics of violations found on the LTL-property set.

Prop	CVE-ID	Type of Vulnerability	Program	Version
PrF_1	CVE-2019-18217	Infinite Loop	ProFTPD	1.3.6
PrF_2	CVE-2019-12815	Illegal File Copy	ProFTPD	1.3.5
PrF_3	CVE-2015-3306	Improper Access Control	ProFTPD	1.3.5
PrF_4	CVE-2010-3867	Illegal Path Traversal	ProFTPD	1.3.3
LV_1	CVE-2019-6256	Improper Condition Handle	Live555	2018.10.17
LV_2	CVE-2019-15232	Use after Free	Live555	2019.02.03
LV_3	CVE-2019-7314	Use after Free	Live555	2018.08.26
LV_4	CVE-2013-6934	Numeric Errors	Live555	2013.11.26
LV_5	CVE-2013-6933	Improper Operation Limit	Live555	2011.12.23
SH_1	CVE-2018-15473	User Enumeration	OpenSSH	7.7p1
SH_2	CVE-2016-6210	User Information Exposure	OpenSSH	7.2p2
SL_1	CVE-2016-6309	Use after Free	OpenSSL	1.1.0a
SL_2	CVE-2016-6305	Infinite Loop	OpenSSL	1.1.0
SL_3	CVE-2014-0160	Illegal Memory Access	OpenSSL	1.0.1f

3.7.2.1 Effectiveness of LTL-Fuzzer

We evaluate LTL-Fuzzer's effectiveness by running it on a set of LTL properties in subject programs where violations are already known; we check the number of LTL properties for which LTL-Fuzzer can find violations. To create such a dataset, we collect event ordering-related CVEs (so that they can be captured as a temporal property) that are disclosed in subject programs, e.g., an FTP client copies files from the server without logging in successfully. Specifically, for each subject, we select 10 such CVEs with criteria: (1) reported recently (during 2010-2020); (2) include instructions to reproduce the bug; (3) relevant to event orderings. Then we manually reproduce them with the corresponding version of the code. If a CVE is reproducible, then we write the property in LTL and put it in our dataset of LTL properties. Based on the aforementioned criteria, we collected 14 CVEs in 7 subjects as shown in Table 3.4; these LTL properties can be found in our dataset[4] and the appendix of our arxiv paper.[5] Our goal is to check experimentally if LTL-Fuzzer can find violations of these LTL properties.

[4] https://github.com/ltlfuzzer/LTL-Fuzzer/tree/main/ltl-property
[5] https://arxiv.org/abs/2109.02312

Table 3.4: Performance of four tools in finding the violations of LTL properties.

Prop	LTL-Fuzzer	AFL_{LTL}		AFLGo		L+NuSMV	
	Time(h)	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$
PrF_1	4.62	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
PrF_2	0.95	2.01	**0.84**	T/O	**1.00**	T/O	**1.00**
PrF_3	1.14	1.89	**0.76**	T/O	**1.00**	T/O	**1.00**
PrF_4	2.06	5.17	**0.85**	T/O	**1.00**	T/O	**1.00**
LV_1	5.29	11.13	**1.00**	11.47	**1.00**	T/O	**1.00**
LV_2	0.22	1.42	**0.91**	1.46	**0.92**	T/O	**1.00**
LV_3	1.27	4.18	**0.98**	T/O	**1.00**	T/O	**1.00**
LV_4	2.73	2.58	0.40	2.21	0.39	T/O	**1.00**
LV_5	1.80	1.99	0.63	1.45	0.33	T/O	**1.00**
SH_1	0.18	0.17	0.44	T/O	**1.00**	24.00	**1.00**
SH_2	0.19	0.19	0.50	T/O	**1.00**	24.00	**1.00**
SL_1	3.77	6.00	**0.74**	6.58	**0.82**	T/O	**1.00**
SL_2	1.45	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
SL_3	1.11	7.31	**1.00**	T/O	**1.00**	T/O	**1.00**
Total Violations	14	12		5		2	
Average Time (hours)	1.91	6.57		17.08		24.00	
Speed-Up	—	3.44×		8.93×		12.55×	

[1] T/O represents tools cannot expose vulnerabilities within 24 hours for 10 experimental runs. We replace T/O with 24 hours when calculating average usage time.
[2] Statistically significant values of $\hat{A}_{12}$ are shown in bold.

3.7.2.2 Comparison with Other Tools

We evaluate LTL-Fuzzer and state-of-the-art techniques on the LTL property dataset above and compare them in terms of the number of LTL properties for which each technique finds the violations and the time that is used to find a violation. For state-of-the-art techniques, we reviewed recent and well-known techniques in model checking, runtime verification, and directed fuzzing domains. We chose the following techniques for comparison with LTL-Fuzzer.

- AFLGo [21]. It is a well-known directed greybox fuzzer that drives execution to a target with a simulated annealing-based power schedule that assigns more energy to inputs that hold the trace closer to the target. We take it as a baseline tool.

- AFL_{LTL}. It is an implementation that enables AFLGo to detect an LTL property violation. Specifically, AFL_{LTL} powers AFLGo with only the LTL test oracle such that it can report an error when the given LTL property is violated

in the execution. By comparing with AFL_{LTL}, we evaluate how effective our automata-guided fuzzing strategy is in finding LTL property violations. Note that AFL_{LTL} is also a tool built by us, but it lacks the automata-guided fuzzing of LTL-Fuzzer.

- L+NuSMV. It combines model learning and model checking to verify properties in a software system. Specifically, it leverages a learning library called LearnLib [77] to build a model for the software system and then verifies given properties on the learned model with the well-known model checker NuSMV [35]. In the paper, we indicate it with L+NuSMV. This technique was published at CAV 2016 [59] and has been subsequently adopted in recent works such as [156] and [58].

We briefly summarize why we did not include certain other model-checkers and fuzzers, and all runtime verification tools for comparison. Model checking tools CBMC [38], CPAChecker [18], Seahorn [66], SMACK [133], UAutomizer [69], DIVINE [15] cannot support LTL property verification. Schemmel's work [138], published at CAV 2018 partially supports LTL property verification. SPIN [72] supports LTL property verification but only works with the modeling language Promela [129], and the tool provided in SPIN for extracting models from C programs failed to work on our subject programs. Some model checking tools [73, 139], and directed fuzzing tools (like UAFL [155], Hawkeye [31], and TOFU [158]) we reviewed, are not publicly available.

Finally, all of the available runtime verification tools [54] (like JavaMOP [80], MarQ [136], and Mufin [44]) cannot check LTL properties in C/C++ software systems. Furthermore, our method is conceptually different and complementary to runtime verification—our method generates test executions, while runtime verification checks a test execution. While the combination of our method with runtime verification is possible, a comparison is less meaningful.

3.7.2.3 Real-World Utility

In this study, we read the RFC specifications that these subject programs follow to extract temporal properties and describe them in LTL. Then we use LTL-Fuzzer to check these properties on the subject programs.

Configuration Parameters. Following fuzzing evaluation suggestions from the community [89], we run each technique for 24 hours and repeat each experiment 10 times to achieve statistically significant results. For the initial seeds, we use seed inputs provided in ProFuzzBench [115] for all subjects. ProFuzzBench is a benchmark for stateful fuzzing of network protocols, which contains a suite of representative open-source network protocol implementations. For Contiki-Telnet, which is not contained in ProFuzzBench, we generate random inputs as its initial seeds. For LTL-Fuzzer, we need to specify the time budget for reaching a single program location, and we configure it with 45 minutes for each target. For AFLGo and AFL_{LTL}, we need to provide a target for an LTL property being checked. We specify the target by randomly selecting from program locations that are associated with atomic propositions that trigger the transition to an accepting state on the automata of the negation of the property. In the example in Section 3.2, we chose one of the loop entries as the target since proposition o triggers the transition to the accepting state shown in Figure 3.1 and it corresponds with the loop entries. For execution environments, we conducted experiments on a physical machine with 64 GB RAM and a 56-core Intel(R) Xeon(R) E5-2660 v4 CPU, running a 64-bit Ubuntu TLS 18.04 as the operating system.

3.7.3 Experimental Results

3.7.3.1 Effectiveness (RQ.1)

Table 3.4 shows property violations found by LTL-Fuzzer for the 14 LTL properties derived from known CVEs. The first column shows identifiers of the properties being checked. The corresponding LTL properties and their descriptions can be found in our dataset. Columns 2–5 represent CVE-IDs, types of vulnerabilities that CVEs represent, subject names, and subject versions, respectively. Column "LTL-Fuzzer" shows the time that is used to find a violation by LTL-Fuzzer. As shown in Table 3.4, LTL-Fuzzer can effectively detect violations of LTL properties in the subjects. It successfully detected the violation for all 14 LTL properties in the dataset. On average, it took LTL-Fuzzer 1.91 hours to find a violation.

LTL-Fuzzer is found to be effective in finding LTL property violations, detecting violations for all 14 properties derived from known CVEs.

3.7.3.2 Comparison (RQ.2)

As shown in Table 3.4, the last three main columns show the time that is used for comparison techniques to find a violation on the 14 LTL properties in the experiment. Note that "T/O" indicates a technique failed to find the violation for an LTL property in the given time budget (i.e., 24 hours). To mitigate randomness in fuzzing, we adopted the Vargha-Delaney statistic $\hat{A}_{12}$ [152] to evaluate whether one tool significantly outperforms another in terms of the time that is used to find a violation. The $\hat{A}_{12}$ is a non-parametric measure of effect size and gives the probability that a randomly chosen value from data group 1 is higher or lower than one from data group 2. It is commonly used to evaluate whether the difference between two groups of data is significant. Moreover, we also use the Mann-Whitney U test to measure the statistical significance of performance gain. When it is significant (taking 0.05 as a significance level), we mark the $\hat{A}_{12}$ values in bold.

LTL-Fuzzer found violations of all of the 14 LTL properties, followed by AFL_{LTL} (12), AFLGo (5), and L+NuSMV (2). We note that AFL_{LTL} is also a tool built by us; it partially embodies the ideas in LTL-Fuzzer and is meant to help us understand the benefits of automata-guided fuzzing. In terms of the time that is used to find a violation, LTL-Fuzzer is the fastest (1.91 hours), followed by AFL_{LTL} (6.57 hours), AFLGo (17.08 hours), and L+NuSMV (24.00 hours). In other words, LTL-Fuzzer is 3.44x, 8.93x, 12.55x faster than AFL_{LTL}, AFLGo, and L+NuSMV, respectively. For CVE-2013-6934 and CVE-2013-6933, AFLGo performed slightly better than other techniques, while AFL_{LTL} exhibited the same performance as LTL-Fuzzer for CVE-2018-15473 and CVE-2016-6210. We investigated these 4 CVEs and found that triggering those vulnerabilities is relatively straightforward. They can be triggered without sophisticated directing strategies. As a result, other techniques achieve a slightly better performance than LTL-Fuzzer for these four CVEs. In terms of the $\hat{A}_{12}$ statistic, LTL-Fuzzer performs significantly better than other techniques in most cases.

LTL-Fuzzer found violations of all 14 LTL properties in the experiment. AFL_{LTL}, AFLGo, and L+NuSMV found 12, 5, and 2 property violations, respectively. LTL-Fuzzer is 3.44x, 8.93x, 12.55x faster than AFL_{LTL}, AFLGo, and L+NuSMV.

3.7.3.3 Real-World Utility (RQ.3)

In this study, we evaluate the utility of LTL-Fuzzer by checking whether it can find zero-day bugs in real-world protocol implementations. We extract 50 properties from RFCs that our subject programs follow (aided by comments in the source code of the programs) and write them in linear-time temporal logic. The details of the 50 LTL properties can be found in our dataset. In the experiment, LTL-Fuzzer achieved a promising result. Out of these 50 LTL properties, LTL-Fuzzer discovered *new* violations for 15 properties, which are shown in Table 3.5. We reported these 15 zero-day bugs to developers and all of them got confirmed by developers. We reported them on the Common Vulnerabilities and Exposures (CVE) system (see https://cve.mitre.org/) and 12 of them were assigned CVE IDs. Out of 15 reported violations, 7 have been fixed at the time of the submission of our paper. Notably, LTL-Fuzzer shows effectiveness in finding violations for liveness properties. In the experiment, LTL-Fuzzer successfully found violations for 4 liveness properties which are PrF_1, SL_2, TD_1, and PuF_5. All 4 violations were confirmed by developers, i.e., they are not spurious results. Moreover, to discover violations for these 4 liveness properties, LTL-Fuzzer only recorded 6, 11, 4, and 9 states, respectively. Since every state is recorded as a 32-bit integer, the memory consumption for recording states is thus found to be negligible in our experiments.

Among 50 LTL properties extracted from protocol RFCs, LTL-Fuzzer found 15 previously unknown violations in protocol implementations, and 12 of these have been assigned CVEs.

3.7.4 Threats to Validity

There are potential threats to the validity of our experimental results. One concern is *external validity*, i.e., the degree to which our results can be generalized

to and across other subjects. To mitigate this concern, we selected protocol implementations that are widely used and have been frequently evaluated in previous research (as shown in Table 3.2). We may have made mistakes in converting informal requirements into LTL properties. To reduce this kind of bias, we let two authors check generated properties and remove those on which they do not agree, or do not think are important properties.

In principle, LTL-Fuzzer can report false positives due to incorrect instrumentation, e.g., if we fail to instrument some target locations for an atomic proposition. We mitigate the risk of false positives by checking the reported counterexamples and validating that they are true violations of the temporal property being checked. We add here that we did not encounter such false positives in any of our experiments.

Another concern is *internal validity*, i.e., the degree to which our results minimize systematic error. First, to mitigate spurious observations due to the randomness in the fuzzers and to gain statistical significance, we repeated each experiment 10 times and reported the Vargha-Delaney statistic $\hat{A}_{12}$. Secondly, our LTL-Fuzzer implementation may contain errors. To facilitate scrutiny, we make LTL-Fuzzer code available.

3.8 Conclusion

In this chapter, we present LTL-Fuzzer, a linear-time temporal logic guided grey-box fuzzing technique, which takes Linear-time Temporal Logic (LTL) properties extracted from informal requirements such as RFCs and finds violations of these properties in C/C++ software implementations. Our evaluation shows that LTL-Fuzzer is effective in finding property violations. It detected 15 LTL property violations in real-world protocol implementations that were previously unknown; 12 of these zero-day bugs have been assigned CVEs. We make the dataset of LTL properties and our tool available for scrutiny. Arguably, we could compare LTL-Fuzzer with more model checkers and fuzzers, experimentally. At the same time, we have noted that many model checkers were found to be not applicable for checking arbitrary LTL properties of arbitrary C/C++ software implementations. Moreover, the problem addressed by LTL-Fuzzer is certainly beyond the reach of fuzzers since fuzzers cannot detect temporal property violations. Overall, we

believe our work represents a practical advance over model checkers and runtime verification, and a conceptual advance over greybox fuzzers.

Table 3.5: Zero-day Bugs found by LTL-Fuzzer; for several of them CVEs have been assigned but CVE ids are not shown.

Prop	Project	Description of violated properties	Bug Status
TD_1	TinyDTLS (0.9-rc1)	If the server is in the `WAIT_CLIENTHELLO` state and receives a `ClientHello` request with valid cookie and the epoch value 0, must finally give `ServerHello` responses.	CVE-2021-42143, Fixed
TD_2	TinyDTLS (0.9-rc1)	If the server is in `WAIT_CLIENTHELLO` state and receives a `ClientHello` request with valid cookie but not 0 epoch value, must not give `ServerHello` responses before receiving `ClientHello` with 0 epoch value.	CVE-2021-42142, Fixed
TD_3	TinyDTLS (0.9-rc1)	If the server is in the `WAIT_CLIENTHELLO` state and receives a `ClientHello` request with an invalid cookie, must reply `HelloVerifyRequest`.	CVE-2021-42147, Fixed
TD_5	TinyDTLS (0.9-rc1)	If the server is in the `DTLS_HT_CERTIFICATE_REQUEST` state and receives a `Certificate` request, must give a `DTLS_ALERT_HANDSHAKE_FAILURE` or `DTLS_ALERT_DECODE_ERROR` response, or set `Client_Auth` to be verified.	CVE-2021-42145, Fixed
TD_{11}	TinyDTLS (0.9-rc1)	After the server receives a `ClientHello` request without renegotiation extension and gives a `ServerHello` response, then receives a `ClientHello` again, must refuse the renegotiation with an Alert.	Confirmed
TD_{12}	TinyDTLS (0.9-rc1)	After the server receives a `ClientHello` request and gives a `ServerHello` response, then receives a `ClientKeyExchange` request with a different epoch value than that of `ClientHello`, server must not give `ChangeCipherSpec` responses.	CVE-2021-42141, Fixed
TD_{13}	TinyDTLS (0.9-rc1)	After the server receives a `ClientHello` request and gives a `ServerHello` response, then receives a `ClientHello` request with the same epoch value as that of the first one, server must not give `ServerHello`.	CVE-2021-42146
TD_{14}	TinyDTLS (0.9-rc1)	If the server receives a `ClientHello` request and gives a `HelloVerifyRequest` response, and then receives a overlarge packet even with valid cookies, the server must refuse it with an `Alert`.	CVE-2021-42144, Fixed
CT_1	Contiki-Telnet (3.0)	After `WILL` request is received and the corresponding option is disabled, must send `DO` or `DONT` responses.	CVE-2021-40523
CT_2	Contiki-Telnet (3.0)	After `DO` request is received and the corresponding option is disabled, must send `WILL` or `WONT` responses.	Confirmed
CT_7	Contiki-Telnet (3.0)	After `WONT` request is received and the corresponding option is disabled, must not give responses.	CVE-2021-38311
CT_8	Contiki-Telnet (3.0)	After `DONT` request is received and the corresponding option is disabled, must not give responses.	Confirmed
CT_{10}	Contiki-Telnet (3.0)	Before `Disconnection`, must send an Alert to disconnect with clients.	CVE-2021-38387
CT_{11}	Contiki-Telnet (3.0)	If conducting `COMMAND` without `AbortOutput`, the response must be same as the real execution results.	CVE-2021-38386
PuF_5	Pure-FTPd (1.0.49)	When quota mechanism is activated and user quota is exceeded, must finally reply a quota exceed message.	CVE-2021-40524, Fixed

Chapter 4

Testing Stateful Protocols

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 chapter, 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.

4.1 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 the required message structure or order by fuzzing *recorded* message sequences [126, 130, 8, 114]. 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 work, 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 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 [115] demonstrate the effectiveness of the LLM-guided approach: Compared to the baseline (AFLNet [126]) 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 [130]). 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

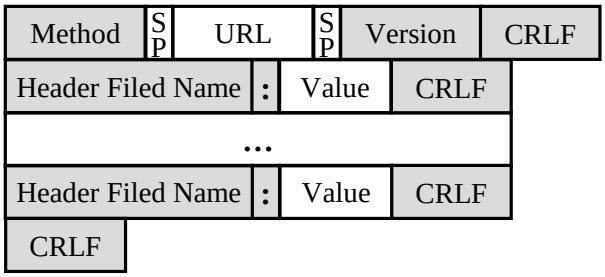
- We conducted experiments that demonstrate that our LLM-guided stateful fuzzer prototype ChatAFL is substantially more effective than the state-of-the-art AFLNet and NSFuzz in terms of the coverage of the protocol state space and the protocol implementation code. Apart from enhanced coverage, ChatAFL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

4.2 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 work. We then provide some background on large language models and our motivation.

4.2.1 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



(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.

Figure 4.1: Structure of RTSP client requests in (a), and a PLAY client request from Live555 in (b).

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 4.1: 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 4.2: 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* [3, 81, 52] 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

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

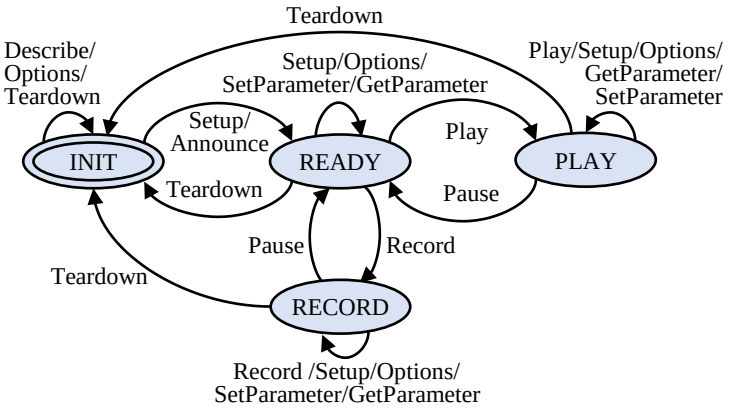


Figure 4.2: The state machine for the RTSP protocol from RFC 2326.

implementation is tedious and error-prone [126].

A *mutation-based protocol fuzzer* [126, 130] 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 [126]. In fact, all recently proposed protocol fuzzers, such as AFLNet [126] and NSFuzz [130] 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.

4.2.2 Large Language Models

Emerging pre-trained Large Language Models (LLMs) have demonstrated impressive performance on natural language tasks, such as text generation [24, 153, 34] and conversations [146, 119]. LLMs have also been proven effective in translating natural language specifications and instructions into executable code [55, 78, 32]. 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 [25]. 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 work, 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.

```
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
```

Figure 4.3: Grammar for the RTSP PLAY client request.

4.3 Case Study: Testing 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 [115]. 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 [126, 8, 130]. 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 4.4.

4.3.1 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 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

[2]Live555 available at http://www.live555.com/
[3]Available at https://platform.openai.com/docs/models/gpt-3-5

fields. Figure 4.3 shows the PLAY message grammar, corresponding to the grammar of the PLAY client request shown in Figure 4.1. 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 4.4, 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.

4.3.2 Enriching 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

[4]We discuss the prompt engineering in Section 4.4.1.

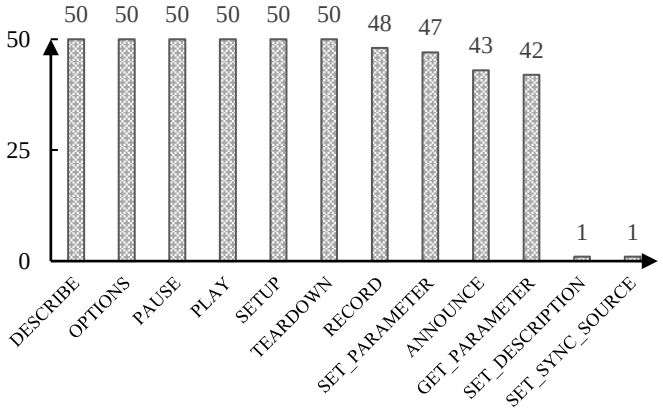


Figure 4.4: Types of client requests in the answer set and the corresponding occurrence times for each type.

absence of the remaining 6 types of client requests leaves a significant portion of the RTSP state machine unexplored, as shown in Figure 4.2. 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 4.2, 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

[5]We discuss the detailed model prompt in Section 4.4.2.

Table 4.1: Processed results of client requests after being sent to the server.

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

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 4.1. Approximately 55% of client requests can be directly 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.

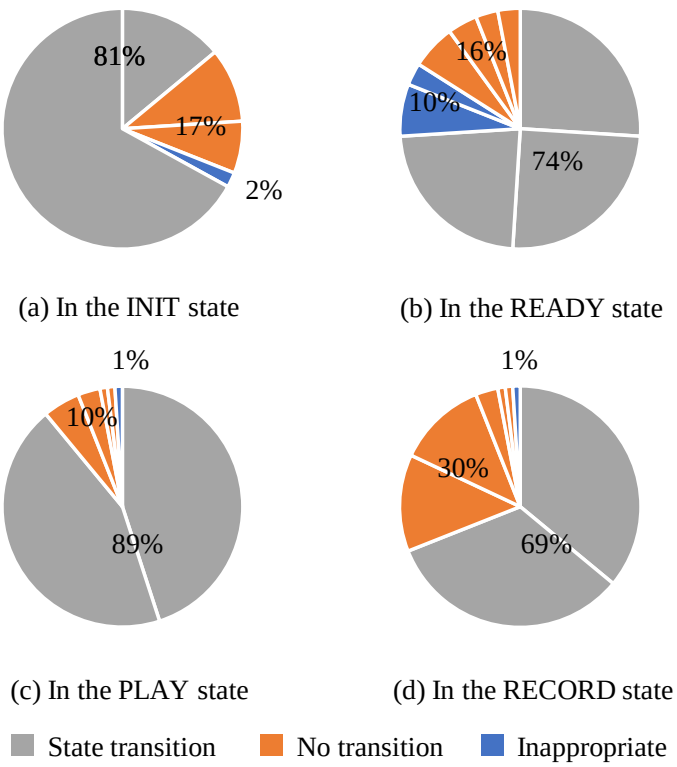


Figure 4.5: The next types of client requests generated by the LLM in each state. The types in gray induce state transitions, the ones in orange appear in the suitable state but do not trigger state transitions, and the ones in blue appear in the inappropriate states. Each segment represents one distinct message type.

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

4.3.3 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 4.5 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 4.5, we can see that there are 81%, 74%, 89%, and 69% client requests, respectively, that can induce state transitions to different states. Additionally, 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.

4.4 LLM-Guided Protocol Fuzzing

Motivated by the impressive capabilities demonstrated by the LLMs in the case study (Section 4.3), we develop LLM-guided protocol fuzzing (LLMPF) to tackle the challenges of existing mutation-based protocol fuzzing (EMPF).

Algorithm 3 (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 $\mathcal{C}_{\text{✗}}$ 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 4.4.1); (2) query the LLM to enrich the initial seeds (line 3) (Section 4.4.2); and (3) leverage the LLM's capability to break out of a coverage plateau (lines 4, 19–21, 26, 30 and 32) (Section 4.4.3). Now we will introduce each component.

4.4.1 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.

4.4.1.1 Grammar Extraction

Before the fuzzer can ask the LLM to generate a grammar for structure-aware mutation [127], 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,

Algorithm 3: 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: $\mathcal{C}_{\text{✗}}$: crashing seeds

$P_f \leftarrow$ Instrument (P_0)
Grammar $G \leftarrow$ ChatGrammar (p)
$C \leftarrow C \cup$ EnrichCorpus (C, p)
PlateauLen $\leftarrow 0$
StateMachine $S \leftarrow \emptyset$
repeat
  State $s \leftarrow$ ChooseState (S)
  Messages M, response $R \leftarrow$ ChooseSequence (C, s)
  $\langle M_1, M_2, M_3 \rangle \leftarrow M$ *(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).*
  for i from 1 to AssignEnergy (M) **do**
    if *PlateauLen* $<$ *MaxPlateau* **then**
      if UniformRandom $() < \epsilon$ **then**
        $M_2' \leftarrow$ GrammarMutate (M_2, G)
        $M' \leftarrow \langle M_1, M_2', M_3 \rangle$
      else
        $M' \leftarrow \langle M_1,$ RandMutate $(M_2), M_3 \rangle$
    else
      $M_2' \leftarrow$ ChatNextMessage (M_1, R)
      $M' \leftarrow \langle M_1, M_2', M_3 \rangle$
      PlateauLen $\leftarrow 0$
    $R' \leftarrow$ SendToServer (P_f, M')
    if IsCrashes (M', P_f) **then**
      $\mathcal{C}_{\text{✗}} \leftarrow \mathcal{C}_{\text{✗}} \cup \{M'\}$
      PlateauLen $\leftarrow 0$
    else if IsInteresting (M', P_f, S) **then**
      $C \leftarrow C \cup \{(M', R')\}$
      $S \leftarrow$ UpdateStateMachine (S, R')
      PlateauLen $\leftarrow 0$
    else
      PlateauLen $\leftarrow$ *PlateauLen* $+ 1$
until timeout T reached or abort-signal

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 [92]. 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.

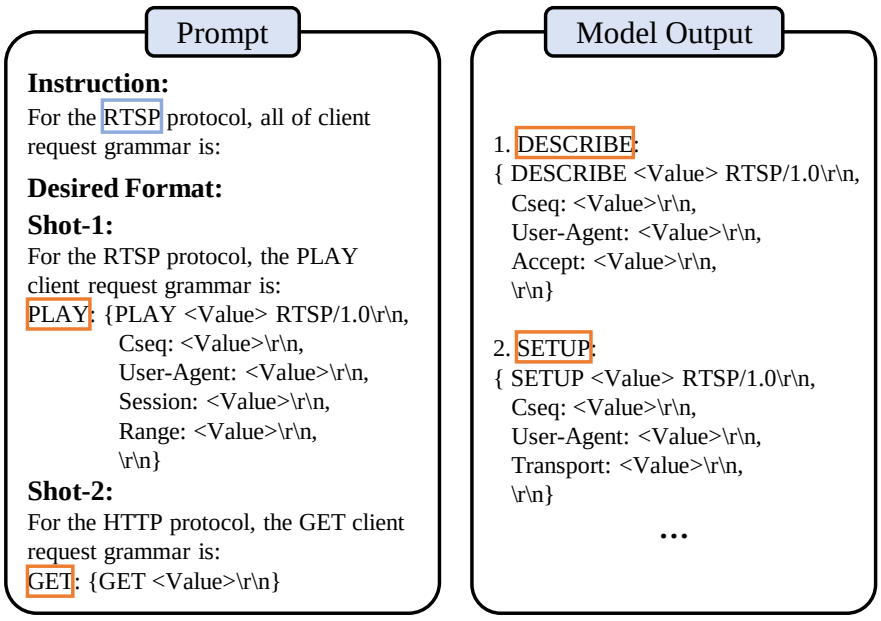


Figure 4.6: Example of the model prompt and the responding response for extracting the RTSP grammar.

To make the LLM generate a machine-readable grammar, we ultimately employ *in-context few-shot learning* [24, 145] within the domain of prompt engineering. With the increasing understanding of LLMs, many prompt engineering approaches have been proposed [24, 157, 161]. 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 4.6 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 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* [157] 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 4.6 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 3 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.

4.4.1.2 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 [74], researchers employed the LLM to generate variants of given inputs by tapping into their ability

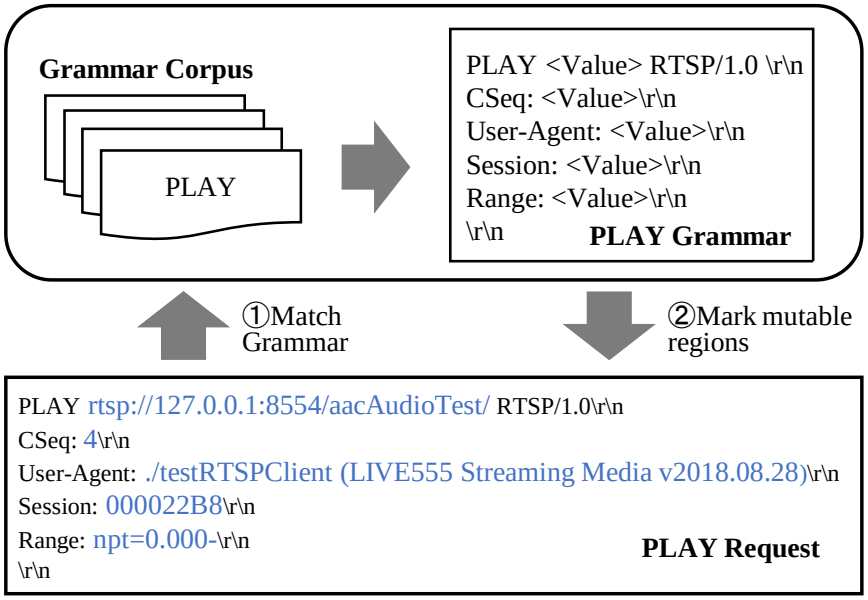


Figure 4.7: Workflow of the grammar-based mutation using the PLAY request of the RTSP protocol as the example.

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 4.4.3. Here, we proceed to introduce the workflow of mutation based on the extracted grammar.

In line 9 of Algorithm 3, 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 4.7. 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 4.7, 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 3. Nonetheless, LLMPF conducts structure-aware mutations with a higher likelihood, considering that valid messages hold a greater potential for exploring a larger state space.

4.4.2 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 4.3.2), we propose to enrich the initial seed corpus used for fuzzing (line 3 of Algorithm 3). 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 4.4.1, 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 4.6. 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* → *type*}.

While detecting the missing message types, we first utilize the grammar corpus *G* obtained in Section 4.4.1 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 4.8. 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.

4.4.3 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 4.2. 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

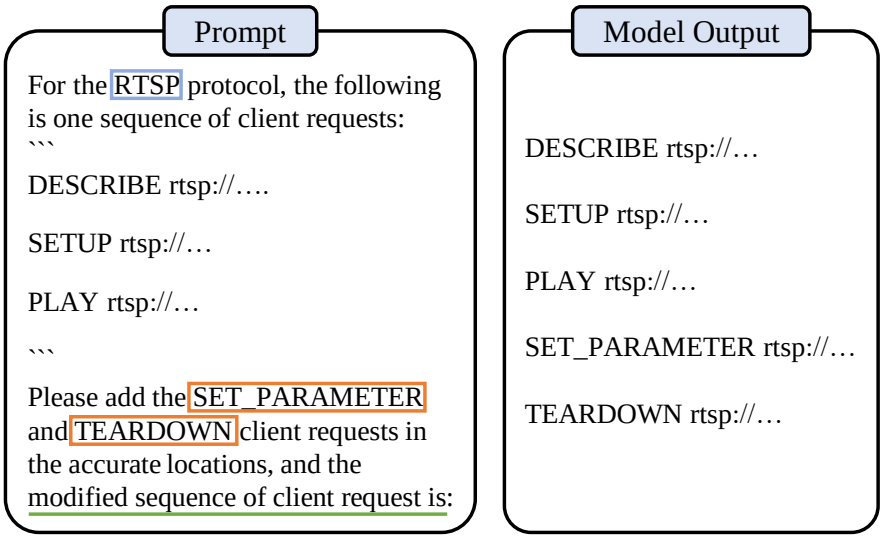


Figure 4.8: Example of the model prompt and the responding response for enriching initial seed corpus (we omit the details of messages).

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 [126, 8]. 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 work, 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 4.3.3, 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 3). 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).

Prompt Template

In the [protocol-name] protocol, the communication history between the [protocol-name] client and the [protocol-name] server is as follows:
Communication history:
```
[Put the communication history here]
```

The next client request that can induce the server's state transition to other states is:

Desired format of one real client request:
```
[Put one real message example from the initial seed corpus here]
```

Figure 4.9: The prompt template for obtaining the next client request that can induce the server's state transition to other states.

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 4.9. 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 $M_1 = \{\text{SETUP}\}$, it responds with $R_1 = \{\text{200-OK}\}$, 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 $H = \{\text{SETUP}, \text{200-OK}\}$. In response, the LLM is highly likely to reply a PLAY or RECORD message, as indicated by the study results in Section 4.3.3. These messages lead the server to transition to a different state, overcoming the coverage plateau.

4.4.4 Implementation

We implemented this LLM-guided protocol fuzzing (*cf.* Algorithm 3) into AFLNet [126], 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.

4.5 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?

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

To answer these questions, we follow the recommended experimental design for fuzzing experiments [89, 23].

4.5.1 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 4.4.3). Once the coverage plateau is reached, ChatAFL prompts the LLM to generate message sequences that surpass the coverage plateau (Section 4.4.3). 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 [132, 157], a temperature of 0.5 was used to extract the grammar and enrich the initial seeds (*cf.* Section 4.4.1 & Section 4.4.2). To generate new messages, J. Qiang *et al.* [74] 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 4.4.3). When extracting the grammar, for the *self-consistency check* [157], we use five repetitions. As confirmed in our case study (*cf.* Section 4.3.1), we found five repetitions sufficient to filter out incorrect cases.

4.5.2 Benchmark and Baselines

Table 4.2 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 [126, 114, 130, 141]. The protocols cover a wide range of applications, including streaming, messaging, and file transfer. The implementations are mature and widely used both in enterprises

Table 4.2: 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

Table 4.3: Average number of state transitions for our ChatAFL and the baselines AFLNet and NSFuzz in 10 runs of 24 hours.

Subject	ChatAFL	Comparison with AFLNet				Comparison with NSFuzz			
		AFLNet	Improv	SpedUp	$\hat{A}_{12}$	NSFuzz	Improv	SpedUp	$\hat{A}_{12}$
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 4.4: 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%	—

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 [126] is a popular open-source, state-of-the-art, mutation-based, code- and state-guided protocol fuzzer. NSFuzz-v [130] 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 [8] which was published around the same time but implemented into LibFuzzer [98]. SGFuzz also uses the sequence of state variable values to implicitly capture the coverage of the protocol state space. Other protocol fuzzers, like StateAFL [114] and BooFuzz [81] have previously been (unfavourably) compared to AFLNet or NSFuzz-v, i.e., the tools that we use as baselines.

4.5.3 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 [89, 23]. 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 [115]. 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.

4.5.4 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.

4.6 Experimental Results

4.6.1 State Space Coverage (RQ.1)

Transitions. Table 4.3 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 [6]).

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 4.4 shows the average number of states covered by our tool ChatAFL versus the two baselines AFLNet and NSFuzz-v and the corresponding

Table 4.5: Average number of branches covered by our ChatAFL and the baselines AFLNet and NSFuzz in 10 runs of 24 hours.

Subject	**ChatAFL**	**Comparison with AFLNet**				**Comparison with NSFuzz**			
		AFLNet	**Improv**	**SpedUp**	$\hat{A}_{12}$	**NSFuzz**	**Improv**	**SpedUp**	$\hat{A}_{12}$
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 4.6: Improvements in terms of branch coverage compared with baseline if we enable each strategy one by one.

Subject	**CL0**	**CL1**			**CL2**			**CL3**		
		Improv	**SpedU**	$\hat{A}_{12}$	**Improv**	**SpedU**	$\hat{A}_{12}$	**Improv**	**SpedU**	$\hat{A}_{12}$
Live555	2,860.20	0.28%	1.60×	0.89	1.49%	8.45×	1.00	2.38%	9.61×	1.00
ProFTPD	4,763.00	3.63%	2.45×	0.60	5.27%	3.69×	0.63	7.99%	4.04×	1.00
PureFTPD	1,056.30	6.67%	1.34×	0.61	6.70%	1.36×	0.86	7.39%	1.60×	0.91
Kamailio	9,404.10	0.60%	1.75×	0.96	2.24%	8.92×	1.00	7.02%	12.69×	1.00
Exim	3,647.60	2.36%	2.48×	0.52	2.54%	2.36×	0.58	3.89%	4.27×	1.00
forked-daapd	2,227.10	4.67%	2.48×	0.68	4.93%	2.98×	1.00	6.18%	4.63×	1.00
AVG	—	3.04%	2.02×	—	3.86%	4.63×	—	5.81%	6.14×	—

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 number of reachable states, the last column of Table 4.4 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.

4.6.2 Code Coverage (RQ.2)

Table 4.5 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.

4.6.3 Ablation Studies (RQ.3)

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 4.6 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 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

Table 4.7: Statistics of nine zero-day vulnerabilities discovered by ChatAFL in widely-used and extensively-tested protocol subjects.

ID	Subject	Version	Bug Description
1	Live555	2023.05.10	Heap use after free in handling PLAY client requests
2	Live555	2023.05.10	Heap use after free in handling SETUP client requests
3	Live555	2023.05.10	Use after return in handling DESCRIBE client requests
4	Live555	2023.05.10	Use after return in handling SETUP client requests
5	Live555	2023.05.10	Heap buffer overflow in handling stream
6	Live555	2023.05.10	Memory leaks after allocating memory for stream parameters
7	Live555	2023.05.10	Heap use after free in calling sendDataOverTCP
8	ProFTPD	61e621e	Heap buffer overflow while parsing FTP commands
9	Kamailio	a220901	Memory leaks after allocating memory in parsing config files

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.

4.6.4 Discovering New Bugs (RQ.4)

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 4.7.

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 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 4.4.2).

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 4.2: INIT, READY, and PLAY. Additionally, several state transitions need to be covered: INIT → READY, READY → PLAY, PLAY → READY, and then READY → 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 → READY transition (*cf.* Section 4.4.3).

Lastly, we should not ignore the contribution of structure-aware mutation. To trigger this bug, a minimal message sequence is required: SETUP → PLAY → PAUSE → 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.

4.6.5 Experience on Manual Effort

During the ChatAFL's usage, no manual effort was needed to run the experiments for all protocols shown in Table 4.2. Specifically, when extracting grammar from the LLM, we utilize the prompt shown in Figure 4.6. 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 4.4.1).

To enrich initial seeds, we utilize the prompt template in Figure 4.8. 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 4.6. In soliciting the LLM's assistance to overcome coverage plateaus, we generate the complete prompt using the template in Figure 4.9. 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.

4.7 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 a 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.

Chapter 5

Testing Stateful Distributed Systems

In this chapter, we present MALLORY: the first framework for greybox fuzz-testing of distributed systems. Unlike popular blackbox distributed system fuzzers, such as JEPSEN, that search for bugs by randomly injecting network partitions and node faults or by following human-defined schedules, MALLORY is adaptive. It exercises a novel metric to learn how to maximize the number of observed system behaviors by choosing different sequences of faults, thus increasing the likelihood of finding new bugs. Our approach relies on timeline-driven testing. MALLORY dynamically constructs Lamport timelines of the system behaviour and further abstracts these timelines into happens-before summaries, which serve as a feedback function guiding the fuzz campaign. Subsequently, MALLORY reactively learns a policy using Q-learning, enabling it to introduce faults guided by its real-time observation of the summaries.

We have evaluated MALLORY on a diverse set of widely-used industrial distributed systems. Compared to the start-of-the-art blackbox fuzzer JEPSEN, MALLORY explores 54.27% more distinct states within 24 hours while achieving a speed-up of 2.24×. At the same time, MALLORY finds bugs 1.87× faster, thereby finding more bugs within the given time budget. MALLORY discovered 22 zero-day bugs (of which 18 were confirmed by developers), including 10 new vulnerabilities, in rigorously-tested distributed systems such as Braft, Dqlite, and Redis. 6 new CVEs have been assigned.

5.1 Introduction

A common approach to finding bugs in distributed systems in practice is *stress-testing*, in which the system is subjected to faults (e.g., network partitions, node crashes) and its behaviour is checked against a property-based specification. This approach is implemented by tools like Jepsen [88], a testing framework that is well-known for its effectiveness in finding consistency violations in distributed databases [101]. An alternative to stress-testing is *systematic testing*, commonly known as *software model checking*. In this approach, the system under test is placed in a deterministic event simulator and its possible schedules are systematically explored [93, 100, 49, 169, 45]. The simulator exercises different interleavings of system events by reordering messages and injecting node and network failures. Systematic testing is well-suited to finding "deep" bugs, which require complex event interleavings to manifest, but is relatively heavyweight, as it requires integration with the system under test either in the form of a manually-written pervasive test harness or a system-level interposition layer. While not as effective at finding deep bugs, stress-testing is widely used due to its low cost of adoption and good effort-payout ratio.

Problem statement. We make the following observation: in terms of the ease-of-use/effectiveness trade-off, blackbox fuzzing of sequential programs is similar to stress-testing of distributed systems, while whitebox fuzzing corresponds to software model checking. However, unlike in the sequential case, there is no greybox fuzzing approach for distributed systems. Our goal is to explore this opportunity by extending Jepsen with the ability to *perform observations* at runtime about the behaviour of the system and to *adapt* its testing strategy based on feedback derived from those observations. In doing so, we are not aiming to match the thoroughness of systematic testing, but to provide a more effective and principled way to conduct stress-testing while maintaining its ease of use.

Challenges. In the last decade, developing a greybox fuzzer for sequential programs has become more streamlined, due to fuzzers like AFL [167]. In short, AFL works by generating and mutating inputs to a program being tested, aiming to trigger crashes or other unexpected behavior. It uses a feedback-driven approach, keeping track of inputs that cause the program to take *new code paths* and prioritizing mutations

that are likely to explore these paths further. Attempting to adapt the greybox approach to Jepsen-style distributed system testing leads us to three questions:

Q1 What is the space of inputs to a distributed system that could be explored adaptively?

Q2 What observations are relevant for a distributed system and how should they be represented?

Q3 How can one obtain feedback from the observations?

Question **Q1** is already answered by Jepsen: the role of "inputs" for distributed systems is played by *schedules*, that can be manipulated by injecting faults. Even though Jepsen can control the fault injection, in the absence of a good feedback function, it (a) requires human-written generators to explore the domain of schedules if something more than random fault injection is required [4] and (b) repeatedly explores equivalent schedules.

To answer **Q2**, we recall perhaps the most popular graphical formalism to represent interactions between nodes in distributed systems: so-called *Lamport diagrams* (a.k.a. *timelines*), i.e., graphs showing relative positions of system events, as well as causality relations between them [90, 104]. Such diagrams have been used in the past to visualize executions in distributed systems [17]. We use them as distributed analogues of "new code paths" from sequential greybox fuzzing. In other words, being able to observe and record new shapes of Lamport diagrams is an insight that brings AFL-style fuzzing to a distributed world.

To make our approach practical, we also need to address **Q3**. The problem with using observed Lamport diagrams to construct a feedback function is that in practice, no two different runs of a distributed system will produce the *same* timeline. That is, such new observations will *always* produce new feedback, even though in practice many runs are going to be equivalent for the sake of testing purposes—something we need to take into account.[1] As a solution, we present a methodology for extracting feedback from dynamically observed timelines by *abstracting* them into concise

[1] A sequential analogue of concrete distributed timelines would be a trace of *all* memory operations—too precise to recognise equivalent executions.

happens-before summaries, which provide the desired trade-off between the feedback function's precision and effectiveness.

Contributions. The solutions to **Q1**–**Q3** provide a versatile conceptual framework for greybox fuzzing of distributed systems. Building on these insights, we present our main practical innovation: MALLORY, the first greybox fuzzer for distributed systems. Unlike the blackbox testing approach of JEPSEN that requires human-written schedule generators, MALLORY reactively *learns* them by (a) observing the behaviour of the system under test as it executes and (b) rewarding actions that uncover new behaviour. Below, we detail the design, implementation, and evaluation of MALLORY.

- *Timeline-driven testing*: dynamically constructing Lamport diagrams (timelines) of the system under test as it runs, and further abstracts the timelines into happens-before summaries. It is used to define a *feedback function* guiding greybox fuzzing.
- *Reactive fuzzing*: a reactive method for making optimal decisions to achieve maximised behaviour diversity. It reactively learns a policy using Q-learning to decide what actions to take in observed states, to incrementally construct a schedule.
- *End-to-end implementation* of MALLORY: a fuzzing framework for distributed systems. MALLORY extends the widely used JEPSEN framework—MALLORY can be seen as an adaptive generator of schedules for JEPSEN tests. The tool is publicly available at

  https://github.com/dsfuzz/mallory

- *Comprehensive evaluation* of MALLORY on several widely-used industrial distributed system implementations. In our experiments, MALLORY covers 54.27% more distinct states within 24 hours and achieves the same state coverage about 2.24× faster than JEPSEN. In terms of reproducing existing bugs, MALLORY speeds up the bug finding by 1.87× and finds 5 more bugs compared to JEPSEN. Moreover, in rigorously-tested distributed systems,

MALLORY found 22 previously unknown bugs, including 10 new security vulnerabilities and 6 newly assigned CVEs. Out of these 22 bugs, 18 bugs have been confirmed by their respective developers. In our experiments, JEPSEN could only detect 4 of these bugs.

5.2 Overview

In this section, we illustrate the workflow of our technique for adaptively detecting anomalies in distributed systems.

5.2.1 Bugs in Distributed Systems

As a motivating example, let us consider a known bug in the implementation of the Raft consensus protocol [118] used by Dqlite, a widely-used distributed version of SQLite developed by Canonical.[2]

The purpose of using a *consensus protocol* in a distributed system is to ensure the system maintains a consistent and reliable state even in the presence of faults. In Raft, one of the most widely used consensus protocols, a single leader accepts client requests and replicates them to all nodes that persist them as log *entries*. Conflicting entries in a Raft cluster can appear when different nodes receive different log entries during a network partition. Over time, the number of replicated entries might grow very large, which, in turn, might cause issues if certain nodes need to be brought up-to-date after having experienced a temporary downtime. To address this issue, Raft periodically takes a *snapshot* of the current system state, discarding old log entries whose outcome is reflected in the snapshot. Additionally, nodes can be removed from the cluster or join it, thus changing the configuration of the system as it runs. When a configuration change is initiated, the current leader replicates a *configuration change entry* to all the nodes in the cluster. A new configuration becomes permanent once it has been agreed upon and committed by a majority of the nodes, yet a server starts using it as soon as the configuration entry is added to its log, even before it is committed [118, §6]—a fact that is important for our example. If there is a failure during the process of agreeing on a new configuration,

[2]Available at `https://dqlite.io`; 3.4k stars on GitHub at the time of writing.

```
145    int membershipRollback(struct raft *r){
146      ...
158      // Fetch last committed configuration entry
159      entry = logGet(&r->log, r->config_index);
160      assert(entry != NULL);
176    }

986    static int deleteConflictingEntries(){
987      ...
1007     // Discard uncommitted config changes
1008     if (uncommitted_config_index>=entry_index){
1009       rv = membershipRollback(r);
1010     }
1042   }
```

Code 5.1: Simplified Dqlite code for membership rollback.

such as a network partition, the new configuration may not be fully replicated to the majority, in which case the leader node will attempt to perform a *membership rollback* by adopting the last committed configuration entry from its log.

The bug in question occurs during a membership rollback happening *immediately after* performing a snapshot operation, leading to a failure to restore the last committed configuration [75]. Code 5.1 shows the affected fragment of the actual implementation in Dqlite, which deals with removing conflicting entries during node recovery. In case there is an uncommitted configuration entry among the conflicting entries to remove, a Dqlite server has to first roll back to the previously committed membership configuration via `membershipRollback` (line 1009). When this happens after a snapshot operation, which has removed the last committed configuration entry, the assertion on line 160 gets violated.

To show how this rather subtle bug can be triggered in a real-world environment, consider a run of a Dqlite cluster depicted in Figure 5.1. The initial cluster comprises five servers S_1–S_5, with S_1 assumed to be a leader. Server S_4 requests (to the leader S_1) to be removed from the cluster. Upon receiving this request, leader S_1 appends the configuration change entry into its log (①) and attempts to replicate it to all other members, but only succeeds to do so for S_2 (①), failing to reach S_3, S_4, and S_5 due to a sudden partition in the network. At the same time, the number of log entries at the server S_2 reaches a threshold value, prompting it to take a snapshot (②), while *already using* the latest configuration (which has not been agreed upon by the majority), thus, discarding the old configuration entry from the snapshot. To

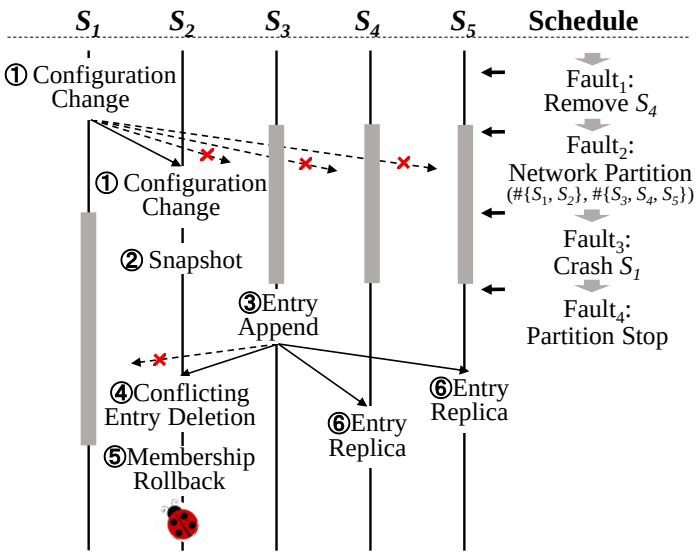


Figure 5.1: A timeline of the Dqlite membership rollback bug. Gray vertical rectangles correspond to node downtimes.

make things worse, S_1 crashes at the same time. Although the network of S_3, S_4, and S_5 recovers soon after, the cluster has already lost its leader. At some point, a new leader election is initiated (details omitted), with S_3 eventually becoming the leader and attempting to synchronise logs across the nodes (③). Prompted to do so, S_2 detects a conflicting entry in its log (i.e., the uncommitted configuration change (①)) and deletes it (④). It then attempts to retrieve the last committed configuration entry to roll back the membership (⑤), which is long gone due to the prior snapshotting (②), triggering the assertion violation at line 160 of Code 5.1.

5.2.2 Fuzzing Distributed Systems via Jepsen

As demonstrated by the example, identifying a bug in a distributed system in some cases boils down to constructing the right sequence of *faults*, such as network partitions and node removals, resulting in an execution that leads to an inconsistent state and, subsequently, to the violation of a code-level assertion or of an externally observable notion of consistency (e.g., linearisability [70]). The state-of-the-art fuzzing tool Jepsen provides a means to randomly generate sequences of faults with the goal of discovering such bugs.

Algorithm 4 provides a high-level overview of the workings of Jepsen (let us

Algorithm 4: Fuzzing with JEPSEN and MALLORY

Input: P_0: system under test (SUT)
Input: *Nem*, *Faults*: a nemesis and the faults it can enact
Input: *Oracles*: a set of test oracles for bug detection
Input: S: number of steps in each schedule
Input: T: total time budget for testing
Output: *Bugs*: a set of bugs detected

```
1:  P_f ← instrumentSystem (P_0)
2:  Policy ← {initState, Faults}
3:  repeat
4:      curState ← initState
5:      repeat
6:          fault ← Policy.getNextFault (curState)
7:          Nem.enactFault(fault)
8:          events ← observeSystemUnderTest (P_f)
9:          timeline ← constructTimeline (events)
10:         nextState ← abstractTimeline (timeline)
11:         rwd ← calculateReward (curState, fault, nextState)
12:         Policy ← learn (Policy, curState, fault, rwd)
13:         curState ← nextState
14:     until maximum steps S reached
15:     resetSystemUnderTest (P_f)
16: until time budget T exhausts
17: Bugs ← Oracles.identifyBugs (events)
```

ignore the grayed fragments for now). JEPSEN requires a lightweight harness for the system under test to define how to start and stop it, to enact faults, introduce client requests, and collect logs (line 1). For brevity, Algorithm 4 does not show the set-up of the SUT at the beginning of each test or the introduction of client requests, which the SUT is constantly subjected to by client processes. Importantly, JEPSEN allows the user to define testing *policies* (a.k.a. "generators") responsible for introducing specific types of external inputs or faults (line 2). The main fuzzing loop of JEPSEN is shown in lines 3–16 of the algorithm.

During each run of the outer loop, the framework generates a system-specific external input or fault (line 6) via a *policy*, and enacts it using a *nemesis*—a special process, not bound to any particular node, capable of introducing faults. Such inputs may, for example, be the decision to remove a node from the system, as, e.g., is done by node S_4 in our running example. As the system is executing, the

framework records its observations (line 8) for future analysis to detect the presence of bugs or specification violations (line 17). This process continues until the time budget T is exhausted (line 16). The test run is segmented into schedules of S steps each, after which the system is reset (line 15).

Getting back to our example, we can see that the membership rollback bug can be exposed by the scheduled sequence of inputs/faults that first initiates the removal of S_4 from the cluster and then creates a network partition ($\#\{S_1, S_2\}$, $\#\{S_3, S_4, S_5\}$), followed by node a crash of S_1. Randomly generating this particular sequence of faults via Jepsen, while possible, is somewhat unlikely. The reason is: before coming across this schedule, Jepsen may try many others, each making very little difference to the system's observable behaviour, e.g., by randomly crashing a number of nodes. In our experiments, Jepsen failed to detect this membership rollback bug (i.e., Dqlite-323 in Section 5.4.3) within 24 hours.

However, with just a little insight into the system, one can conjecture that enacting a partition right after a configuration change leads to novel system states more often than, e.g., performing another configuration change, thus, increasing the likelihood of witnessing a new, potentially bug-exposing, behaviour. Our goal is to retrofit Jepsen so it could derive these insights at run time and adapt the policies accordingly.

5.2.3 Learning Fault Schedules from Observations

The high-level idea behind Mallory, our fuzzing framework, is to enhance Jepsen with the ability to *learn* what kinds of faults and fault sequences are most likely going to result in previously unseen system behaviours. To achieve that, we augment the baseline logic of Algorithm 4 by incorporating the grayed components that keep track of the observations made during the system runs. The first change is to add instrumentation to the system under test (line 1) to record significant *events* (e.g., taking snapshots or performing membership rollbacks in Figure 5.1) during the execution, additional to those Jepsen already records, i.e., client requests, and responses. More interestingly, the fault injection policy is now determined not just by the kinds of faults and inputs that can be enacted, but by the latest *abstract state* of the system, whose nature will be explained in a bit and that is taken to be

some default *initState* at the start of the fuzzing campaign (line 2).

The main addition consists of lines 9–13 of the algorithm. Now, while running the system, the fuzzer collects sequences of events recorded by the instrumented nodes, as well as message-passing interactions between them; the exact nature of events and how they are collected will be described in Section 5.3.1. The information about the recorded events and their relative ordering is then used to construct a (Lamport-style) *timeline* and subsequently *summarised* to obtain the new abstract state *nextState* (lines 9–10)—the design of these two procedures, detailed in Section 5.3.2, is the central technical contribution of our work. The newly summarised abstract state is used to calculate the reward *rwd* by estimating how dissimilar it is compared to abstract states observed in the past (line 11). Finally, the reward is used to dynamically update the policy, after which the loop iteration repeats with the updated abstract state (lines 12–13).

Postponing until Section 5.3 the technicalities of computing abstract states, calculating rewards, and updating the policy, let us discuss how the introduced changes might increase the likelihood of discovering the bug-inducing system behaviour from Figure 5.1. We now pay attention to the six kinds of events (①-⑥) that can be recorded in the system, as well as their relative happens-before ordering is computed across multiple nodes. Consider a fault injection policy that introduces a sequence of node removals (such as Fault_1). After triggering several configuration changes (i.e., event ①), such a policy will not introduce many new behaviours in a long run, which will prompt our adaptive fuzzer to prefer other faults, e.g., network partitions. By iterating this process, observing new behaviours (i.e., different event sequences) in the form of novel abstract states and de-prioritising policies that have not generated new behaviours, the fuzzer will eventually discover a sequence of faults leading to the membership rollback bug.

It is important to note that the fact that a particular policy has not produced a new abstract state (i.e., a new observable behaviour) in a particular run does not necessarily mean that it needs to be discarded for good. Due to the nature of the applications under test, Mallory, similarly to Jepsen, does not provide a fully deterministic way to inject faults, hence some behaviours might depend on the absolute timing of faults. This is taken into account by Mallory's learning (*cf.* Section 5.3.3), which leaves a possibility for such a policy to be picked again in

the future, albeit, with a lower probability.

In our experiments, due to MALLORY's adaptive learning, the membership rollback bug was discovered in 8.68 hours (JEPSEN failed to discover it in 24 hours). In the following, we give a detailed description of MALLORY's design (Section 5.3) and provide thorough empirical evidence of its effectiveness and efficiency for discovering non-trivial bugs in distributed systems (Section 5.4).

5.3 Mallory Framework

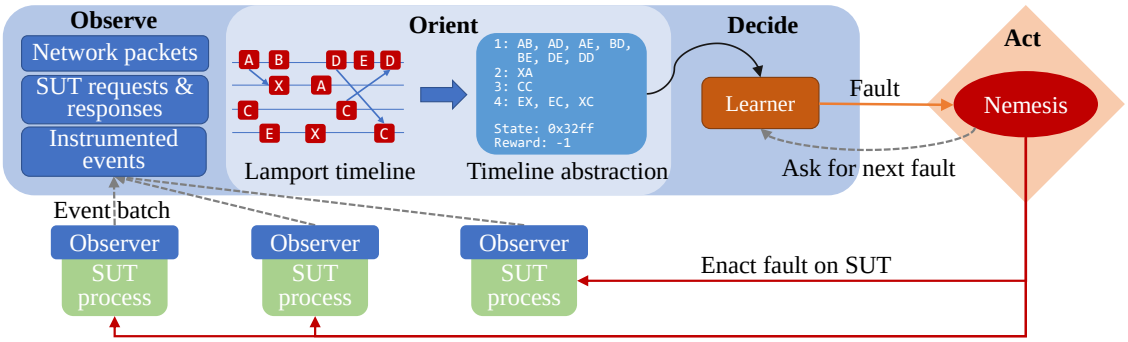


Figure 5.2: The central observe-orient-decide-act loop in MALLORY. A centralised *mediator* collects events from *observers* distributed at the nodes in the SUT, and drives the test execution. Faults decided by MALLORY are enacted by JEPSEN.

At its core, MALLORY implements an adaptive *observe-orient-decide-act* (OODA) loop:

- *Observe*—observe each node's internal behaviour and intercept all network communication between nodes;
- *Orient*—construct a global Lamport timeline of the system's behaviour to obtain a bird's eye view of the execution, and abstract the timeline into a manageable representation, called a happens-before summary, used to understand the current state of the system and to determine the effectiveness of previous actions;
- *Decide*—choose a fault to inject based on the current observed summary and the past execution history;

- *Act*—inject the fault and repeat the loop.

Unlike whitebox fuzzers, which rely on encapsulating the system under test in an event simulator [49, 122], Mallory operates on the actual system in its normal distributed environment—a firm requirement to minimise the friction (i.e., adoption effort). In particular, Mallory does not have the luxury of being able to "pause" the system and observe its state before deciding what actions to take, as it operates in real-time, in a reactive manner. This means Mallory itself is a distributed system, which complicates its implementation slightly. Nonetheless, its architecture is designed to hide this as much as possible from users, as will become apparent.

To explain Mallory's design, we will walk through an entire observe-orient-decide-act loop, step by step, gradually introducing its architectural components.

5.3.1 Observing System Under Test

Mallory's first task is to observe the system under test (SUT). Broadly, there are three types of observations that we can make: (1) network observations, which capture communication between nodes in the SUT (e.g., a packet was sent from node A to node B and received by node B), (2) external observations, which capture the input-output behaviour of the system (e.g., requests and responses for a database), and (3) internal observations, which capture a node's internal behaviour (e.g., a function was executed, a conditional branch was taken, an error message was logged). In the following, we use "observation" and "event" interchangeably.

Events happen on a particular node at a particular time. However, as is well known, in a distributed system there is no globally shared notion of time. We postpone the explanation of how Mallory constructs a global timeline without assuming precise clock synchronisation and without tagging messages with vector clocks. For now, it suffices to say that each event carries a node identifier and a *monotonic timestamp* returned by the node's system clock.

Below, we outline how Mallory observes the defined above types of events capturing the patterns of communication (Section 5.3.1.1), externally observable input/output (Section 5.3.1.2), and internal behaviour of the nodes (Section 5.3.1.3).

5.3.1.1 Packet Interception

To keep our framework lightweight and require as little modification of the system under test as possible, we capture TCP and UDP packets at the IP network layer using Linux's firewall infrastructure, rather than require users to instrument the application layer to identify protocol-level messages.

By necessity, MALLORY's architecture is distributed, matching the structure of the SUT. As shown in Figure 5.2, MALLORY consists of a number of *observer* processes, one at each node, that observe local events (bottom half of the figure), and a central *mediator* process that collates information from all observers and coordinates the execution of the test (large blue rectangle in the top half). At every node, the observer, which the JEPSEN test harness starts before the system under test, installs a NETFILTER firewall queue that intercepts all IP packets sent to or from the node. During the test, the kernel copies packets to the observer process in user space, where each packet is assigned a monotonic timestamp, recorded, and then emitted unchanged.

Mediator interception. Observers collect packet events in batches and forward those to the mediator periodically, by default every 100ms. Rather than include the entire packet in the batch, which would entail trebling network traffic, observers only record and send to the mediator a 64-bit packet identifier obtained from the source and destination IP addresses and ports and from the IP and UDP or TCP headers' identifiers, respectively. Yet we do want the mediator to have access to the packet contents: for instance, the content of messages might determine what is the best fault to introduce. To achieve this, we set up the test environment that the SUT executes in such that *all packets pass through the node running the mediator*. Concretely, we place each node on its own separate (virtual) Ethernet LAN, with the mediator acting as the gateway for all the LANs. The MALLORY mediator acts as a *man-in-the-middle* for all packets in the SUT. It can then reconstruct the identifier for each packet, and cross-reference it with the batches received from the sending and receiving nodes' observers to determine the respective timestamps. The mediator, unlike the observer, which is passive, is active and implements a full user-space firewall using NETFILTER. It can delay and drop packets when instructed to do so by the decide step of the OODA loop.

Using this infrastructure, the mediator builds up a complete picture of the system under test's communication.

5.3.1.2 Requests and Responses

The observations about input/output of nodes are made at the application layer.

We built MALLORY on top of JEPSEN, and reuse JEPSEN's infrastructure to define the test harness that: (a) sets up and starts the SUT, (b) defines and executes a workload (a sequence of client requests to the SUT), (c) enacts faults (e.g., crashing a node), and (d) checks the validity of the SUT's response to the workload. JEPSEN already captures requests and responses for validity checking (e.g., for linearisability), and we hook into this existing code, attach monotonic timestamps to events, and relay them to the mediator reactively.

5.3.1.3 Code Instrumentation

The final kind of observation we make is at the code level. We want MALLORY to be able to peek into the internal workings of the SUT, beyond what is observable to clients of the system or to eavesdroppers on the network. For this, we reuse the compile-time instrumentation infrastructure used by greybox fuzzers (e.g., AFL) for sequential programs. Like those fuzzers, MALLORY adds *instrumentation code* to the SUT to capture and expose runtime information about the program's execution. The key question is: what about the execution should we capture?

In our early experiments, we used the notion of *edge coverage*, the type of instrumentation that has become standard for fuzzing sequential programs due to its empirically-observed effectiveness. It maintains a global bitmap of code edges, and increments an approximate counter for each edge that is traversed during program execution. At the end of the execution, the bitmap serves as a summary of "what the program did," and is used by the fuzzer to assign energy and mutate its input during subsequent runs. This is a great metric for certain kinds of programs, e.g., command-line utilities and file-parsing libraries, but—as we quickly discovered—not particularly meaningful for distributed systems. The goal in fuzzing sequential programs is to generate inputs that go "deep" into a program and explore all "cases" (i.e., conditional branches). For such programs, the thoroughness of exploration is naturally defined in terms of code coverage. But this is not the case at all for

distributed systems. Distributed systems tend to be implemented as reactive event loops and run almost the same code for every request, with minor variations. Code coverage metrics tend to saturate very quickly when testing such systems.

A natural behavioural metric for distributed systems, which we came to adopt, is that of the *event trace*. Executions in a distributed system are distinguished not so much by which events happen, but by the order in which they happen. Moreover, as has been empirically observed, what tends to uncover bugs are specific subsequences of events, e.g., A before B before C, with potentially many events between them [165]. The disadvantage of event traces compared to code coverage is that the former can become very large and expensive to store and operate on, especially if every basic block is instrumented. To alleviate this issue, for now we require from the user a small amount of manual annotation of the SUT's code, in the form of `//INSTRUMENT_FUNC` and `//INSTRUMENT_BLOCK` comments, to indicate which basic blocks and functions are "interesting" and should be tracked by Mallory.

Our instrumentation creates a POSIX shared memory object accessible by the observer process, and stores in it a fixed-size global array of events along with an atomic index. We implement a LLVM pass that assigns a unique ID to every annotated basic block and function in the SUT, and inserts the hooking code at the start of the block or function. During program execution, this code gets a monotonic timestamp and records the event in the global array at a fresh position. The observer process periodically reads the shared memory object, copies the trace, and resets the counter; it also includes the trace in the periodic batch it sends to the mediator.

5.3.2 Making Sense of Observations

For the second phase of the loop, Mallory needs to make sense of the events it received from observers. The goal of this phase is to transform the "raw" event data into a form more amenable to analysis and decision-making. For this purpose, Mallory leverages timeline-driven testing. At the core of Mallory's OODA loop lie its *view* of the world, a dynamically constructed Lamport timeline of events in the SUT, and its *model* of the world, a user-defined abstraction of the timeline. Mallory first builds a birds-eye view of the SUT's execution by constructing a global timeline, then makes sense of the timeline by abstracting it into a summary

consisting of its "essential" parts, which is used to judge the effectiveness of previous actions and to decide which faults to introduce next.

As the system is executing, the MALLORY mediator receives batches of events from all the observers and adds them to a single global timeline. Every event is associated with a particular node in the SUT and has an attached monotonic timestamp from that node's system clock. In the mediator, MALLORY constructs the timelines using these events. After having a Lamport timeline describing the system's behaviour, obtained in almost real-time, we use it to drive our fuzzing campaign. More concretely, we want our decisions to adapt the system's behaviour and drive the execution towards new behaviours. But what counts as novel behaviours? A naive approach is to use the timeline itself as feedback for our decision: we want to see timelines different from what we have seen before. This does not work because the timeline is a low-level representation of the system's behaviour; all observed timelines are unique, even discounting event timestamps and packet contents. Clearly, to be able to operate effectively with timelines, we need in some fashion to *abstract* them into "what really matters". Eliminating timestamps and packets is a form of abstraction, but it is not enough: we need something a bit more clever. In default, MALLORY summarizes the constructed timelines into happens-before pairs. The result is a summary of the timeline's events that captures some of the history's essential aspects, and which we use in the next stage to decide which faults to introduce.

5.3.3 Making Decisions with Q-Learning

Equipped with a way to understand the behaviour of the SUT, in the form of timeline abstractions, MALLORY must decide which actions to take in response to what it observes.

For fuzzing sequential programs, *mutation-based power scheduling* has become the standard approach to generate novel inputs for the program under test based on observations: test inputs that exercise new behaviours are stored and mutated many times to obtain new inputs. However, this technique is ill-suited for testing distributed or reactive systems. The issue is that in the mutation-based paradigm, behavioural feedback is given for the whole input to the system under test (SUT)

after execution ends. This is reasonable for sequential programs, but not for reactive programs—the (temporal and causal) connection between fault introduced and behaviour induced is lost. Indeed, we want our fuzzer itself to be reactive and give behavioural feedback after every action taken rather than only at the end of a long schedule. This complements JEPSEN's *generative fuzzing* approach by giving behavioural feedback after every generated fault and makes MALLORY adapt in real-time to the SUT.

The concept of *timeline-driven testing* is key to how MALLORY adapts to the SUT. To decide which faults to introduce into the SUT, MALLORY employs Q-learning [159, 160], a model-free reinforcement learning approach. Q-learning enables an agent to dynamically learn a *state-action policy*, i.e., a pairing between the observed state of the environment and the optimal action to take. Based on the observed states, this policy guides the agent in selecting actions to take. After performing an action, the agent receives an immediate reward, which further refines the policy. By associating observed states with optimal actions, the agent maximises the expected rewards over its lifetime.

Q-learning can be seamlessly integrated into our distributed system fuzzer. With this approach, our fuzzer (i.e., the agent) learns a policy throughout the fuzzing campaign, which serves as a guide to explore diverse states. Specifically, during the fuzz campaign, our fuzzer observes the state $s_i \in \mathcal{S}$ of the SUT, and then employs the learned policy to select a fault $a_i \in \mathcal{A}$ (i.e., an action) to introduce, causing it to transition to the next state s_{i+1}. Simultaneously, the fuzzer receives an immediate reward $r_i \leftarrow \mathcal{R}(s_i, a_i, s_{i+1})$. Using the received reward, our fuzzer further refines the policy $\pi : \mathcal{S} \mapsto \mathcal{A}$. Subsequently, our fuzzer selects the next fault $\pi(s)$ to introduce. Over time, our fuzzer progressively learns an optimal policy that maximises the number of distinct states observed (i.e., rewards). In the following, we elaborate this process further.

Capturing States. In the context of Q-learning, states should represent the behaviour of the environment. As elaborated in Section 5.3.2, to describe the behaviour of the SUT, we adopt the timeline abstractions (e.g., a set of happens-before pairs). To incorporate Q-learning into our framework, we take a hash of the timeline abstractions to serve as *abstract states*. However, while computing states, we encounter a challenge due to the sensitivity and precision of our timeline abstractions.

In order to improve the learning speed of the state-action policy, we aim to treat states that differ only insignificantly as identical. Using a standard hash function proves ineffective, as it tends to be overly sensitive to minor variations in the timeline abstraction. Such variations can arise even when the system is operating without any injected faults, owing to the inherent non-deterministic nature of distributed systems.

To address this issue, we adopt MinHash, a locality-sensitive hash function that maps similar input values to similar hash values. In our specific case, to decide whether an observed timeline abstraction corresponds to a new distinct state, we hash it into a signature and then compare this signature to those of previously encountered states. We classify a state as distinct if the similarity falls below a threshold ε. To choose ε, we conduct a calibration stage before fuzzing by running the SUT without any faults and under constant load and observing the timeline abstractions thus obtained. The convention in [114, 102], which we also adopt, is to choose an ε value that makes 90% of such "steady" states coincide.

Learning the Policy. In our approach, the policy π is represented as a Q-table, where each column corresponds to a specific type of fault, and each row represents a distinct state. The available fault types are provided by the fuzzer and enabled at the start of the fuzzing campaign. As new distinct states are observed and added, the rows dynamically increase to accommodate them. Within the Q-table, each cell stores the Q-value for a state-action pair. When a new distinct state is added to the table, the Q-values for each state-action pair associated with that state are initialised to 0. With this setup, the process of refining the policy becomes simplified, involving adjustments to the Q-values.

Q-values are updated in response to rewards. After executing a fault, the fuzzer receives an immediate reward determined by the reward function. The reward function is devised based on our goal. Since the goal is to maximise the number of distinct states observed, we set our reward function to give a constant negative reward (-1) to states that have been observed previously. This approach incentivises Mallory to steer the SUT towards unobserved behaviors, promoting exploration.

We design the reward function as follows:

$$\mathcal{R}(s_i, a_i, s_{i+1}) = \begin{cases} -1, & \text{if } s_{i+1} \in \mathcal{S}_{observed} \\ 0, & \text{if } s_{i+1} \notin \mathcal{S}_{observed} \end{cases} \tag{5.1}$$

Once receiving the rewards, our fuzzer dynamically adjusts Q-values using the Q-learning function $\mathcal{Q}$: $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, which determines the Q-value for a specific state-action pair. When an action a_i is executed, it leads to a new state s_{i+1} from the previous state s_i (i.e., $s_i \xrightarrow{a} s_{i+1}$). Subsequently, we update the Q-value as follows:

$$\mathcal{Q}(s_i, a_i) \leftarrow (1-\alpha)\mathcal{Q}(s_i, a_i) + \alpha \left(\mathcal{R} + \gamma \underset{a'}{max}\, \mathcal{Q}(s_{i+1}, a')\right) \tag{5.2}$$

where $\alpha \in (0, 1]$ indicates the learning rate and $\gamma \in (0, 1]$ is a discount factor. The default values chosen for these parameters are $\alpha = 0.1$ and $\gamma = 0.6$, which we determined to work well empirically. With the Q-function, the new Q-value is computed and subsequently updated into the Q-table. As the fuzz campaign proceeds, our fuzzer gradually fine-tunes the Q-values, thus refining the policy to make better decisions.

Getting Next Fault. When the SUT is in a state s, our fuzzer selects the next action a based on the learned policy. In the current state s, we first obtain the Q-values for all n actions and then utilise the softmax function to convert these Q-values into a probability distribution $\mathcal{D}$. The probability $\mathcal{D}(i)$ for each action a_i $(i \in n)$ is calculated as follows:

$$\mathcal{D}(i) \leftarrow \frac{e^{\mathcal{Q}(s,a_i)}}{\sum_{j=1}^{n} e^{\mathcal{Q}(s,a_j)}} \tag{5.3}$$

To decide the next action to take, we sample from the probability distribution $\mathcal{D}$. To achieve this, we generate a random number $p \in [0, 1]$, and then check the cumulative probability of $\mathcal{D}(i)$ until we identify the first action where the cumulative probability exceeds p. This action is chosen as the next action to execute. Actions with higher probabilities have a greater likelihood of being chosen, while actions with lower probabilities still have a chance of being selected. This approach ensures a balance between favoring actions with higher probabilities while maintaining the possibility of choosing actions with lower probabilities.

Thus far, we have introduced each individual component of Algorithm 4. To kickstart the fuzzing campaign, we set the default maximum steps as 12, each with

a 2.5-second window. Following each schedule, there is a 5-second period allocated to reset the system to a stable state.

5.4 Evaluation

We implement Mallory on top of Jepsen-0.2.7, to test distributed system implementations written in C, C++, and Rust. To enable code-level instrumentation, we created a LLVM compiler pass (similar to that used by AFL [167]) to add into the compiled binary our event instrumentation, as described in Section 5.3.1.3. The code implementing this pass measures roughly 1,000 lines of C/C++ code. The observers at each node that collect events and the mediator which collates events from all observers, intercepts packets, constructs and abstracts timelines, and learns the policy required to guide fault injection, are implemented in Rust. The code for these components measures roughly 9,000 lines of Rust code. To enact faults, we implemented a linker in Jepsen that asks Mallory for the next fault to execute. This linker consists of 140 lines of Clojure code.

5.4.1 Evaluation Setup

To evaluate the effectiveness and efficiency of Mallory in exploring distinct program behaviours and finding bugs in industrial distributed system implementations, we have designed experiments to address the following questions:

RQ.1 Coverage achieved by Mallory. Can Mallory cover more distinct system states than Jepsen?

RQ.2 Efficiency of bug finding. Can Mallory find bugs more efficiently than Jepsen?

RQ.3 Discovering new bugs. Can Mallory discover new bugs in rigorously-tested distributed system implementations?

5.4.1.1 Baseline Tool

We selected Jepsen as our baseline tool due to its popularity in stress-testing distributed system implementations. To our knowledge, Jepsen is the only widely-used

Table 5.1: Detailed information about our subject programs.

Subject	Description	Protocol	Lang.	#LOC	#Stars
Braft	Baidu Raft implementation	Raft	C++	31.6k	3.5k
Dqlite	Distributed SQL DBMS	Raft	C	54.2k	3.4k
MongoDB	Distributed NoSQL DBMS	Raft	C++	1121.6k	23.6k
Redis	Distributed in-memory DBMS	Raft	C	211.4k	59.6k
ScyllaDB	Distributed NoSQL DBMS	Raft/Paxos	C++	122.4k	9.8k
TiKV	Distributed key-value DBMS	Raft	Rust	404.5k	13.0k

blackbox fuzzer in this domain. It has gained recognition for its user-friendliness and has helped to uncover numerous bugs in real-world implementations of distributed systems. By building on top of Jepsen, we have developed Mallory to enhance the effectiveness of fuzzing while preserving Jepsen's ease of use. Another blackbox fuzzer, called Namazu [113], is less popular and can only test Go/Java programs.

As described in the introduction, whitebox fuzzers such as MoDist [162] and FlyMC [100] require an extensive manually-written test harness or heavy deterministic control at the system level, and are used for systematic testing as opposed to stress-testing. Due to their heavy-weight nature, they target a different use case compared to Mallory and are less practical to adopt in industry.

5.4.1.2 Subject Programs

Table 5.1 presents the subject programs included in our evaluation. It consists of six open-source distributed system implementations written in C, C++, and Rust. We selected these subjects because: (1) they are widely used in the industry, (2) they can be instrumented by our LLVM pass, and (3) they have undergone rigorous testing using Jepsen either by contracting Jepsen's author[3] or by rolling their own Jepsen test harness. Finding new bugs in these systems would be a strong indication that Mallory performs better than Jepsen.

5.4.1.3 Event Annotations

To instrument code events, we annotated a total of 103 to 157 functions and basic blocks in our subject programs using `//INSTRUMENT_FUNC` and `//INSTRUMENT_BLOCK` forms. Specifically, we made 120 annotations for Braft, 108 annotations for Dqlite,

[3]Test reports are public at https://jepsen.io/analyses

```
145    int membershipRollback(struct raft *r){
146      ...
158      // Fetch last committed configuration entry
159      entry = logGet(&r->log, r->config_index);
160      assert(entry != NULL);
176    }

985    // INSTRUMENT_FUNC
986    static int deleteConflictingEntries(){
987      ...
1007     // Discard uncommitted config changes
1008     if (uncommitted_config_index>=entry_index){
1009       // INSTRUMENT_BLOCK
1010       rv = membershipRollback(r);
1011     }
1043   }
```

Code 5.2: Annotating Dqlite for membership rollback.

103 annotations for MongoDB, 129 annotations for Redis, 157 annotations for ScyllaDB, and 138 annotations for TiKV. This process required one of our authors to dedicate a total of 2 hours, averaging approximately 20 minutes for each subject.

We annotate "interesting" code events primarily based on the Raft and Paxos TLA+ specifications.[4] We now demonstrate the annotation process for events in the motivation example (Code 5.1), and present the annotated code in Code 5.2. In the Raft TLA+ specification, one event involves the removal of conflict entries. To annotate this event, we search for the "conflict" keyword in the Dqlite source code, resulting in 7 matched locations. However, among these matches, only one location pertains to functions or basic blocks, specifically the "deleteConflictingEntries" function. Therefore, we add the annotation `//INSTRUMENT_FUNC` before this function. Instead of annotating a function, we can also decide to annotate (some of) the basic blocks representing the calling locations of the function. However, this may require additional work in terms of determining which calling location to instrument. For example, for the "membershipRollback" function, we can annotate the basic block in line 1009 that invokes this function with the annotation `//INSTRUMENT_BLOCK` shown in Code 5.2. But choosing this calling location, takes into consideration the condition (line 1008) that there must be an uncommitted configuration entry among the conflicting entries to be removed (explained in Section 5.2.1). Once the code is annotated, our compile-time instrumentation automatically instruments the annotated code without any manual effort.

[4]The specifications are taken from https://github.com/tlaplus/Examples.

Discussion. It is worth mentioning that we adopt the heuristic based on TLA+ specifications to instrument events in our evaluation. Nevertheless, users of MALLORY have the flexibility to instrument their own custom event types in accordance with their own heuristics. For instance, this might involve the instrumentation of error handlers [165] or *enum* cases [8]. The identification of such events can be achieved fully automatically, and we have integrated this functionality into the MALLORY tool.

5.4.1.4 Configuration Parameters

To distinguish between distinct states, we set the similarity threshold ε to 0.70. To detect bugs, we adopt several test oracles: (1) AddressSanitizer (i.e., ASan) for exposing memory issues, (2) log checker to detect issues in application logs by scanning for keywords such as "*fatal*", "*error*" and "*bug*", and (3) consistency checker ELLE [87] to check consistency violations. We set up the same number of nodes as the existing JEPSEN tests: 9 nodes for MongoDB and 5 nodes for the other subjects under test. To ensure a fair comparison, we enabled the same faults in MALLORY as those used in the original JEPSEN tests (i.e., our tool does not have access to more fault types).

All experiments were conducted on Amazon Web Services using the m6a.4xlarge instance type. This instance type has 64 GB RAM and 16 vCPUs running a 64-bit Ubuntu TLS 20.04 operating system. Following community suggestions, we ran each tool for 24 hours and repeated each experiment 10 times. To reduce statistical errors, we report as results the average values obtained over the 10 runs.

5.4.2 Coverage Achieved by Mallory (RQ.1)

For the first experiment, we monitor the number of distinct states (see Section 5.3.3 for the definition of states) exercised by MALLORY and JEPSEN over time, and compare their state coverage capabilities.

To observe states exercised by JEPSEN, we ran JEPSEN in the same setup as MALLORY, but without controlling the fault injection. We collected the average number of distinct states achieved by MALLORY and JEPSEN within 24 hours across 10 runs, and we present the comparison in Figure 5.3. As shown in this figure, MALLORY outperforms JEPSEN by covering more distinct states in the same time

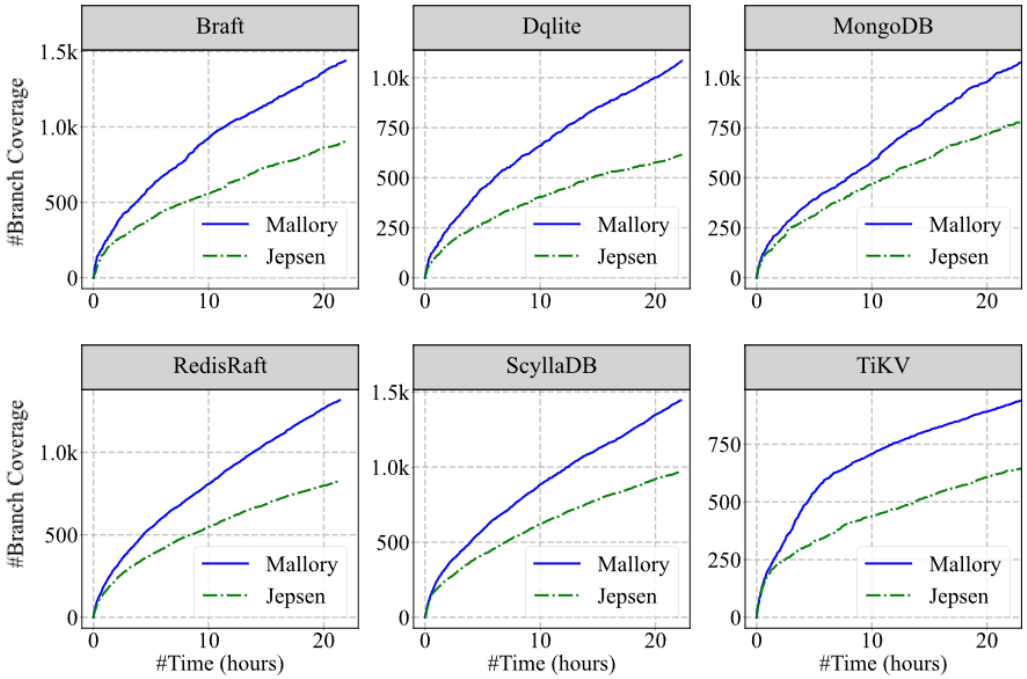


Figure 5.3: The trends in the average number of distinct states within 24 hours across 10 runs.

budget, thus exercising the SUT under more diverse scenarios.

Initially, at the start of each experiment, the number of distinct states achieved by Mallory is similar to that achieved by Jepsen, which is expected. Mallory utilises Q-learning to learn an optimal state-action policy during fuzzing, guiding it to make better decisions about actions to take in observed states. However, during the initial phase of fuzzing, this policy is not yet well-learned and lacks sufficient knowledge to avoid exploring redundant states. Consequently, both Mallory and Jepsen struggle to select the optimal actions for the observed states.

However, as fuzzing progresses, Mallory gradually refines the policy by assigning negative rewards to certain state-action pairs, thereby disincentivising certain actions selected in particular states. This disincentive effectively curtails the exploration of repetitive states and steers Mallory towards exploring more diverse states. The policy learned using Q-learning proves to be highly effective. As evident in Figure 5.3, over time, the number of distinct states covered by Mallory is significantly more than that covered by Jepsen. We do not observe the number of distinct states saturating in either tool, but Mallory's exploration rate of distinct

Table 5.2: Statistics of distinct state numbers achieved by MALLORY compared to that achieved by JEPSEN.

Subject	**State-impr**	**Speed-up**	$\hat{\boldsymbol{A}}_{\mathbf{12}}$	$\boldsymbol{U}$
Braft	59.34%	2.28×	1.00	<0.01
Dqlite	76.14%	2.56×	1.00	<0.01
MongoDB	36.48%	1.57×	1.00	<0.01
Redis	58.92%	2.06×	1.00	<0.01
ScyllaDB	48.82%	1.88×	1.00	<0.01
TiKV	45.93%	3.07×	1.00	<0.01
AVG	54.27%	2.24×	—	—

states is higher than that of JEPSEN. The gap between MALLORY and JEPSEN consistently widens, indicating that the learned policy continuously improves and becomes increasingly effective in guiding exploration.

The state coverage statistics of MALLORY over JEPSEN are listed in Table 5.2. The "State-impr" column shows the average improvement of MALLORY in the number of distinct states at the end of 24 hours, over 10 runs. Our results show that MALLORY covers an average of 54.27% more distinct states than JEPSEN on our test subjects, with an improvement ranging from 36.48% to 76.14%. The "Speed-up" column indicates the average speed-up of MALLORY compared to JEPSEN in achieving the same number of observed states. On average, MALLORY archives a 2.24× speed-up over JEPSEN. To mitigate the effect of randomness, we measured the Vargha-Delaney ($\hat{A}_{12}$) and Wilcoxon rank-sum test (U) of MALLORY against JEPSEN. $\hat{A}_{12}$ is a non-parametric measure of effect size that provides the probability that random testing of MALLORY is better than random testing of JEPSEN. U is a non-parametric statistical hypothesis test that determines whether the number of distinct states differs across MALLORY and JEPSEN. We reject the null hypothesis if $U < 0.05$, indicating that MALLORY outperforms JEPSEN with statistical significance. For all subjects, $\hat{A}_{12} = 1$ and $U < 0.01$ for MALLORY against JEPSEN. This demonstrates that MALLORY significantly outperforms JEPSEN.

Furthermore, we measured the memory consumption required to maintain the data structure of the Lamport-style timeline. The average memory consumption was 3.21 GB, which we consider acceptable. Memory consumption remains stable over time, as we only retain the portion of the timeline needed for abstraction and

remove already-abstracted portions. Additionally, our fuzzer is designed to learn and react to observations in real time (see Section 5.3.3). We measured the time taken from the point of fault injection to receiving the behaviour feedback and found that in 92.20% of cases, this process took less than one window time, i.e., MALLORY receives feedback and adapts its policy before it has to decide the next action.

In terms of state exploration, MALLORY covers 54.27% more distinct states than JEPSEN with a 2.24× speed-up.

5.4.3 Efficiency of Bug Finding (RQ.2)

To evaluate the efficiency of MALLORY at finding bugs, we compared MALLORY and JEPSEN with regards to the time required to reproduce existing bugs. To this end, we created a dataset of bugs by selecting 10 recent issues from each subject's GitHub issue list (from early 2019 to April 2023) that contained instructions for reproduction. We attempted to reproduce the bugs manually and included any successfully reproduced bugs in our dataset. We finally collected a total of 16 bugs across all subjects. The bug IDs and types of bugs are presented in the first two columns of Table 5.3.

We ran both tools, MALLORY and JEPSEN, on buggy versions of the subjects for 24 hours, repeated 10 times. The last main column shows the time used for each tool to expose the bug. We marked "T/O" if one tool failed to find the bug within the given time budget. Overall, in these 16 bugs, MALLORY successfully exposed 14 bugs, while JEPSEN only found 9 bugs. In terms of time usage, MALLORY takes much less time (i.e., 6.13 hours on average), while JEPSEN needs 11.45 hours. Hence, compared with JEPSEN, MALLORY achieves a speed-up of 1.87× in bug finding.

For shallow bugs whose states are easy to reach, such as Dqlite-338 and Dqlite-327, MALLORY and JEPSEN perform well and perform similarly. However, for deep bugs that are harder to expose, MALLORY performs much better than JEPSEN. For example, to expose Redis-51, JEPSEN took 6.40 hours, while MALLORY only took 1.66 hours. This is attributed to a faster state-exploration speed of MALLORY. In addition, since MALLORY explored more distinct states than JEPSEN, MALLORY was also able to expose more bugs. Specifically, MALLORY successfully exposed

Table 5.3: Statistics of reproduced known bugs and the performance of both MALLORY and JEPSEN in exposing these bugs.

Bug ID	**Type of bug**	**Time to exposure**		
		Mallory	**Jepsen**	$\hat{A}_{12}$
Dqlite-416	Null pointer deference	**0.76h**	1.44h	**1.00**
Dqlite-356	Snapshot installing failure	T/O	T/O	0.50
Dqlite-338	Election fatal with split votes	0.16h	0.16h	0.50
Dqlite-327	Member removal failure	0.06h	0.05h	0.49
Dqlite-324	Log truncation failure	**5.94h**	T/O	**1.00**
Dqlite-323	Membership rollback failure	**8.68h**	T/O	**1.00**
Dqlite-314	Crashing on disk failure	T/O	T/O	0.50
Redis-54	Snapshot panic	**3.33h**	5.00h	**0.95**
Redis-53	Committed entry conflicting	**0.87h**	1.17h	**0.89**
Redis-51	Not handling unknown node	**1.66h**	6.40h	**1.00**
Redis-44	Loss of committed write logs	**0.34h**	0.58h	**0.60**
Redis-43	Snapshot index mismatch	0.16h	0.16h	0.50
Redis-42	Snapshot rollback failure	0.29h	0.26h	0.50
Redis-28	Split brain after node removal	**9.56h**	T/O	**1.00**
Redis-23	Aborted read with no leader	**7.29h**	T/O	**1.00**
Redis-17	Split brain and update loss	**11.06h**	T/O	**1.00**
Bugs exposed in total		14	9	—
Average time usage		6.13h	11.45h	—
Speed-up on time usage		—	1.87×	—

Dqlite-324, Dqlite-323, Redis-28, Redis-23, and Redis-17, while JEPSEN had difficulty in exposing them. We further investigated the two bugs (i.e., Dqlite-356 and Dqlite-314) missed by MALLORY, and found exposing these bugs needs to inject specific environment faults (e.g., disk faults), which were not included in our evaluation.

To mitigate randomness, we adopt the Vargha-Delaney ($\hat{A}_{12}$) to measure the statistical significance of performance gain. The last subcolumn of Table 5.3 shows these results. We mark $\hat{A}_{12}$ values in bold if they are statistically significant (taking 0.6 as a threshold). In most cases, MALLORY significantly outperformed JEPSEN.

In terms of bug finding, MALLORY finds 5 more bugs and finds bugs 1.87× faster than JEPSEN.

Table 5.4: Statistics of the zero-day bugs discovered by MALLORY in rigorously tested systems; a total of 22 previously unknown bugs, 18 bugs confirmed by their developers, and 10 software vulnerabilities.

ID	Subject	Bug description	Checker	Bug status	Jepsen
1	Braft	Read stale data after a newly written update is visible to others	ELLE	Investigating	✓
2	Braft	Leak memory of the server when killed before its status becomes running	ASan	CVE-Granted, fixed	✗
3	Dqlite	Two leaders are elected at the same term due to split votes	Log checker	Confirmed	✗
4	Dqlite	No leader is elected in a healthy cluster with an even number of nodes	Log checker	Confirmed, fixed	✗
5	Dqlite	A node reads dirty data that is modified but not committed by another node	ELLE	Confirmed	✗
6	Dqlite	Lose write updates due to split brain	ELLE	Confirmed	✗
7	Dqlite	A null pointer is dereferenced due to missing the pending configuration	ASan	CVE-Requested	✓
8	Dqlite	Leak allocated memory when failing to extend entries	ASan	CVE-Requested, fixed	✗
9	Dqlite	Buffer overflow happens while restoring a snapshot	ASan	CVE-Requested	✗
10	Dqlite	A node has an extra online spare	Log checker	Confirmed	✗
11	Dqlite	Violate invariant as a segment cannot open while truncating inconsistent logs	Log checker	CVE-Requested	✗
12	MongoDB	Not repeatable read due to missing the local write update	ELLE	Confirmed	✗
13	MongoDB	Not read committed due to missing the newly written update	ELLE	Confirmed	✗
14	Redis	Read stale data after new data is written to the same key	ELLE	Confirmed	✗
15	Redis	Buffer overflow due to writing data to a wrong data structure	ASan	CVE-Granted, fixed	✗
16	Redis	Runtime panic on initialising a cluster due to database version mismatch	Log checker	CVE-Granted	✓
17	TiKV	No leader is elected for a long time in a healthy cluster	Log checker	Investigating	✗
18	TiKV	Lose write updates due to split brain	ELLE	Investigating	✗
19	TiKV	Runtime fatal error when one server cannot get context before the deadline	Log checker	CVE-Granted	✗
20	TiKV	Runtime fatal error in a server when the placement driver is killed	Log checker	CVE-Granted	✗
21	TiKV	Runtime fatal error when failing to update max timestamp for the region	Log checker	CVE-Granted	✗
22	TiKV	Monotonic time jumps back at runtime	Log checker	Investigating	✓

5.4.4 Capability of Exposing New Bugs (RQ.3)

To evaluate Mallory's capability of exposing new bugs, we utilised Mallory on the latest versions of our subjects. In the course of the experiment, Mallory produced promising results, as demonstrated in Table 5.4. Although all of these subjects have been rigorously tested by Jepsen and other tests, Mallory was still able to find a total of 22 previously unknown bugs, and 18 bugs of them were confirmed by their developers. Out of these 22 bugs, 10 bugs were associated with vulnerabilities, and we have requested CVE IDs for them. As of the paper submission, we have already obtained 6 CVE IDs and the remaining requests are still being processed.

We conducted a thorough analysis of the nature of these new bugs found by Mallory, shown in Table 5.4. The table also includes the bug checkers used to uncover these bugs. Among these 22 bugs, 7 bugs were determined to be consistency violations exposed by the Elle consistency checker. Three bugs (#3, #4, and #17) violated the Raft protocol due to missing leaders or the existence of two leaders in the same term, and they were detected by the log checker. AddressSantizer (i.e., ASan) exposed 5 memory issues. Furthermore, the log checker detected 7 runtime failures or invariant violations. These results indicate that Mallory is beneficial to expose diverse types of previously unknown bugs.

In addition, we applied Jepsen to detect these 22 new bugs; however, under the allotted time limit, Jepsen was only able to detect four of them (i.e., the bugs #1, #7, #16, and #22), as shown in the last column of Table 5.4. This result is expected because these subject systems routinely undergo Jepsen testing by their developers, making it challenging for Jepsen to discover new bugs.

In the following, we provide two case studies to illustrate bugs that were exposed by Mallory.

Case Study: Bug #2 in Braft. Braft is a robust Raft implementation designed for industrial applications, which is widely used within Baidu to construct highly available distributed systems. However, a critical vulnerability, known as Bug #2, remained undetected in all Braft release versions from 2019 until its recent patch. This bug occurs when a server cannot release its allocated memory before failure, resulting in a memory leak issue.

To trigger this issue, a minimum of three environmental faults must be introduced sequentially. Initially, the server dynamically allocates enough memory for its operation, which is explicitly managed by itself. However, before releasing the allocated memory, the server is paused, and its memory remains in use. Subsequently, the server is resumed, only to become coincidentally isolated from the cluster due to a network partition, resulting in a failed start. Hence, the server crashes without the chance to release the memory allocated. This bug happens due to a flawed logic design; specifically, the allocated memory can only be released when the process is in the running status, and the allocated memory cannot be released before running. This logic design is reasonable in stable environments without faults, as only the running server may have allocated memory. However, in this extreme environment, the shortcoming in the logic is exposed.

Bug #8 of Dqlite is another memory leak issue, similar to Bug #2 in Braft. It remained hidden in Dqlite for approximately four years and affected all its release versions before we found it. Bug #2 in Braft and Bug #8 in Dqlite both evaded the rigorous testing efforts by their developers, demonstrating how our tool Mallory can significantly reduce the exposure of systems to vulnerabilities.

Case Study: Bug #11 in Dqlite. Although Jepsen is already part of Dqlite's Continuous Integration process, Mallory has managed to expose several new bugs in Dqlite. The developers have expressed a keen interest in Mallory and are awaiting its open-source release so that they can incorporate it into their testing.

Bug #11 in Dqlite is caused by the snapshotting of uncommitted logs, and it is reminiscent of the membership rollback bug shown in Figure 5.1. The schedules required to trigger these bugs are quite similar, but Bug #11 is not triggered by the configuration change. Specifically, the event ① involves one plain read/write log entry instead of the configuration change. After this event, the cluster undergoes the same sequence of environment faults, including a network partition ($\#\{S_1, S_2\}$, $\#\{S_3, S_4, S_5\}$), leader S_1 crashing, and network healing. As a result of these faults, server S_2 ends up with conflicting entries with the leader S_3, which must be removed. However, the conflicting entries are in a snapshot, which makes the removal fail. This failure leads to the server becoming unavailable. Although the schedule to trigger Bug #11 is slightly shorter than that of the membership rollback bug in Figure 5.1,

Jepsen, which adopts a random search strategy, failed to expose it during our experiments within the allotted time. In contrast, Mallory, guided by the policy learned with Q-learning, successfully exposed this bug. This is easily explained considering the number of states explored by Jepsen and Mallory shown in Figure 5.3. Within 24 hours, Jepsen only covered 600 states, while Mallory explored over 1,000 states. With the capability to explore more novel behaviors, Mallory significantly increased its chances of exposing bugs. This instance demonstrates the effectiveness of the policy learned using Q-learning in guiding Mallory to explore more behaviors and, consequently, enhance the likelihood of bug exposure.

Mallory found 22 zero-day bugs in rigorously tested implementations, and 18 bugs out of them have been confirmed by their developers. 10 of these 18 bugs correspond to security vulnerabilities, and out of these 6 CVEs have been assigned.

5.5 Conclusion

In this chapter, we proposed Mallory—the first adaptive greybox fuzzer for distributed systems. The key insight behind Mallory's design is to summarise the runtime behaviour of the distributed system under test in the form of Lamport-style *timelines* that capture the causality of events, and use the timelines to define a feedback function for guiding the search for bugs. We evaluated Mallory on six widely-used and rigorously-tested industrial distributed system implementations such as Dqlite and Redis. The experimental results show the effectiveness and efficiency of Mallory in achieving significantly higher state coverage and faster bug-finding speed than the state-of-the-art tool Jepsen. Finally, Mallory discovered 22 previously unknown bugs (10 new vulnerabilities among them) which have contributed to new CVEs.

Chapter 6

Capturing Different Environmental States

Computer programs are not executed in isolation, but rather interact with the execution environment, which drives the program behaviors. Software validation methods thus need to capture the effect of possibly complex environmental interactions. Program environments may come from files, databases, configurations, network sockets, human-user interactions, and more. Conventional approaches for environment capture in symbolic execution and model checking employ environment modeling, which involves manual effort. In this work, we take a different approach based on an extension of greybox fuzzing. Given a program, we first record all observed environmental interactions at the kernel/user-mode boundary in the form of system calls. Next, we replay the program under the original recorded interactions, but this time with selective mutations applied, in order to get the effect of different program environments—all without environment modeling. Via repeated (feedback-driven) mutations over a fuzzing campaign, we can search for program environments that induce crashing behaviors. Our $\mathcal{E}$FUZZ tool found 33 previously unknown bugs in well-known real-world protocol implementations and GUI applications. Many of these are security vulnerabilities, and 16 CVEs were assigned.

6.1 Introduction

Computer programs are not executed in isolation, but rather interact with a complex *execution environment* which drives the program behaviors. Inputs received from the environment, such as configuration files, terminal input, human-user interactions, and network sockets, directly affect the internal program state which, in

turn, governs how the program executes. Outputs sent to the environment, such as terminal output and sockets, provide useful clues that reflect these program states and behaviors. If the program is buggy, some environmental interactions may cause the program to crash or otherwise misbehave. *Fuzz testing* (or *fuzzing*) [20] is a widely-used automatic method that can find such program (mis)behavior. Ideally, fuzzing should be executed under different execution environments to comprehensively explore diverse program behaviors. However, capturing the effect of complex environments has always been a challenge for all program-checking methods—be it software verification, analysis, or testing.

In this work, we take a fresh look at the problem of program environment capture, and provide a solution in the context of fuzz testing. Greybox fuzzing uses a biased random search over the domain of program inputs to find crashes and hangs. We aim to extend greybox fuzzing over the *full environment* without resorting to modeling. Our approach is to first run the program normally, but also to *record* all interactions between the program and environment that can be observed at the user/kernel-mode boundary (e.g., *system calls*). These interactions serve as the set of initial seeds. Next, the program is iteratively run again as part of a fuzzing loop, but this time *replaying* the original recorded interactions. During the replay, the fuzzer will opportunistically mutate the interactions recorded for system calls to observe the effect of environments different from that of the original recording. In effect, the program environment is fuzzed at the system call layer. Our approach does not conduct any abstraction of possible environments; it (implicitly) works in the space of real concrete environments.

We present a generic approach for fuzzing the full program environment. Existing greybox fuzzers are limited to fuzzing *specific* input sources, such as an input file specified by the command line (e.g., AFL [167] and AFL++ [57]), or a network socket over a specific network port (e.g., AFLNet [126] and Nyx-Net [141]). Our approach extends the scope of fuzzing to include *all* environmental inputs, meaning that any input is considered a fuzz target, regardless of source. We also propose a generic fuzzing algorithm to (implicitly) generate different program environments, thereby exploring diverse program behaviors. We have implemented our approach of program environment fuzzing in the form of a new greybox fuzzer called $\mathcal{E}$fuzz. We evaluate $\mathcal{E}$fuzz against two categories of user-mode programs under Linux: network

protocol implementations and GUI applications, both of which are considered challenging subjects for existing fuzzers [57, 20]. In real-world and well-known applications, such as Vim and GNOME applications, $\mathcal{E}$FUZZ found 33 previously unknown bugs (24 bugs confirmed by developers, which include 16 new CVEs). The bugs found include null-pointer dereferences, buffer overreads, buffer overwrites, use-after-frees, and bad frees, all triggered by fuzzing diverse environmental inputs including sockets, configuration files, resources, cached data, etc.

In summary, we make the following main contributions:

- We propose a new greybox fuzzing methodology to capture the effect of complex program environments—all without environment modeling or manual effort.
- We present a new fuzzing algorithm based on the full environmental record and replay at the user/kernel-mode boundary.
- We implemented the approach as a generic fuzzer ($\mathcal{E}$FUZZ) capable of testing various program types, including two categories of recognized challenging subjects. In our evaluation, we found 33 previously unknown bugs and received 16 CVE IDs.

Our tool is publicly available at

https://github.com/GJDuck/EnvFuzz

6.2 Background and Motivation

6.2.1 Motivating Example

As an initial motivating example, we consider a calculator application implemented using a *Graphical User Interface* (GUI). A human user makes inputs in the form of mouse movements, keystrokes, button presses, *etc.*, and the application reacts by generating outputs that update the graphical display. For example, by pressing the button sequence $\langle$`1`, `+`, `2`, `=`$\rangle$, the application responds by displaying the answer (`3`).

Like all software, the calculator application may contain bugs, and these bugs can be discovered using automatic software testing methods such as fuzzing. For example, a fuzzer could apply the mutations (`+`)→(`/`) and (`2`)→(`0`) to construct a new button press sequence ⟨`1`, `/`, `0`, `=`⟩ that will cause a crash (`SIGFPE`) if the calculator application were to not properly handle division by zero. Although most mutations will be benign (non-crashing), typical fuzzers mitigate this with a combination of high throughput (e.g., 100s of executions per second), program feedback (e.g., code coverage), and power scheduling (e.g., controlling mutation counts), increasing the likelihood of finding crashing inputs within a given time budget.

However, most existing greybox fuzzers, such as AFL [167, 57] and AFLNet [126], do not consider all input sources when producing mutated inputs. These fuzzers only target a specific class of inputs by default. For example, AFL only targets standard input (`stdin`) or a file specified by the command line. Similarly, AFLNet only targets network traffic over a specific port for a specific popular protocol (e.g., `ftp` and `smtp`). Essentially, these existing fuzzers use a simplified program environment, where program behaviors (and potential crashes) are driven by a single input source, and it is up to the tool user to decide *which* input source to fuzz. All other input sources are considered as "static", i.e., unmutated and unchanged between test cases. Furthermore, most existing fuzzers are specialized to specific *types* of inputs, such as regular files or popular network protocols.

In reality, most programs have a more complicated interaction with the environment beyond that of a single input source. For example, if we consider the `gnome-calcuator` application as part of the GNOME Desktop Environment for Linux. This application will open 706 distinct file descriptors under a minimal test (i.e., open and close the application window), including:

- 674× regular files, including configuration, cache, and GUI resources (icons/-fonts/themes).
- 7× socket connections to the windowing system, session manager, and other services.
- Miscellaneous (e.g., special files, devices, and `stderr`).

The calculator application with a full environment is illustrated in Figure 6.1 (a).

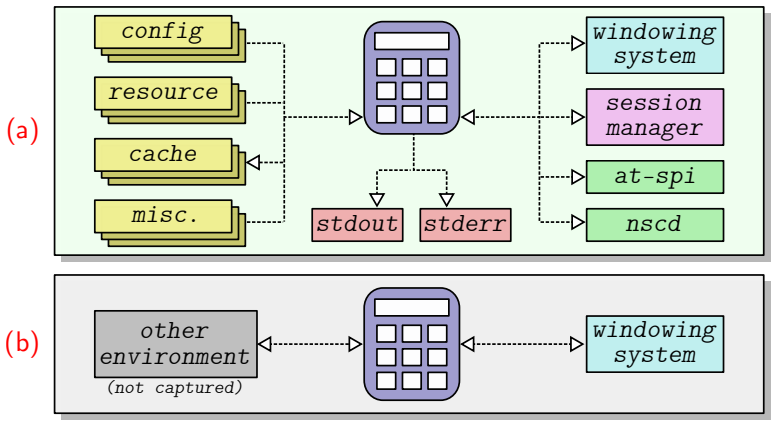


Figure 6.1: (a) is a calculator application with the *full* environment, including regular file I/O, standard streams, and socket/event fds to various system services. (b) is a *simplified* environment with a single input/output (windowing system socket), where all other interactions are not captured.

6.2.2 Limitations of Conventional Fuzzing

Fuzzing requires two key decisions to be made before use:

- *Input Selection*: Which input should be fuzzed?
- *Environment Modelling*: How to handle other inputs?

For the button-press example, the fuzz target would be the windowing system socket over which button-press events are received. Thus, for the purposes of fuzzing, we use a simplified environment as illustrated in Figure 6.1 (b). In the case of the calculator application, the simplified environment is somewhat naïve, since the target socket is only one of many possible input sources (706 possibilities). Consequently, only a small fraction of the actual environment is subjected to fuzzing. Assuming, for the sake of example, that the windowing system socket is selected. The next step is to choose a fuzzer. Since the input is a socket rather than a file, a network protocol fuzzer, such as AFLNet, will be suitable. AFLNet works by fuzzing inbound network messages and parsing the response codes from outbound messages as feedback to guide the fuzzing process. However, AFLNet only supports a limited set of pre-defined network protocols, and this does not include support for windowing system protocols. Even if the necessary protocol support is available, the environment beyond the fuzz target must still be handled.

One approach is to fix all the remaining environments as most existing fuzzers do, where the program is consistently checked within a single environment across test cases. Obviously, this approach limits the explored program behaviors. Moreover, in some cases, such as fuzzing the calculator application and other GUI applications, this approach is impractical for existing fuzzers. Handling regular file I/O is relatively straightforward since files can be read from disk for each executed test case, with outputs easily discarded (e.g., piped to `/dev/null`). However, a program can interact with more than one external service, such as session managers, service daemons, and even human users. In order to execute a single test case as part of the fuzzing process, the system-environment interactions would need to be "reset" for each individual test case—something known to be slow. For human-driven inputs, this also implies that a human-in-the-loop is necessary, since the fuzzer needs human interaction to proceed from one test execution to another.

Another approach is to build a *model* of possible environmental interactions. However, modeling is non-trivial. For example, each external service will typically use its own specialized protocol, and there can be an arbitrary number of services in the general case. Furthermore, any model would need to be *accurate*, as an invalid interaction may cause the test subject to terminate early due to an error condition, thus hindering reaching potential bug locations. Environment modeling is a known problem in the context of model checking and symbolic execution. Many existing works [10, 27] address it by modeling the environment *manually*. However, these approaches tend to be limited to specific problem domains and lack scalability for the general case.

6.2.3 Core Idea

We now describe our approach. We do not explicitly enumerate all possible environments in a search space and then navigate this very large search space. Our approach (below) is more implicit.

- *Input Selection*: All environmental inputs are fuzzed.
- *Environment Modelling*: Avoid modelling. The inputs are executed under a given environment and the effect of different environments is captured by mutating the environmental interactions represented by system calls.

For the calculator example, we consider all environmental inputs as fuzz targets regardless of *type*. Thus, various files (e.g., configuration, cache, and resource), sockets (e.g., those utilized by the windowing system), and any other input sources, are abstracted as generic *inputs* to subject fuzzing, eliminating the need for special handling. Since the whole environment is the fuzz target, any remaining residual environment is essentially eliminated, avoiding the need for additional modeling.

Building upon this concept, our approach first *records* all environmental interactions between the target program and its environment. Subsequently, the program is iteratively run again as part of a fuzzing loop, this time by *replaying* the interactions from the previous recording to substitute the original environment. Instead of replaying exactly the original recording, some of the interactions are *mutated* to implicitly generate the effect of different program environments, potentially uncovering new program behaviors.

To record the full environmental interactions, our approach works at the system call layer. This is motivated by the observation that most user-mode applications in Linux interact with the environment through the kernel/user-mode interface. For example, button presses and corresponding GUI updates flow through `recvmsg` and `sendmsg` systems calls over a socket. Similarly, pipes, streams and file I/O flow through standard `read` and `write` system calls. As a result, by recording system calls, we also record the full environmental interactions of the program, including system-environment interactions and human interactions, regardless of the underlying input type or the nature of the program.

6.3 System Overview

Based on our core idea, we design a generic program-environment fuzzer using a *record and replay* methodology. This fuzzer, called $\mathcal{E}$FUZZ, is illustrated in Figure 6.2. At a high level, $\mathcal{E}$FUZZ consists of two phases: the *record* phase (Phase I) records the full interactions between the program-under-test P and its environment $\mathcal{E}$, and the *replay-with-greybox-fuzzing* phase (Phase II) replays and fuzzes the recorded interactions. Phase II captures the effect of different program environments and uncovers new program behaviors.

Phase I: Recording. In the *recording* phase, the program P is run normally

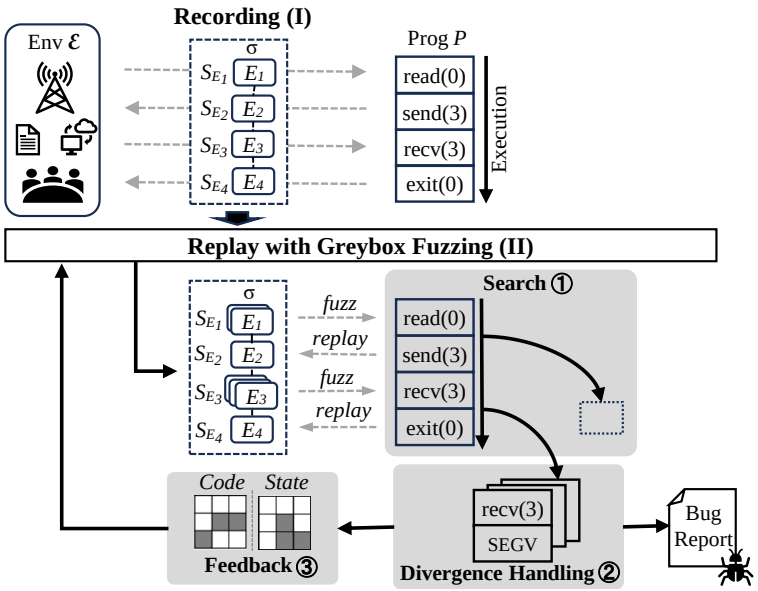


Figure 6.2: Overview of the program environment fuzzer $\mathcal{E}$FUZZ.

within some test environment $\mathcal{E}$. The program interacts with the environment (files, sockets, human input, etc.) via a sequence of system calls, which are intercepted by $\mathcal{E}$FUZZ and saved into a *recording* σ. Here, σ is an in-order sequence of *records* (e.g., $\sigma = [E_1, E_2, E_3, E_4]$), where each record E stores all of the necessary details for reconstructing each corresponding environmental interaction in Phase II. These details include the system call number, system call arguments, buffer contents (if applicable), and the return value. These records are then saved into a respective seed corpus ($\mathcal{S}_E = \{E\}$) to serve as the initial seeds for the subsequent replay and greybox fuzzing.

Phase II: Replay with Greybox Fuzzing. Phase II combines environment replay with greybox fuzzing. The idea is two-fold:

1. Faithfully *replay* the recorded environment interactions to reconstruct (deep) program states observed during Phase I;

2. *Fuzz* each reconstructed state using greybox fuzzing.

Faithful replay works by re-running the program, but using the recording σ as a substitute for the original test environment $\mathcal{E}$. This again works by intercepting system calls, but this time the corresponding record ($E \in \sigma$) is *replayed* as a substitute for the real interaction.

To uncover different program behaviors for bug discovery, the core of $\mathcal{E}$FUZZ lies in greybox fuzzing. However, unlike traditional fuzzers, $\mathcal{E}$FUZZ works by fuzzing the recorded environmental interactions ($E \in \sigma$), rather than targeting specific files, as with AFL, or sockets, as with AFLNET. This works as follows: for each state reconstructed, $\mathcal{E}$FUZZ faithfully replays the next environmental interaction E in sequence from σ to advance state reconstruction. *In addition*, $\mathcal{E}$FUZZ selects seeds from the seed corpus $\mathcal{S}_E$, assigns energy, and introduces mutations to generate *mutant* interactions. Each mutant interaction is replayed in a *forked* branch of execution, where the program's behavior is observed (see Figure 6.2 ①).

Following the execution of a mutant interaction, the program behavior may *diverge* significantly from the original recording. Such divergence can include the program invoking different system calls, or invoking existing system calls but in a different order. For example, as shown in Figure 6.2, the `exit(0)` system call could be changed into `recv(3)`. Such behavior divergence presents a technical challenge for advancing replay, since only the original recording (σ) is available. Indeed, the main goal of fuzzing is to explore novel (divergent) program behaviors in order to discover bugs. To resolve this challenge, $\mathcal{E}$FUZZ introduces the notion of *relaxed* replay (as opposed to *faithful* replay) that is designed to progress divergent program execution after mutation (see Figure 6.2 ②).

At the end of each execution, similar to traditional greybox fuzzing, program feedback is used to determine *interesting* mutant interactions (see Figure 6.2 ③). The interesting interactions are saved into the seed corpus for future mutation. Additionally, mutations triggering program crashes are saved and reported to the user. The fuzzing campaign repeatedly iterates over reconstructed program states until a time budget is reached.

6.4 Environment Fuzzing

We describe and explain the $\mathcal{E}$FUZZ algorithm in this section.

6.4.1 Environment Recording and Replay

For recording the environment, $\mathcal{E}$FUZZ implements a system call *interceptor routine* that acts as a proxy (i.e., "man-in-the-middle") between the program P and

the kernel. Thus, when the program invokes a system call, such as a `read` or `write`, the call will be routed to the interceptor routine. The routine first *forwards* the system call to the underlying kernel and waits for the result. Once the underlying system call completes, the interceptor routine will then save relevant information about the system call into a record E, including: the system call number (e.g., `read` and `write`), arguments (e.g., file descriptor, buffer pointer, and buffer size), buffer contents (where applicable), current thread ID, and the return value. The system call result is then returned back to the program P, which continues executing as normal.

Each individual record E represents an interaction between the program P and its environment $\mathcal{E}$. During recording, each record is appended onto an in-order sequence σ, otherwise known as the *recording*, and is also saved into the respective seed corpora. The recording σ contains the information necessary to reconstruct all program states previously observed during the recording phase. For *faithful* replay, the program is run once more, but this time the interceptor routine instead *replays* (rather than forwards) the previously-recorded E. For fuzzing, the original record is replayed, but with one or more mutations applied first. Such mutations represent modified environmental interactions, and can change the program behavior.

We now use an example to illustrate this process. Suppose that during recording, the program P calls `read`$(0, \mathit{buf}, 100)$, which is forwarded to the kernel, and the user enters "`quit\n`" into `stdin` (fd=0). The interceptor routine will record the returned buffer contents ("`quit\n`") and the returned value (=5 bytes read) into a record E. Then, during replay with greybox fuzzing:

- For *faithful* replay, the program P is re-run, and calls the same `read` system call as before. Instead of forwarding the system call to the kernel, the interceptor routine copies the previously recorded contents from E, copying "`quit\n`" into *buf* and returning 5. This causes the program's execution to proceed equivalently to the original recording.

- For fuzzing, the record E is first *mutated* before it is replayed. For example, the buffer contents could be mutated into "`quip\n`", and this will likely cause the program's behavior to diverge as if this were the original user interaction—possibly exposing new behaviors and bugs.

```
int main()   {
   char buf[SIZE]; int r;
   r = read( 0, buf, SIZE); ...;
   r = send( 3, buf, r);    ...;
   r = recv( 3, buf, SIZE); ...;
   r = send( 3, buf, r);    ...;
   r = recv( 3, buf, SIZE); ...;
   r = write(1, buf, r);    ...;
   exit(0); }
```

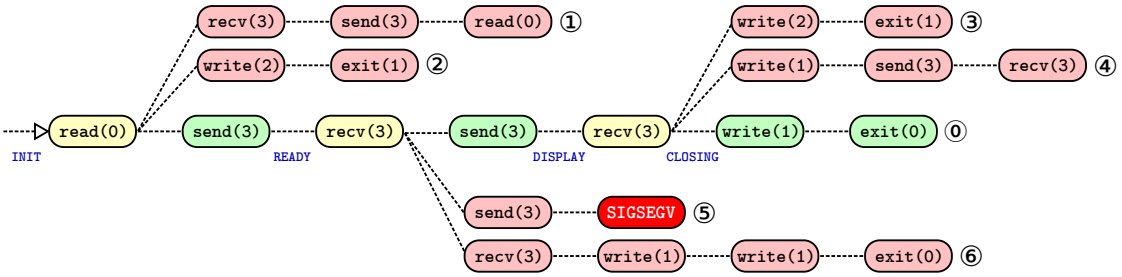


Figure 6.3: Illustration of the underlying fuzzing algorithm. Here, the example program reads from file descriptor `0`, then interacts with socket (file descriptor `3`). The fuzzer faithfully *replays* a previously *record*ed interaction ⓪, as well as several *mutant* interactions ①/②/③/④/⑤/⑥. Each mutant interaction is generated by mutating at least one input system call from the faithful replay. This causes the program's behavior to diverge, including exit with error ②/③, system call reordering ①/⑥, new I/O system call ④, and a crash ⑤. The program state {`INIT`, `READY`, `DISPLAY`, `CLOSING`} between select system calls is also illustrated.

The mutation is applied to the buffer contents of *input* system calls (e.g., `read`) as this can affect the program behaviors and cause behavior divergence. Other system calls, that do not affect the program behaviors (e.g., `write`), will not be mutated. The combination of faithful replay, and replay with mutation, forms the basis of $\mathcal{E}$FUZZ's greybox fuzzing algorithm.

6.4.2 Reflections on Search Challenges

After the recording phase, $\mathcal{E}$FUZZ has collected a set of initial seeds representing real environment interactions. Using these as a basis, $\mathcal{E}$FUZZ employs greybox fuzzing to generate new interesting seeds representing interactions with new program environments—each with the potential to induce novel program behaviors. In designing an efficient algorithm for searching the program environment space, there are two main challenges: (i) *statefulness*: how to effectively explore deep program

behaviors? (ii) *throughput*: how to maintain high fuzzing throughput?

Challenge (i): Statefulness. To better understand Challenge (i), we consider a calculator program that interacts over multiple input and output sources, as shown in Figure 6.3. The program begins in an `INIT` state, where it first parses a configuration file (file descriptor `0`), then creates a user interface (GUI) by sending a message over the windowing system socket (file descriptor `3`). The program then transitions into a `READY` state—i.e., waiting to accept mathematical expressions from the user interface. Subsequently, the program processes one user input expression (received from `3`), and then sends the result back to the interface (send to `3`), and the program transitions into the `DISPLAY` state. Finally, the user closes the interface (received from `3`), and the program transitions into a `CLOSING` state. Here, the program writes a message to the terminal (file descriptor `1`) before exiting. Our example is a simplification for brevity, as a real calculator program will typically interact with thousands of system calls, and may have many more internal states.

At the layer of system calls, the program is *stateful*, as it accepts a sequence of environmental inputs and adjusts its state accordingly. Some program behaviors are only reachable by specific states, which are in turn reachable only through specific input sequences. When fuzzing stateful programs, greybox fuzzers aim to exercise each observed state in order to explore the neighborhood of potential program behaviors, thereby having a greater chance to expose new bugs. However, state identification remains a challenging problem for fuzz testing in general. Existing works [7, 126, 130, 8] propose several heuristics for program state detection. For example, IJON requires states to be manually annotated, whereas AFLNet utilizes response codes from outbound messages to detect new states for well-known protocols. Either way, existing approaches require manual effort or are specialized to specific input sources.

We propose a generic approach that considers all input sources, such as files, sockets, and pipes, and consider how they affect program states. We consider each input system call as a *potential* state transition. For example, in Figure 6.3, after executing each input system call in sequence, the program transitions from the `INIT` to `READY` state, then from the `READY` to `DISPLAY` state, and finally from the `DISPLAY` to `CLOSING` state. Thus, each input can be fuzzed as a distinct transition between states, regardless of the input type (file, socket, etc.). However, some of the inputs

may not trigger new transitions. This is mitigated by the power schedule [22], where inputs that fail to induce real state transitions are also less likely to expose new program behaviors observable via program *feedback*. As such, the corresponding input will be assigned less *energy*, and is naturally deprioritized for future mutations.

Challenge (ii): Throughput. To reach each observed state, $\mathcal{E}$FUZZ conducts a faithful replay of the recorded system calls. Upon reaching a state (i.e., before executing the corresponding input system call), $\mathcal{E}$FUZZ applies mutations to explore the neighboring program behaviors. If the fuzzer must always replay system calls from the root point, then multiple system calls need to be replayed to reach a specific state. For example, in the calculator example, a total of four system calls must be faithfully replayed to reach the `DISPLAY` state. This can significantly slow fuzzing throughput, especially for real-world examples where thousands of system calls may be required to reach a given state. To address this challenge, we propose a tree-based fuzzing algorithm that avoids (re)executing the same prefix sequence of system calls repeatedly. The algorithm is illustrated by the tree shown in Figure 6.3. Specifically, the original recording is faithfully replayed (without mutation), forming the "spine" of the tree, which is represented by the middle trace ⓪. Upon reaching an input system call, $\mathcal{E}$FUZZ additionally `fork`s-off some number of *mutant* traces, creating the "branches" of the tree (e.g., ①/②). Each branch starts by replaying an original input with one or more *mutation* operators applied, and may involve further mutations of subsequent inputs. After executing each branch, $\mathcal{E}$FUZZ continues the faithful replay to grow the spine until the next input point, after which $\mathcal{E}$FUZZ `fork`s-off more branches (e.g., ⑤/⑥). The process repeats once more (e.g., ③/④) before ⓪ terminates.

6.4.3 Fuzzing Search Algorithm

Based on the environment recording and replay technique, along with the efficient search strategy, we introduce a novel environment fuzzing algorithm, illustrated in Algorithm 5. The recording is shown in line 2 of Algorithm 5. After the recording, the program is executed normally, but with the interceptor routine `FuzzSyscall` replacing the standard system call interface (line 8). There are two main cases to consider: the replay is in the spine or in a branch (e.g., see Figure 6.3), and the

Algorithm 5: Program Environment Fuzzing Algorithm.

Input : Program P, environment interaction $\mathcal{E}$
Output : Crashing events $\mathcal{C}_{\text{✗}}$
Globals : Input-specific corpora $\mathcal{S}_E$

```
1:  func EnvFuzz(P, E):
2:      σ ← Record(P, E)                                   ▷ Recording
3:      for E ∈ σ do S_E ← {E}
4:      repeat
5:          FuzzReplay(P, σ)
6:      until timeout reached or abort
7:  func FuzzReplay(P, σ):                                 ▷ Replay with Fuzzing
8:      exec(P_[replace syscall with FuzzSyscall, isBranch←false], σ)
9:  func FuzzSyscall(e):                                   ▷ Tree-based Search
10:     if isBranch then
11:         return EmulateSyscall(e, σ)                    ▷ Divergence Handling
12:     else /* if isSpine then */
13:         E ← head(σ); σ ← tail(σ)
14:         if ¬isInput(e) then return ReplaySyscall(E)
15:         for E′ ∈ S_E, i ∈ 1..energy(E′) do
16:             E″ ← mutate(E′)
17:             pid ← fork()
18:             if pid = 0 then                            ▷ In child:
19:                 isBranch ← true
20:                 return ReplaySyscall(E″)
21:             else                                       ▷ In parent:
22:                 waitpid(pid, &status)
23:                 if isCrash(status) then    add E″ to C_✗
24:                 if isInteresting(E″) then add E″ to S_E
25:         return ReplaySyscall(E)                        ▷ Grow Spine
```

program starts with running in the spine (*isBranch*←*false*). For the spine of the tree (line 12-line 25), $\mathcal{E}$FUZZ retrieves the next record E to be processed (line 13). For non-input system calls (e.g., `write`), the original record E is faithfully replayed "as-is" (line 14). Conversely, all input system calls (e.g., `read`) are treated as potential fuzzing targets, and a greybox fuzzing algorithm is used (line 15–line 24). Specifically, for each record E corresponding to the input syscall, $\mathcal{E}$FUZZ will iterate over each seed E' from corpus $\mathcal{S}_E$. For each E', $\mathcal{E}$FUZZ applies one or more standard *mutation operators*, to further mutate the input buffer contents, and thereby generating a new

seed E'' (line 16). The current implementation uses mutation operators from other fuzzers, e.g., `havoc` from AFL [167, 126]. The number of mutations is controlled by a power schedule (*energy*) (line 15).

To execute the new seed E'', the algorithm first forks the program into a *parent* and *child* process (line 17). The seed E'' is executed in the child, forming a *branch* of the tree (e.g., see Figure 6.3), while the parent waits for the child's termination (line 22). After applying a mutation in the child, the interceptor routine `FuzzSyscall` processes the subsequent system calls using a different method (line 11), which will be discussed in Section 6.5. Following the termination of the child, the parent examines the result. Crashing mutations are saved into a special corpus $\mathcal{C}_{\text{✗}}$ that forms the output of Algorithm 5 (line 23). Otherwise, the fuzzing feedback (discussed in Section 6.4.4) is used to determine whether the mutated seeds are *interesting* or not, and interesting seeds are saved into $\mathcal{S}_E$ for future mutation see line 24 ; the decision on whether a seed is interesting or not, is conducted based on fuzzing feedback which is discussed in the next subsection. Subsequently, $\mathcal{E}$FUZZ grows the spine by continuing faithful replay (line 25). After the fuzzing campaign is complete, the $\mathcal{E}$FUZZ infrastructure also supports replaying any of the $\mathcal{C}_{\text{✗}}$ corpus to reproduce discovered bugs.

An illustration of this fuzzing algorithm on a simple example program appeared in Figure 6.3.

6.4.4 Fuzzing Feedback

Greybox fuzzing relies on feedback to select "interesting" seeds (line 24 in Algorithm 5) to guide the search towards novel program behaviors, thereby increasing the likelihood of discovering bugs [89]. A common form of feedback is *branch coverage*, as used by many modern fuzzers [167, 57]. Here, seeds that cover new branches (code paths) have the potential to explore different behaviors, and thus are considered interesting and saved into the corpus for future mutation. Most fuzzers collect branch coverage feedback using compiler instrumentation (e.g., `afl-gcc`). Instrumentation can also be inserted directly into binary code using *static binary rewriting*, such as with E9AFL [61]. $\mathcal{E}$FUZZ supports branch coverage feedback and operates directly on binaries to maximize generality.

For the case of stateful programs, branch coverage alone is generally considered insufficient [7, 126]. As such, *state feedback* has been proposed in collaboration with branch coverage to guide the fuzzing process. Here, seeds that cover new state transitions are also considered "interesting" and are similarly added to the corpus. However, as discussed in Section 6.4.3, automatically inferring program states is challenging, especially for binary code. Our approach is to treat each input message as a *potential* state transition. We leverage program *outputs* (e.g., `write`) as a proxy for detecting states. Our heuristic is that, under certain inputs, a program will generate output that is contingent on its internal states, and thus outputs can provide insights into these states. To mitigate the impact of outputs with unknown structures/formats, we employ locality-sensitive hashing and clustering based on the *Hamming distance* [114, 56]. $\mathcal{E}$FUZZ can utilize both branch and state feedback to guide the search.

6.5 Relaxed Replay for Divergence

After a mutated input is replayed in a branch, it is common for the program's behavior to *diverge* from the original recording, as illustrated by the branches ①,...,⑥ in Figure 6.3. Divergence could include: exiting with error ②/③, system call reordering ①/⑥, new system call invoking ④, or even the program crashing ⑤. For example, suppose the last input from Figure 6.3 receives a command "`quit\n`" from the socket, causing the program to enter the `CLOSING` state and exit. However, mutant replay could change the command to "`quip\n`", foiling the state transition, and causing the program's behavior to diverge from the original recording.

This poses a challenge that is described as follows. During the recording phase, $\mathcal{E}$FUZZ will construct an in-order sequence of records σ. Assuming that $\sigma = [\sigma_1, E, \sigma_2]$, where E is an input, then during the fuzzing phase, $\mathcal{E}$FUZZ faithfully replays the prefix σ_1 (as part of the spine) before reaching E. Next, $\mathcal{E}$FUZZ mutates E to generate one (or more) mutant E', after which E' is replayed as a substitute for E. After replaying E', the faithful replay of σ_2 may no longer be possible due to program behavior divergence, i.e., the mutant sequence $\sigma' = [\sigma_1, E', \sigma_2]$ may be *infeasible*. The problem is that $\mathcal{E}$FUZZ only has the original recording σ to work with.

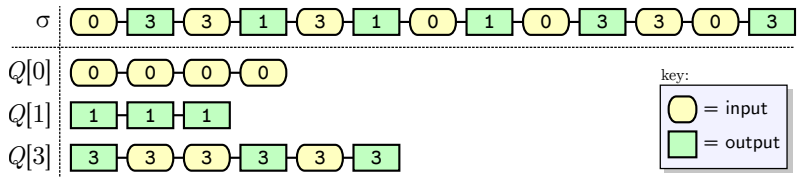


Figure 6.4: Illustration of the *global* ordering (σ) for faithful replay and a *local* ordering (Q) for relaxed replay. The relaxed replay partitions σ into a set of miniqueues ($Q[fd]$) indexed by the file descriptor, each of which defines a local ordering specific to each *fd*.

To address this problem, we introduce the notion of *relaxed replay*. The key idea is to use system call *emulation* (line 11 in Algorithm 5) to construct continuations of program execution that diverge from σ. Relaxed replay uses a set of *emulation routines*, one for each syscall number, where each routine takes the syscall arguments and returns a result (i.e., return value, buffer contents) on a "best-effort" using available information. Unlike faithful replay, these routines can be called at any time and in any order, and do not necessarily need to follow the original recorded system call ordering. Crucially, the emulation routines should only return *plausible* results—i.e., there exists a real (modified) environment $\mathcal{E}'$ from which the result could occur. Plausibility is necessary to avoid *false positives*—i.e., crash reports that are irreproducible under any real environment. For plausibility, we generalize assumptions used by existing fuzzers, wherein any I/O modification (e.g., in AFL) and reordering (e.g., in AFLNet) are considered plausible. We now discuss these two cases in detail.

6.5.1 Relaxing I/O System Call Ordering

After mutation, programs often invoke I/O system calls in a different order from that of the original recording. To handle this case, our approach is to first *partition* σ into a set of *miniqueues* $Q[fd]$, with one miniqueue specific to each I/O source (i.e., *file descriptor*, *fd*). The approach is illustrated by example in Figure 6.4. Here, under the *global ordering* (σ) for faithful replay, only a read system call from file descriptor 0 can be serviced. However, after mutation, the program may attempt I/O on a different file descriptor. To handle such cases, our approach allows I/O system calls to be directly serviced from the corresponding miniqueue $Q[fd]$ under a

local ordering specific to each *fd*, rather than the original global ordering (σ). The partitioning and local ordering is plausible under the assumption that I/O system calls can be reordered.

It is also common for programs to use the `poll` system call[1] to query which I/O operations are currently possible. Relaxed replay must also handle the poll system call using emulation. The algorithm is shown in Algorithm 6, and is a concrete example of an emulation routine. Here, poll is emulated based on the *current* state of Q (line 3–line 12) and returning:
(i) *End-of-file* (`POLLHUP`) for an empty miniqueue (line 7);
(ii) *input ready* (`POLLIN`) or *output ready* (`POLLOUT`) if the queue head matches the requested event (line 9–line 10);
(iii) `0x0` (a.k.a. no event) otherwise.
If at least one of the returned events is non-zero, then the poll operation successfully completes (line 12) and execution continues. Otherwise, the poll operation will block. To avoid blocking, the algorithm heuristically *picks* a file descriptor and *reorders* the corresponding miniqueue (line 13), allowing Algorithm 6 to always terminate (without blocking) in the next iteration of the outer-loop.

6.5.2 Relaxing I/O System Calls

Input system calls are emulated by an implicit `poll` operation, followed by popping the corresponding miniqueue $Q[fd]$. The popped record is replayed, possibly subject to further mutation. If the implicit poll operation indicates the miniqueue is empty (`POLLHUP`), the input system call returns 0 indicating an end-of-file (`EOF`).

Emulated output system calls similarly pop the corresponding miniqueue, but always succeed even if the queue is empty. This handles the common case where a mutation causes the program to generate additional output, such as a warning or error message that is not present in the original recording σ. Modified or extraneous outputs can generally be ignored, as outputs do not affect the program behavior. However, outputs do provide useful hints about the program state, which is used as fuzzing feedback.

[1]See the `poll` manpage for more information.

Algorithm 6: Emulated `poll` routine.

Input : Array of `pollfd` structs, Q derived from σ
Output : Number of non-zero *revents*

```
1:  func EmulatePoll(fds, Q):
2:      while true do
3:          r ← 0; h ← 0
4:          for i ∈ 0..|fds|−1 do
5:              E ← head(Q[fds[i].fd])
6:              if E = EOF then
7:                  fds[i].revents = POLLHUP; h++
8:              else
9:                  fds[i].revents = fds[i].events &
10:                     (isInput(E)? POLLIN: POLLOUT)
11:                 r += (fds[i].revents? 1: 0)
12:         if r > 0 ∨ h > 0 then return r
13:         fd ← pick(fds, Q); Q[fd] ← reorder(Q[fd])
```

6.5.3 Relaxing Non-I/O System Calls

Other system calls are handled using heuristics, such as:

- *Emulate*: emulate (plausible) effects of the system call;
- *Forward*: pass the system call "as-is" to the underlying O/S;
- *Fail*: fail the system call with an error condition (e.g., `ENOSYS`).
- *Exit*: as a last resort, terminate the branch with `exit`.

In addition to I/O system calls, $\mathcal{E}$FUZZ also implements several specialized emulation routines for other common system calls, including time (e.g., `clock_gettime`/etc.) and thread-related (e.g., `futex`/`clone`/etc.) system calls. For example, time-related system calls are handled by emulating a global *monotonically increasing* clock t. The clock t is first initialized to the last time observed in the recording before the branch, and t is then incremented for each subsequent emulated system call after the branch. This ensures that emulated time-related system calls always return *plausible* results—i.e., the system time always flows forwards. The system calls related to memory management (e.g., `brk`/`mprotect`/`madvise`/etc.) are generally *forwarded* to the underlying O/S "as-is" without special handling.

Sometimes neither system call emulation nor forwarding is applicable. For example, due to behavior divergence, the program may attempt to access a file that does not appear in the original recording (σ). As such, $\mathcal{E}$FUZZ has no information about the file contents, or whether the file even exists. In such cases, relaxed replay can *fail* the system call with an error (e.g., `ENOSYS`), allowing for execution to proceed and giving the program a chance to recover. As a last resort, relaxed replay will exit the program if no other alternative is possible. This occurs when a program ignores failure, e.g., by re-invoking the same system call again in a loop.

6.6 Implementation

We implemented the approach of $\mathcal{E}$FUZZ as a generic program environment fuzzer that can handle a diverse range of user-mode Linux applications, including GUI applications and network servers. $\mathcal{E}$FUZZ is built on top of a full environment record and replay infrastructure, similar to that of `rr`-debug [117]. In total, the $\mathcal{E}$FUZZ toolchain is implemented in over ∼13k source lines of `C++` code.

The recording phase records all information that is necessary to faithfully replay the program P during fuzzing. In addition to system calls (the main focus of our discussion), the recording also includes additional information, such as the command-line arguments, environment variables, signals, thread interleavings, and special non-deterministic instructions (e.g., `rdtsc`). System call interception is implemented using a variety of techniques. The common case is handled using *static binary rewriting* to rewrite the `syscall` instruction in `libc`, which diverts control-flow to the framework's interceptor routine. For this, we use the E9PATCH [50] binary rewriting system. In addition, the framework also rewrites the *virtual Dynamic Shared Object* (vDSO) at runtime, and also uses `seccomp` to generate a signal (`SIGSYS`) that is used to intercept system calls outside of `libc` (less common case). Our framework does not use `ptrace`, and thus avoids kernel/user-modes switches during replay for the common case.

Multi-threaded programs are handled by *serializing* system calls during the recording phase, meaning that only one thread will run at a given time. The recording phase runs the program normally using serialized system threads, whereas the replay-with-fuzzing phase uses lightweight *fibers* as a replacement of system

threads. This design avoids one of the main technical limitations of `fork()`, namely, that only the callee system thread will actually be cloned during a fork operation.[2] In contrast, fibers are threads of execution that are implemented purely in user-mode, and where context switching is determined by the recorded schedule (σ). Since there is no user-kernel interaction during replay, fibers can be used as a drop-in replacement of system threads without special handling. Furthermore, since fibers are purely implemented in user-mode, they survive the fork operation intact, which is necessary for the $\mathcal{E}$FUZZ fuzzing algorithm.

$\mathcal{E}$FUZZ is also designed to operate directly on binaries without the need for source code. $\mathcal{E}$FUZZ uses both state and branch coverage as feedback. To collect the branch feedback, binaries can be instrumented using a modified version of E9AFL [61]. State coverage feedback does not require instrumentation. Our implementation can record and fuzz large applications, including the subjects listed in our evaluation.

6.7 Evaluation

To evaluate the effectiveness of $\mathcal{E}$FUZZ, we seek to answer the following research questions:

RQ.1 New bugs. Can $\mathcal{E}$FUZZ find previously unknown bugs in real-world and widely-used programs? Is fuzzing the program environment necessary to reveal these bugs?

RQ.2 Comparisons. How many additional bugs does $\mathcal{E}$FUZZ discover over the baseline? How much more code coverage does $\mathcal{E}$FUZZ achieve compared to the baseline? Are the additional bugs and code coverage improvements related to program environment fuzzing? How many tests can $\mathcal{E}$FUZZ execute per second compared to the baseline?

RQ.3 Ablations. What is the impact of each component on the performance of $\mathcal{E}$FUZZ?

[2] See the `fork` manpage for more information.

Table 6.1: Subject programs used in the evaluation.

	Subject	Version		Subject	Version
Network Protocols	DCMTK	8326435	GUI & UI Applications	Gnome editor (gedit)	v41.0
	DNSmasq	b676923		Gnome Calculator	v42.9
	Exim	5a8fc07		Gnome System Monitor	v42.0
	Kamailio	2e2217b		Glxgears	v23.0.4
	Live555	2c92a57		Midnight Commander (MC)	v4.8.27
	OpenSSH	7cfea58		nano	v6.2
	OpenSSL	a7e9928		Vim	v8.2
	ProFTPD	7892434		Wireshark	v3.6.2
	Pure-FTPd	3296864		Xcalc	v1.8.6
	TinyDTLS	0e865aa		Xpdf	v3.04

6.7.1 Experiment Setup

Subject Programs. $\mathcal{E}$FUZZ is a generic fuzzer capable of testing a broad spectrum of user-mode programs in Linux. Given the scope of applications that $\mathcal{E}$FUZZ can fuzz, we shall focus on two core categories of program: network protocols and *(Graphical) User Interface* GUI/UI applications that interact with a human user via the windowing system or terminal. These two categories have been recognized as challenging for fuzzing [20]. For example, fuzzing GUI applications with AFL++ [57] is "*not possible without modifying the source code*".[3] Since $\mathcal{E}$FUZZ works at the abstraction of the kernel/user-mode boundary, it can fuzz GUI applications and other difficult subjects without special handling. By targeting challenging fuzzing targets, we aim to demonstrate the generality of $\mathcal{E}$FUZZ.

In total, we collect 20 subjects as detailed in Table 6.1. For network protocols, we collect subjects from PROFUZZBENCH [115], a widely-used benchmarking platform for evaluating the network-enabled fuzzers. However, for GUI applications under Linux, there is no existing fuzzing dataset. We therefore select subjects from frequently-used and well-known applications and frameworks, including text editors (UI), visual shells (UI), GNOME desktop environment (GUI), Qt (GUI), and the underlying windowing system (GUI).

Comparisons. To the best of our knowledge, no existing fuzzers target the full program environment. In the realm of fuzzing network protocols, AFLNET is the first network fuzzer, and also recommended by AFL++ for fuzzing network

[3] https://aflplus.plus/docs/best_practices/#fuzzing-a-gui-program (as of writing).

services. NYX-NET enhances the fuzzing throughput of AFLNET by introducing innovative hypervisor-based snapshots. Unfortunately, we cannot compare with AFL++ since it does not work on many of the network protocols. As shown in the paper of NYX-NET [141], AFL++ only works on 5 of the PROFUZZBENCH subjects. Furthermore, for these 5 subjects, AFL++ performs significantly worse than both AFLNET and NYX-NET. Therefore, for network protocol subjects, we use AFLNET and NYX-NET as baselines for comparison. AFL++ and AFL++-based fuzzers are also not able to fuzz GUI applications in Linux with user interactions [57]. Recent work [82] uses *test harness* generation to enable GUI fuzzing, but only for Windows applications. As such, there is no available fuzzer to compare against GUI applications under Linux.

Performance Metrics. We evaluate the performance of $\mathcal{E}$FUZZ based on three primary metrics: *bug-finding capability*, *code coverage*, and *fuzzing throughput*. As recommended by the fuzzing community [89, 23], the ultimate metric of a fuzzer is the number of distinct bugs found. Since a fuzzer cannot find bugs in uncovered code, code coverage is important too, and thus serves as a secondary metric. While fuzzing throughput is not a mandatory evaluation metric, it may affect the efficacy of a fuzzer. We also report throughput to demonstrate the robustness of the fuzzer.

Experimental Infrastructure. All experiments were conducted on an Intel® Xeon® Platinum 8468V CPU with 192 logical cores clocked at 2.70GHz, 512GB of memory, and running Ubuntu 22.04.3 LTS. Each experiment runs for 24 hours. We report the average over 10 runs to mitigate the impact of randomness.

6.7.2 Discovering New Bugs (RQ.1)

Method. We ran $\mathcal{E}$FUZZ on the subjects listed in Table 6.1 to discover bugs. We utilized the same bug oracles as traditional fuzzers (e.g., AFLNET and NYX-NET), including crashes, hangs, assertion failures, and sanitizer violations. For initiating the fuzz campaign, we used initial seeds provided by the programs if available; otherwise, we provided standard user inputs as initial seeds. In the case of network protocols, we utilized their clients to send request messages. For GUI applications, we simulated typical user interactions; as an example, with a calculator, the application is opened to perform a simple addition calculation before it is closed. All inputs represent

Table 6.2: Statistics of bugs discovered by $\mathcal{E}$FUZZ; a total of 33 previously unknown bugs found, 24 bugs confirmed by developers, 16 bugs assigned CVE IDs, and 16 bugs fixed. (Note that, each color represents a distinct category of applications)

ID	Subject	Environment	Bug Type	Bug Status
1	Dcmtk	Cached data	Buffer overflow	CVE-requested, fixed
2	Exim	Configuration	Buffer overflow	Reported
3	Exim	Special file	Null pointer dereference	Reported
4	Kamailio	Socket	Null pointer dereference	Reported
5	Live555	Socket	Heap use after free	CVE-granted, fixed
6	Live555	Media resource	Buffer overflow	Reported
7	OpenSSH	Configuration	Null pointer dereference	CVE-requested, fixed
8	OpenSSH	Configuration	Null pointer dereference	Reported
9	Pure-FTPd	Time resource	Null pointer dereference	Reported
10	gedit	Configuration	Null pointer dereference	CVE-granted
11	gedit	Socket	Null pointer dereference	CVE-granted
12	Calculator	Socket	Buffer overflow	CVE-granted, fixed
13	Calculator	Socket	Null pointer dereference	CVE-granted
14	Calculator	Socket	Null pointer dereference	CVE-requested, fixed
15	Monitor	Cached data	Bad free	CVE-granted
16	Monitor	Theme resource	Null pointer dereference	CVE-requested, fixed
17	Glxgears	Socket	Buffer overflow	CVE-granted
18	Glxgears	Socket	Buffer overflow	CVE-granted
19	MC	Configuration	Null pointer dereference	CVE-granted
20	MC	Configuration	Arithmetic exception	CVE-granted
21	MC	Socket	Null pointer dereference	CVE-granted
22	nano	Configuration	Null pointer dereference	Reported
23	nano	Cached data	Null pointer dereference	CVE-granted, fixed
24	Vim	Configuration	Null pointer dereference	CVE-granted
25	Vim	Cached data	Null pointer dereference	CVE-granted, fixed
26	Wireshark	Socket	Null pointer dereference	CVE-granted, fixed
27	Xcalc	Socket	Null pointer dereference	CVE-requested, fixed
28	Xcalc	Socket	Out-of-bounds write	CVE-granted, fixed
29	Xcalc	Cached data	Out-of-bounds read	CVE-requested, fixed
30	Xpdf	Configuration	Null pointer dereference	Reported
31	Xpdf	Configuration	Null pointer dereference	Reported
32	Xpdf	Socket	Bad free	CVE-requested, fixed
33	Xpdf	Socket	Out-of-bounds read	CVE-requested, fixed

normal usage scenarios encountered in the real world. We subjected each program to a *24-hour run* (typical recommended length of a fuzz campaign [89]) to identify bugs. Upon finding bugs, we reported them to the developers for confirmation. In the case of bugs with potential security implications, we requested CVE IDs from the CVE Numbering Authority. All activities were conducted in a one-month period, including bug finding, debugging, reporting to developers, and requesting CVEs.

Results. Table 6.2 shows the distinct and previously unknown bugs found by $\mathcal{E}$FUZZ. In the *Bug Description* column, we elucidate the root causes responsible for these

bugs, and illustrate the immediate environmental factors in the *Environment* column. It is important to note that triggering a bug often requires hundreds of diverse environmental inputs. Therefore, we only listed the most relevant environmental input that exposed the bugs after mutation. Furthermore, we provide details about the bug types and their current status in the last two columns.

In total, we discovered 33 previously unknown bugs, out of which 24 have received confirmation from their respective developers. Developers had fixed 16 of these bugs by the time of paper submission. 16/24 bugs have been assigned CVE IDs. These bugs span various categories, including buffer overflows, use-after-frees, null pointer dereferences, and arithmetic exceptions. Furthermore, these bugs are triggered by fuzzing a diverse range of environmental inputs, including sockets, configuration files, multiple types of resource files, cached data, etc. Therefore, a fuzzer that exclusively concentrates on a singular input cannot expose all of these bugs.

These results highlight the significant bug-finding capability of $\mathcal{E}$FUZZ. Moreover, they demonstrate the importance of program environment fuzzing, and $\mathcal{E}$FUZZ has shown its effectiveness in this regard. Two case studies below illustrate bugs discovered by $\mathcal{E}$FUZZ.

Case Study: GNOME Desktop Environment. GNOME client applications (e.g., `gnome-calculator`, etc.) interact with the windowing system and several other services (Figure 6.1). $\mathcal{E}$FUZZ is able to expose several bugs in multiple different input sources, including several bugs related to the windowing system and client libraries, bugs in the DBus socket connection to the session manager, as well as bugs in non-socket inputs (`loaders.cache`, `gtk.css`, etc.). As an example, we can consider Bug #12, which affects the `XESetWireToEvent()` function from `libX11`. This function fails to check whether the event values are within the bounds of the arrays that the functions write to. Instead, the function directly uses the value as an array index, leading to an intra-object overwrite and probable crash. This bug stems from the implicit trust that `libX11` functions place in the values supplied by an X server, following X11 protocol. However, the environment cannot be fully trusted, as a malicious server or proxy can impact applications. This bug was assigned CVE ID by X11 developers and received a CVSS score HIGH 7.5. We note that other subjects, including many GNOME applications, are also affected by this bug.

Case Study: Bug #23 in GNU nano. GNU nano is a text editor for Unix-

like operating systems and is part of the GNU Project. This bug appears in `read_the_list()` of `browser.c`. This function initiates an initial iteration over a directory using `readdir()` to obtain the current entries, followed by a rewinding action using `rewinddir()` to cache these entries. Subsequently, a second iteration employing `readdir()` is performed to directly access these cached entries. Unfortunately, before this second iteration, there is no boundary-checking mechanism. As a result, any environmental changes, such as directory deletions, can easily trigger a crash during the second iteration. This is precisely how $\mathcal{E}$FUZZ exposes it. This bug existed from the first version of GNU nano in 2005 and had been hidden for 18 years!

$\mathcal{E}$FUZZ discovered 33 previously unknown bugs in widely used network protocols and GUI applications, with 24 confirmed and 16 fixed by their developers. 16 of them were assigned CVEs.

6.7.3 Comparisons with Baselines (RQ.2)

Method. For network protocols, we compare $\mathcal{E}$FUZZ against two baselines AFLNET and NYX-NET under three aspects: the number of bugs found, code coverage, and fuzzing throughput. We omit GUI programs due to the lack of a suitable baseline. We configure all fuzzers employing the same initial seeds obtained from PROFUZZBENCH. Our evaluation of code coverage focuses on measuring branch coverage achieved on binaries. We utilize the original scripts provided by PROFUZZBENCH, to collect code coverage data and present their trends over time. We report the total number of bugs found, the average coverage, and the average fuzzing throughput achieved by each fuzzer across 10 runs of 24 hours.

Comparing Results on Bug Finding. Table 6.3 shows the total number of unique bugs found by each fuzzer. In all subjects, $\mathcal{E}$FUZZ discovered a total of 9 unique bugs, as detailed in Table 6.2. However, both AFLNET and NYX-NET could only find 2 (i.e., Bug #4 and Bug #5 in Table 6.2); in addition, neither fuzzer found any additional bug. The remaining 7 bugs were exposed by fuzzing non-socket environment inputs, such as cached data and resources. Since these environment inputs are not fuzzing targets for AFLNET and NYX-NET, they were

Table 6.3: Number of unique bugs found by AFLNet, Nyx-Net and $\mathcal{E}$fuzz on subjects of network protocols.

Fuzzer	AFLNet	Nyx-Net	$\mathcal{E}$fuzz
#Bug	2	2	9

Table 6.4: Average branch coverage across 10 runs of 24 hours achieved by $\mathcal{E}$fuzz compared to AFLNet and Nyx-Net.

Subject	$\mathcal{E}$**fuzz**	**Compare with AFLNet**			**Compare with Nyx-Net**		
		Coverage	**Improv**	$\hat{\boldsymbol{A}}_{\mathbf{12}}$	**Coverage**	**Improv**	$\hat{\boldsymbol{A}}_{\mathbf{12}}$
DCMTK	15181.7	7564.9	+100.69%	1.00	9362.0	+62.16%	1.00
DNSmasq	8090.9	4066.7	+98.95%	1.00	4009.0	+101.82%	1.00
Exim	5642.7	4594.4	+22.82%	1.00	4935.2	+14.34%	1.00
Kamailio	23425.6	13466.1	+73.96%	1.00	17960.0	+30.43%	1.00
Live555	14319.0	10379.5	+37.95%	1.00	11436.0	+25.21%	1.00
OpenSSH	8584.5	7920.0	+8.39%	1.00	7631.5	+12.49%	1.00
OpenSSL	26225.9	19820.4	+32.32%	1.00	25330.1	+3.54%	1.00
ProFTPD	19478.0	17654.0	+10.33%	1.00	16504.0	+18.02%	1.00
Pure-FTPd	7182.8	5309.0	+35.29%	1.00	6766.5	+6.15%	1.00
TinyDTLS	2747.5	1901.5	+44.49%	1.00	2052.5	+33.86%	1.00
Average	—	—	+46.52%	—	—	+30.80%	—

unable to expose them. Furthermore, regarding bugs induced by network sockets, $\mathcal{E}$fuzz successfully exposed the same number as AFLNet and Nyx-Net. This demonstrates that $\mathcal{E}$fuzz maintains the effectiveness in fuzzing a single environment source, although it fuzzes all environment sources.

In the aspect of bug finding, $\mathcal{E}$fuzz discovered 9 previously unknown bugs, while AFLNet and Nyx-Net only discovered 2 without any additional bug found.

Comparing Results on Code Coverage. Figure 6.5 illustrates trends in average code coverage over time for AFLNet, Nyx-Net and $\mathcal{E}$fuzz. Across all subjects, $\mathcal{E}$fuzz significantly outperformed both AFLNet and Nyx-Net. Initially, at the start of each experiment, all three fuzzers covered a similar number of code branches. However, over time, $\mathcal{E}$fuzz substantially covered more code than AFLNet and Nyx-Net. Even after 24 hours, $\mathcal{E}$fuzz still had the potential to discover new code, whereas, in most cases, the code coverage for AFLNet and Nyx-Net tended to plateau quickly.

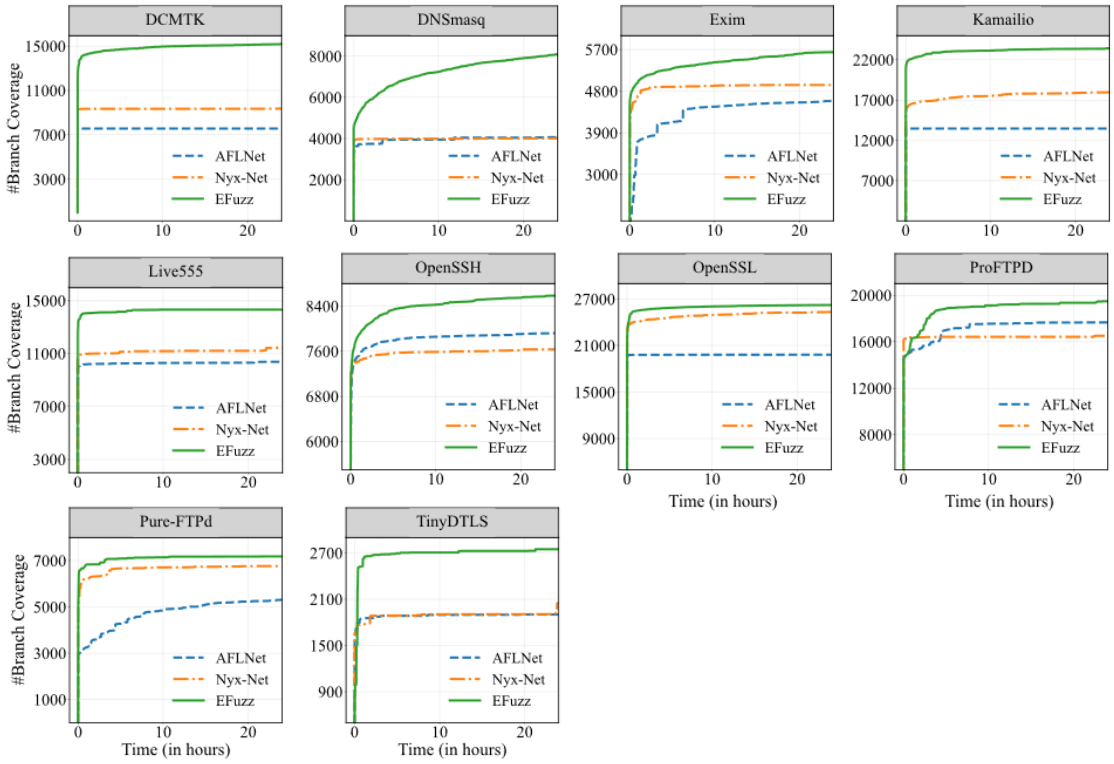


Figure 6.5: Code covered over time by AFLNet, Nyx-Net and $\mathcal{E}$fuzz across 10 runs of 24 hours on ProFuzzBench subjects.

Table 6.4 shows the final code coverage of $\mathcal{E}$fuzz and two baselines. To quantify the improvement of $\mathcal{E}$fuzz over baselines, we report the number of branches covered by $\mathcal{E}$fuzz, AFLNet and Nyx-Net (*Coverage*), respectively, the percentage improvement of $\mathcal{E}$fuzz (*Improv*), and the probability that a random campaign of $\mathcal{E}$fuzz outperforms a random campaign of baselines ($\hat{A}_{12}$). For all subjects, $\mathcal{E}$fuzz covers more code than both baselines. Specifically, $\mathcal{E}$fuzz averagely covers 46.52% more code than AFLNet with a range from 8.39% to 100.69%. When compared to Nyx-Net, $\mathcal{E}$fuzz covers 30.80% more code on average from 3.54% to 101.82%. The Vargha-Delaney [116] effect size $\hat{A}_{12} \geq 0.70$ demonstrates a substantial improvement of $\mathcal{E}$fuzz over both baselines in terms of code coverage.

To investigate the correlation between improved code coverage and program environment fuzzing, we conducted a comprehensive analysis of the additional code covered by $\mathcal{E}$fuzz, focusing on the subject DCMTK. DCMTK is a widely-used implementation of the DICOM (Digital Imaging and Communication in Medicine)

Table 6.5: Fuzzing throughput (execs/s) in 10 runs of 24 hours achieved by $\mathcal{E}$FUZZ compared to AFLNET and NYX-NET.

Subject	**$\mathcal{E}$fuzz**	**Compare with AFLNet**		**Compare with Nyx-Net**	
		AFLNet	**Speedup**	**Nyx-Net**	**Speedup**
DCMTK	101.7	22.3	4.57×	815.4	0.12×
DNSmasq	393.0	22.6	17.40×	1126.8	0.35×
Exim	713.4	5.1	139.06×	514.5	1.39×
Kamailio	121.9	5.2	23.62×	234.9	0.52×
Live555	237.4	16.8	14.13×	133.1	1.78×
OpenSSH	1320.9	38.6	34.19×	1031.1	1.28×
OpenSSL	124.5	32.7	3.80×	227.4	0.55×
ProFTPD	293.4	7.2	40.91×	333.5	0.88×
Pure-FTPd	528.9	9.8	54.08×	596.0	0.89×
TinyDTLS	640.9	3.1	206.74×	1354.0	0.47×
Average	—	—	53.85×	—	0.82×

protocol. While fuzzing DCMTK using $\mathcal{E}$FUZZ, we observed multiple environment sources that undergo mutation. These included the configuration file, the database responsible for storing patient records, various patient cases, and network sockets utilized for hospital communication. Among the 5819 additionally covered branches, 69% of them demonstrated direct connections to environmental mutations, such as parsing and changing the configuration settings and adding entries to the database. Therefore, full environment fuzzing significantly contributes to increased code coverage.

$\mathcal{E}$FUZZ covers 46.52% and 30.80% more code than AFLNET and NYX-NET, respectively, with most additional code coverage resulting from program environment fuzzing.

Comparing Results on Fuzzing Throughput. The experimental results on fuzzing throughput are shown in Table 6.5. For each fuzzer, the corresponding fuzzing throughput is shown in the respective columns. In the *Speedup* columns, we present how much faster $\mathcal{E}$FUZZ executes compared to AFLNET and NYX-NET, respectively. $\mathcal{E}$FUZZ achieves a fuzzing throughput ranging from 101.7 to 1320.9 executions per second. The fuzzing throughput of AFLNET is from 3.1 to 38.6 executions per second, and $\mathcal{E}$FUZZ executes 53.85 × faster than AFLNET on average.

When compared to NYX-NET, $\mathcal{E}$FUZZ executes faster than NYX-NET on some subjects (e.g., 1.78× faster on Live555) but slower on others. These results are expected as AFLNET always replays each input sequence from the root, while both $\mathcal{E}$FUZZ and NYX-NET avoid replaying repetitive input sequences by faithful replay and state snapshots, respectively. In addition, compared to NYX-NET, $\mathcal{E}$FUZZ introduces some time overhead on certain subjects (e.g., DCMTK) to explore more behaviors. However, this overhead is justified by the evident improvement in bug finding and code coverage. Overall, $\mathcal{E}$FUZZ still maintains a robust throughput.

$\mathcal{E}$FUZZ maintains a robust fuzzing throughput while enhancing the capability of bug finding and code coverage.

6.7.4 Ablation Studies (RQ.3)

Impact of Algorithm Components. $\mathcal{E}$FUZZ employs two strategies to enhance the search efficiency of the program environment: behavior divergence handling based on the relaxed replay, and feedback guidance. To evaluate the impact of each strategy on the improvement of the code coverage, we conducted an ablation study. For this purpose, we developed two ablation tools:

- EF1: based on $\mathcal{E}$FUZZ, without behavior divergence handling,
- EF2: based on $\mathcal{E}$FUZZ, without fuzzing feedback.

We compare the average code coverage achieved by $\mathcal{E}$FUZZ with that of EF1 and EF2 across 10 runs of 24 hours in each subject, and report the percentage improvements.

Table 6.6 shows the results of the percentage improvements in terms of average code coverage. Overall, across all subjects, both strategies contributed to the increase in code coverage, with none exhibiting a negative impact. Compared to EF1 without behavior divergence handling, $\mathcal{E}$FUZZ resulted in an average increase of 30.59% in code coverage. Notably, in DCMTK, TinyDTLS and MC, $\mathcal{E}$FUZZ exhibited code coverage improvements exceeding 60%. Compared to EF2 without fuzzing feedback, $\mathcal{E}$FUZZ increased the code coverage by 2.39% to 27.79%, with an average increase of 10.38%. Furthermore, comparing $\mathcal{E}$FUZZ with both tools across all subjects, $\hat{A}_{12}$=1, which indicates that $\mathcal{E}$FUZZ significantly outperforms EF1 and EF2. These results

Table 6.6: Improvement of code coverage achieved by $\mathcal{E}$FUZZ in comparison to ablation tools EF1 and EF2. The results show that the impact of behavior divergence handling and fuzzing feedback is significant.

Subject	***vs.* EF1**	***vs.* EF2**	**Subject**	***vs.* EF1**	***vs.* EF2**
DCMTK	+60.83%	+22.52%	gedit	+22.14%	+8.17%
DNSmasq	+39.28%	+27.79%	Calculator	+27.12%	+6.61%
Exim	+12.24%	+9.66%	Monitor	+14.24%	+4.44%
Kamailio	+28.89%	+10.52%	Glxgears	+12.01%	+2.39%
Live555	+30.26%	+14.61%	MC	+68.60%	+13.46%
OpenSSH	+10.92%	+3.99%	nano	+20.48%	+8.75%
OpenSSL	+12.98%	+8.06%	Vim	+12.50%	+20.47%
ProFTPD	+26.81%	+9.21%	Wireshark	+17.90%	+8.17%
Pure-FTPd	+46.21%	+6.75%	Xcalc	+27.66%	+5.55%
TinyDTLS	+98.57%	+8.59%	Xpdf	+22.19%	+7.79%
Average				+30.59%	+10.38%

demonstrate the importance of $\mathcal{E}$FUZZ's divergence handling and the effectiveness of fuzzing feedback in guiding the search.

We further measured the fuzzing throughput of $\mathcal{E}$FUZZ, EF1, and EF2 across each subject. On average, $\mathcal{E}$FUZZ achieves a fuzzing throughput of 447.6 executions per second, while EF2 achieves a similar throughput of 454.9 executions per second. However, EF1 executes faster than $\mathcal{E}$FUZZ with a fuzzing throughput of 698.3 executions per second. This higher throughput is due to $\mathcal{E}$FUZZ's strategy of handling behavior divergence, which explores longer traces, trading raw throughput for better coverage.

Divergence handling and feedback guidance enable $\mathcal{E}$FUZZ to increase code coverage by 30.59% and 10.38%, respectively. The contribution of each strategy to enhancing code coverage is significant.

Analysis of Relaxed Replay. To further analyze the impact of relaxed replay for divergence handling, we examined the following additional questions:

- How often do executions resort to relaxed replay (#Freq.)?
- How many system calls in tree branches use relaxed replay (#RelaxSysCs *vs* #TotalSysCs)?

Table 6.7: Statistical analysis of relaxed replay proposed by $\mathcal{E}$FUZZ, including the frequency of the executions resorting to relaxed replay (#Freq.), the total number of system calls executed in each tree branch (#TotalSysCs), the number of system calls resorting to relaxed replay in each tree branch (#RelaxSysCs), and the point at which a tree branch starts to resort to relaxed replay (#StartPoint).

Subject	**#Freq.**	**#TotalSysCs**	**#RelaxSysCs**	**#StartPoint**
DCMTK	93.67%	130.0	116.7	10.27%
DNSmasq	34.52%	99.1	11.1	88.82%
Exim	98.78%	90.8	23.4	74.18%
Kamailio	73.85%	141.2	90.5	35.93%
Live555	45.12%	375.1	130.2	65.30%
OpenSSH	87.66%	105.3	10.7	89.87%
OpenSSL	91.02%	51.3	11.0	78.57%
ProFTPD	95.01%	172.1	24.8	85.61%
Pure-FTPd	60.08%	114.9	14.5	87.41%
TinyDTLS	78.58%	312.6	295.5	5.46%
gedit	99.89%	397.3	332.1	16.42%
Calculator	99.65%	152.1	80.7	46.98%
Monitor	94.68%	114.5	69.6	39.21%
Glxgears	94.31%	92.6	35.8	61.37%
MC	89.98%	278.1	127.6	54.11%
nano	98.99%	164.6	89.0	45.93%
Vim	93.69%	386.2	307.7	20.32%
Wireshark	81.43%	198.4	157.5	20.60%
Xcalc	89.37%	147.5	51.7	50.78%
Xpdf	89.30%	234.1	182.0	22.26%
Average	84.48%	187.9	108.1	49.97%

- How early after a branch does relaxed replay start (#StartPoint)?

For this purpose, we collect the statistical data from 20 subjects over 24-hour runs and report them in Table 6.7. On average, 84.48% of all executions across the 20 subjects have to resort to relaxed replay. The total number of system calls executed in each tree branch (calculated from forking points) is 187.9, with 108.1 of those system calls using relaxed replay. In addition, the starting points of the relaxed replay vary among different subjects, ranging from 5.46% of the tree branch on TinyDTLS to 89.87% on OpenSSH. On average, the relaxed replay starts to resort to relaxed replay around halfway (49.97%). These results demonstrate that the relaxed replay for divergence handling is necessary and commonly used by $\mathcal{E}$FUZZ.

6.7.5 Discussion

Manual Effort. The manual effort needed for using $\mathcal{E}$FUZZ is minimal. The only manual involvement is the user inputs necessary for testing GUI applications in the recording phase. For example, when testing the calculator, the user needs to open the application, execute a simple addition operation, and then close it. After recording, the rest of the fuzzing workflow is fully automatic. To collect code coverage feedback, $\mathcal{E}$FUZZ can directly instrument the binaries of the program under test, eliminating the need to recompile from source code. Similarly, the system-call interception infrastructure (for record and replay) is designed to work with binary code.

Limitations. In this work, we leverage greybox fuzzing over complex program environments. We have demonstrated that the approach of $\mathcal{E}$FUZZ is effective in exposing previously unknown bugs and enhancing code coverage. Like other fuzzers, the efficacy of $\mathcal{E}$FUZZ depends on the quality of the initial seed(s), and $\mathcal{E}$FUZZ is not guaranteed to cover the entire search space. Limited seed recordings (e.g., open and immediately close a GUI) generally result in a limited exploration compared to diverse recordings (e.g., exercising different GUI elements). However, the dependence on quality seeds is an inherent limitation of fuzzing in general, in contrast to model checking and other verification techniques that attempt to systematically explore the entire search space. While $\mathcal{E}$FUZZ can fuzz a broad range of programs, its scope is limited to Linux user-mode environments. This limitation stems from our underlying environmental record and replay infrastructure. Despite this, and compared to existing fuzzers, $\mathcal{E}$FUZZ still maintains its generality, and can fuzz even challenging subjects such as network protocols and GUI applications.

Relaxed replay assumes that I/O system calls can be mutated and reordered arbitrarily. This is a straightforward generalization of what existing fuzzers already assume. For example, AFL implicitly assumes the input file can be mutated arbitrarily, while AFLNET assumes messages can be reordered. However, these assumptions may not always hold for some edge cases. Special files (e.g., `/proc/*` and `/dev/zero`) and self-pipes are not mutable. Fortunately, such examples are rare and can be avoided using a predefined special-case list. As such, no false positives were detected during our evaluation. By design, $\mathcal{E}$FUZZ does not use modeling,

allowing it to fuzz programs without any manual effort or prior knowledge. In addition, $\mathcal{E}$FUZZ supports fuzzing "new" syscalls not present in the original recording. During relaxed replay, $\mathcal{E}$FUZZ provides inputs to the syscall based on file descriptors, regardless of the specific syscall number. However, due to behavior divergence, if the program invokes a system call to access inputs from a file descriptor that is not originally recorded, $\mathcal{E}$FUZZ will fail the system call to maintain the plausibility of the replay.

6.8 Conclusion

In this chapter, we present a methodology, tool, and evaluation to handle complex program environments. Our $\mathcal{E}$FUZZ tool avoids environment modeling by recording program executions and selectively mutating (in the style of greybox fuzzing) the recorded executions during replay to capture the effect of different environments. Evaluation of $\mathcal{E}$FUZZ found 33 previously unknown bugs, out of which 24 were confirmed by developers. The applications tested include well-known GUI applications and protocol implementations. $\mathcal{E}$FUZZ presents a general approach for handling software environments, which is different from (a) the practitioners' approach of procuring sample environments for testing code on them one by one, or (b) the current established research on environment modeling. We do not model environments, and we do not procure environments. Instead $\mathcal{E}$FUZZ is an automated framework for implicitly navigating the space of program environments via mutational fuzzing.

Chapter 7

Related Work

In this chapter, we discuss how our techniques proposed in this thesis advance related work, including model checking, runtime verification, and fuzzing.

7.1 Model Checking

7.1.1 Model Checkers

Model checking is a well-known property verification technique dating back to the 1980s [40, 131]; it is used to prove a temporal logic property in a finite-state system, or to find property violation bugs. The early works check a temporal logic property against a finite-state transition system. There exist well-known model checkers such as [72, 35, 83], which can be used to check temporal properties on a constructed model (via state-space exploration). To construct models, one method is manual construction through a modeling language. This requires substantial effort and can be error-prone [111, 65]. LTL-Fuzzer directly checks software implementations and does not separately extract models from the software, thus reducing the modeling effort.

Early works on model checking have been extended to automatically find bugs in software systems, which are typically infinite-state systems. Model checking of software systems usually involves either some extraction of finite-state models or directly analyzing the infinite-state software system via techniques such as symbolic analysis. Automatic model extraction approaches [41, 11, 68, 137] include the works on predicate abstraction and abstraction refinement [11, 12], which build up a hierarchy of finite-state abstract models for a software system for proving a property. These approaches extract models that are conservative approximations and capture

a superset of the program behavior. There are a number of stateful software model checkers, such as CMC [111], Java Pathfinder [154], MaceMC [84], CBMC [38], CPAchecker [18], which find assertion violations in software implementations. Many of these checkers do not check arbitrary LTL properties for software implementations. These model checkers either suffer from state space explosion or suffer from other kinds of explosion, such as the explosion in the size/solving-time for the logical formula in bounded model checking. In contrast, LTL-Fuzzer does not save any states for safety property checking and saves only certain property-relevant program states in liveness property checking. At the same time, LTL-Fuzzer does not give verification guarantees and does not perform a complete exploration of the state space. We now proceed to discuss incomplete validation approaches.

7.1.2 Incomplete Checkers

Instead of exploring the complete set of behaviors, or a superset of behaviors, one can also explore a subset of behaviors. Incomplete model learning approaches [148] can be mentioned in this regard. The active model learning technique, such as LearnLib [77], is widely used to learn models of real-world protocol implementations [58, 43, 60, 59]. It does not need user involvement. However, it is time-consuming and hard to determine whether the learned model represents the complete behavior of the software system [148, 163]. Compared with active learning, LTL-Fuzzer can more rapidly check properties, as shown in our experimental comparison with LearnLib+NuSMV. To alleviate the state-explosion problem, stateless checkers such as VeriSoft [62] and Chess [112] have been proposed; these checkers do not store program states. These works typically involve specific search strategies to check *specific* classes of properties such as deadlocks, assertions, and so on. In contrast, LTL-Fuzzer represents a general approach to find violations of *any* LTL property.

7.1.3 Environment Capturing

Capturing different execution environments poses a fundamental challenge for any testing, analysis, and verification techniques. For example, model checking and symbolic execution require a comprehensive view of the execution environment to reason accurately about program behavior. Even traditional testing may miss

critical behaviors when the search is conducted under a single, fixed environment. Many existing approaches tackle this by manually abstracting the environment via a model [10, 27, 63, 142, 111]. However, crafting abstract models is often laborious and error-prone. To reduce this effort, some alternatives [33, 128] leverage virtualization to execute programs in realistic environments without requiring manual abstraction. Nevertheless, the path-explosion problem persists when analyzing an entire software stack [9, 33]; the presence of many program environments further exacerbates the path-explosion problem while finding bugs in software. $\mathcal{E}$FUZZ introduces a new direction in environment capturing by using greybox fuzzing to implicitly generate the effects of diverse environments. Compared to existing approaches, our method offers greater usability and scalability.

7.1.4 Model Checking of Distributed Systems

Of the existing model-checking frameworks for distributed systems, MoDist [162] is the most similar to our approach Mallory, as it requires no modification in the SUT code; instead, it manipulates the system's execution by intercepting calls to the Windows API. The main conceptual distinction of MoDist from our work is its interposition layer: unlike our observers, which are passive, MoDist intercepts network, timing, and disk-related system calls and *pauses* the SUT. This provides more control—in our terminology, it allows the mediator to act as a *scheduler*—but comes at the expense of requiring a complex interposition framework that replicates and replaces most of the API of a specific OS. The design innovations of Mallory that enable summarising observations about SUT (Section 5.3.2) are orthogonal to the use of an interposition layer, and, therefore, such an interposition layer could be integrated within our architecture—rather than choosing only which faults to introduce, our nemesis would choose every action. In this work, we focused on finding bugs in non-Byzantine fault-tolerant distributed systems [91] thus, side-stepping the challenge of modelling the behaviour of possibly malicious nodes. We believe that Mallory's workflow can be combined with existing techniques for Byzantine system testing that emulate attacks by running several copies of the same node, but, for now, only allow for execution in a network emulator [14].

7.2 Runtime Verification

Runtime verification is a lightweight and yet rigorous verification technique [96, 16]. It analyzes a single execution trace of a system against formally specified properties (e.g., LTL properties). It originates from model checking and applies model checking directly to the real implementations. Model checking checks a model of a target system to verify the correctness of the system, while runtime verification directly checks the implementation, which could avoid different behaviors between models and implementations. LTL-Fuzzer shares the same benefit as runtime verification. Besides, runtime verification deals with finite executions, as one single execution has necessarily to be finite. This avoids the state explosion problem that model checking suffers from. Meanwhile, it leads to that the runtime verification approaches [29, 80, 136, 44] often only check safety properties. LTL-Fuzzer, however, is able to check liveness properties by leveraging the strategy of saving program states.

Conceptually, our method is very different from runtime verification. Runtime verification focuses on checking (a temporal logic property) on a single execution. Our method is focused on using temporal logic properties to guide the construction of an execution that violates the property. Thus, our work is more of a test generation method. Since runtime verification methods need tests whose execution will be checked, our method can be complementary to runtime verification. In other words, our method can generate tests likely to violate a given temporal property, and these tests can be further validated by run-time verification.

7.3 Greybox Fuzzing

7.3.1 Finding Violations of Complex Properties

There are three broad variants of fuzzing: blackbox fuzzing [110], whitebox fuzzing or symbolic execution [27, 138, 73], and greybox fuzzing [167, 22, 135, 21, 31, 155]. We first discuss greybox fuzzing since they are the most widely used in the industry today. In contrast to software model checking, blackbox/greybox fuzzing techniques represent a random/biased-random search over the domain of inputs

for finding bugs or vulnerabilities in programs. Most greybox fuzzing techniques are used to detect memory issues (e.g., buffer overflow and use after free) that can produce observable behaviors (e.g., crashes). However, LTL-Fuzzer can not only witness simple properties like memory corruption but also detect LTL property violations, for *any* given LTL property, however complex. Recent advances in greybox fuzzing use innovative objective functions for achieving different goals, such as [21], directing the search to specific program locations. The capabilities of LTL-Fuzzer go beyond visiting specific locations, and LTL-Fuzzer is used to witness specific event ordering constraints embodied by the negation of an arbitrary LTL property. PGFUZZ [86] is a greybox-fuzzing framework to find safety violations for robotic vehicles, but it is customized to be used on implementations of robotic vehicles. LTL-Fuzzer can be used to find violations of *any* LTL property for software from any application domain.

7.3.2 Grammar-Awared Fuzzing

Generation-based fuzzing generates messages from scratch based on manually constructed specifications [97, 52, 3, 81, 1, 13]. 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.

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 [71, 123] and whitebox fuzzers [26, 42, 99]. Blackbox fuzzers such as TreeFuzz [123] 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 [26], 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.

7.3.3 Stateful Fuzzing

Recently, greybox fuzzing has been extended to stateful reactive systems, such as network protocol implementations. Mutation-based fuzzers [167, 21, 127, 98, 57, 140, 5] 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 reactive systems, branch coverage alone is a useful but insufficient metric for guiding the fuzzing campaign, as elucidated in existing works [7]. 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 [8, 126, 114, 130] proposes various state representation schemes. AFLNet [126] utilizes the response code as states, constructs a state machine during the fuzzing campaign, and employs it as state-coverage guidance. StateAFL [114], SGFuzz [8], and NSFuzz [130] propose distinct state representation schemes based on program variables. In ChatAFL, 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. While fuzzing distributed systems, Mallory does not make any such assumptions about state variables. It captures the events executed so far in a reactive system via a timeline abstraction. Mallory also suggests for the first time, the fuzzer of stateful reactive systems, itself as a reactive system. Thus, the fuzzer feedback, instead of being given for an entire schedule or execution, is given incrementally action by action, with the probabilities for the actions being adjusted via Q-learning.

7.3.4 Snapshot Fuzzing

When fuzzing stateful reactive systems, achieving a deep exploration of program states often requires a lengthy sequence of messages. For instance, AFLNet [126] opts for replaying each message sequence from initial states, somewhat impeding

its fuzzing speed. To address this limitation, SnapFuzz [5] employs an in-memory filesystem to efficiently reset to specific interesting states, overcoming the impediment faced by AFLNet. In a similar vein, Nyx-Net [141] introduces a hypervisor-based technique to dump program states at points of interest, including all memory contents. Our $\mathcal{E}$fuzz algorithm eliminates the need for snapshots or hypervisors, and dynamically *reconstructs* states on demand through replay. Our algorithm has similarities with `fork`-based fuzzers such as AFL [167] and AFL++ [57]. Rather than employing a global fork server at program entry, $\mathcal{E}$fuzz implements a mini-fork server at each program input, avoiding replaying system call sequence prefixes.

7.3.5 Concurreny Program Fuzzing

The vast majority of existing greybox fuzzers aim at testing sequential software systems, with most of the recent research efforts dedicated to generating more diverse inputs [144, 51], defining better feedback functions [8, 134] and test oracles [105]. With this mindset, fuzzing distributed systems poses unique challenges since (*i*) the inputs include not just plain data but also schedules consisting of environmental faults and (*ii*) code coverage is not as efficient, since distributed systems typically do not have complex control flow and their behavioral complexity come from the asynchrony of operations across multiple nodes. The recent Muzz [30] framework for fuzzing of (single-node) multi-threaded programs, addresses (*i*)-(*ii*) by extending the edge coverage metric with possible thread interleavings, while also identifying equivalent schedules. Muzz' approach does not extend to distributed systems as it is tailored to tracking the ordering of specific threading functions, while Mallory works with arbitrary events and communication patterns. Furthermore, Muzz relies on instrumenting the system scheduler, which is difficult to implement for a distributed system without virtualizing the networking layer. In Mallory, our timeline-based approach offers a more general way to observe program behaviours for *any* non-sequential program, including stand-alone multithreaded as well as distributed systems.

7.3.6 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 [95] was the first to apply LLMs to fuzzing (i.e., the automatic generation of test cases for Python modules). Later, TitanFuzz [46] and FuzzGPT [47] 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 [74] takes a mutational one by asking the LLM to modify human-written test cases. Ackerman *et al.* [2] 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.

7.4 Symbolic Execution

Symbolic execution or whitebox fuzzing approaches are typically used to find violations of simple properties such as assertions [63, 27]. Recent whitebox fuzzing techniques do find violations of certain classes of properties. Schemmel's work [138] checks liveness properties while CHIRON [73] checks safety properties. [168, 164] proposed regular-property guided dynamic symbolic execution to find the program paths satisfying a property. However, all of these approaches require a long time budget for heavy-weight program analysis and back-end constraint solving. As a result, these techniques face challenges in scalability. In contrast, LTL-Fuzzer is built on top of greybox fuzzing; it can validate arbitrarily large and complex software implementations.

7.5 Blackbox Fuzzing

The majority of state-of-the-art testing frameworks that explore behaviours of a distributed system assume *full control* over the inherent non-determinism of runtime executions, with Jepsen being a notable exception [88]. They achieve this by either (a) replacing the networking layer [162], (b) explicitly modifying SUT to include a test harness [169, 93, 121], or (c) implementing the system in a testing-friendly language [85, 166, 48]. Controlling the asynchrony makes it possible to employ techniques from software model checking such as *partial order reduction* [79] to avoid redundancy when exhaustively exploring the space of bounded runtime executions [122, 49, 121, 100]. While these approaches allow for more effective behaviour exploration than Jepsen/Mallory, they are far more difficult to apply, requiring, (a) specific OS setup, (b) protocol-aware SUT modifications, or (c) using a domain-specific language.

Chapter 8

Conclusion

In this chapter, we summarize the proposed approaches, reflect on this thesis, and discuss our perspectives and potential future research directions.

8.1 Summary

In this thesis, we develop effective, scalable, and usable approaches for finding bugs in reactive systems. The key problem in the automated validation of reactive systems lies in their inherent statefulness: sending the same inputs twice might yield a different response every time, depending on the internal state. To address this problem, we propose a new technical framework that synergizes the concepts of model checking with recent advances in greybox fuzzing, without giving formal verification guarantees. Model checking is traditionally adopted by developers for systematically validating reactive systems, while greybox fuzzing is the most popular automated testing technique today. By synergizing both techniques, we retain the bug-finding capability from model checking, and the scalability and usability from greybox fuzzing.

We have used this technical solution to address the statefulness problem. First, we developed LTL-Fuzzer, an automated testing method that brings the ability of model checking to find stateful property violations into the domain of greybox fuzzing. Unlike traditional approaches that require modeling the system, LTL-Fuzzer directly uses greybox fuzzing on system implementations to search for violating traces. This approach has demonstrated effectiveness in practice: among 50 LTL properties derived from protocol RFCs, LTL-Fuzzer uncovered 15 previously unknown violations for both safety and liveness properties. Second, we developed ChatAFL, which leverages large language models (LLMs) to automatically identify

states from specifications, and then uses this state feedback to guide greybox fuzzing to explore previously unseen states. This approach transports the state exploration capability of model checking into greybox fuzzing. In ChatAFL, we are not aiming to have a thorough state exploration, but we still have a more principal way to explore a larger state space. In our evaluation, this lightweight approach can cover 30% more states than the state-of-the-art approaches. Beyond single-node, stateful reactive systems also operate in distributed settings. We developed Mallory, a greybox fuzzing method for testing stateful distributed systems. In Mallory, we leverage Lamport timelines to visualize executions of distributed systems and then abstract them into states, which serve as the feedback function to guide the fuzzing search. In addition, Mallory itself behaves as a reactive system, which dynamically decides what inputs to inject based on observed states to maximize the state space. In our evaluation, Mallory can cover over 50% more states than the best practical tool Jepsen. Last, system behaviors often heavily depend on complex environmental states. We developed $\mathcal{E}$fuzz to capture the effect of different environmental states. Rather than abstracting the possible environments into models, we use greybox fuzzing to mutate the environments, implicitly generating the effect of different environments. This approach is highly effective in practice. So far, $\mathcal{E}$fuzz has found about 100 previously unknown bugs in widely used programs and libraries such as `Vim` and `glibc`, many of which arise in previously unconsidered environments. All these approaches address the statefulness challenge in validating reactive systems, but from different dimensions.

8.2 Reflections

What is the impact of practical techniques for validating reactive systems within the community? In our work [108], we reflected on this question by examining the impact of AFLNet in the past five years. AFLNet is the first work that targets testing stateful network protocols, which integrates automated state model inference with coverage-guided fuzzing. AFLNet was released as an open-source tool in March 2020 [126]. Over the past five years, AFLNet has made significant contributions to research, practice, and education. In terms of research impact, the short tool demo paper of AFLNet has been cited over 381 times (as of July 2025,

according to Google Scholar), with many citations appearing in premier conferences and journals in Security and Software Engineering. Regarding practical impact, AFLNet has garnered 942 stars on GitHub and currently supports 17 protocols, 12 of which were contributed by other researchers, demonstrating its versatility and community engagement.

Security researchers have also published experience reports and tutorials on using AFLNet for challenging targets. For example, the NCC Group explored the challenges of fuzzing 5G protocols and demonstrated AFLNet's ability to uncover bugs in this critical domain. Similarly, researchers from the University of Melbourne extended AFLNet to support IPv6 for fuzz testing the software development kit (SDK) of Matter, a novel application-layer protocol designed to unify fragmented smart home ecosystems. This extension has discovered zero-day vulnerabilities in the Matter SDK. Moreover, ETAS, a subsidiary of Robert Bosch GmbH, highlighted AFLNet as a potential open-source protocol fuzzing solution in the context of the ISO/SAE 21434 standard for road vehicle cybersecurity engineering. In education, AFLNet has been introduced to hundreds of Master's students through modules such as "Security and Software Testing (SWEN90006)" at the University of Melbourne and "Fantastic Bugs and How to Find Them (17-712)" at Carnegie Mellon University.

Why has AFLNet generated such practical and academic impact in a short period of fewer than five years? We can see two reasons: (i) the open science approach and (ii) providing a practical solution to a long-standing problem of validating reactive systems. As for the *open science* approach, we strongly believe that sound progress in science requires reproducibility and that effective impact in practice requires open source, which is followed by this thesis. AFLNet is an excellent case demonstrating the success of the open science approach.

AFLNet is a practical solution to the long-standing problem of validating reactive systems. Looking back and reflecting on it, we feel this is because of the sheer dearth of suitable approaches for testing reactive systems, though there exist many approaches for testing sequential transformational systems. Prior to the greybox approach of AFLNet, reactive system validation would typically need to be carried out via algorithmic whitebox verification approaches such as model checking. AFLNet frees the practitioners from the heavy manual effort. Together, these reflections

underscore the value of having practical solutions for validating reactive systems

8.3 Path Forward

Beyond the network protocols and distributed systems mainly studied in this thesis, a growing number of modern software systems, such as autonomous vehicles and multi-agent systems, are being designed as stateful reactive systems to support interaction and communication between diverse components. Modern reactive systems are becoming increasingly complex, integrating AI-driven components, operating in dynamic environments, and facing ever-evolving security and reliability challenges.

To address these challenges, we must develop advanced testing techniques to investigate the statefulness problems more deeply. While significant progress has been made, many critical state-related problems from modern reactive systems remain and continue to evolve. For example, most of current testing techniques focus on exploring input-driven state space, where states are reached through user or environmental inputs. However, in practice, reactive systems are often implemented with concurrency (e.g., multithreading) to enhance responsiveness and throughput. In such cases, the states are shaped not just by input sequences but also by thread interleavings. Without considering these concurrency-induced execution paths, a significant portion of the state space will not be able to be exposed.

Moreover, modern software systems are increasingly heterogeneous. In the era of Artificial Intelligence (AI), many reactive systems are embedded with AI components, such as autonomous vehicles and even agentic systems. As a result, the program behaviors or states are not only determined by the source code, but also by embedded AI components. Reasoning about and validating these AI-induced states poses a new set of challenges. At the same time, today's generative AI shows impressive performance in automated programming, and more and more source code of states is generated by AI models based on specifications. However, recent studies have shown that code is more insecure with AI assistants [125, 124]. Ensuring trustworthiness in such AI-generated components, particularly in stateful and safety-critical systems, requires robust validation at both the code level and the state level.

These examples represent just a fraction of the broader state problems introduced

by modern software ecosystems. To address these statefulness problems, we have developed a technique framework that synergizes model checking with greybox fuzzing in this thesis. This technique framework can be used to address more statefulness problems. However, we can of course go far beyond this solution, and we can propose advanced techniques to better validate reactive systems. For example, one promising direction is to develop intelligent testing techniques that leverage the code understanding capability of AI models to steer testing. Traditional testing methods, such as greybox fuzzing and even symbolic execution, usually depend on mechanistic information to steer search, while intelligent testing opens the door to semantic-level reasoning, including understanding system states.

Publications Appeared

[1] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time Temporal Logic guided Greybox Fuzzing", in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1343–1355.

[2] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large Language Model guided Protocol Fuzzing", in *Proceedings of the 31st Annual Network and Distributed System Security Symposium*, 2024.

[3] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Greybox Fuzzing of Distributed Systems", in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2023, pp. 1615–1629.

[4] R. Meng, G. J. Duck, and A. Roychoudhury, "Program Environment Fuzzing", in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2024, pp. 720–734.

[5] R. Meng, V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet five years later: On coverage-guided protocol fuzzing", *IEEE Transactions on Software Engineering*, vol. 51, no. 4, pp. 960–974, 2025.

Note that this thesis does not include the algorithm for building and abstracting the Lamport timeline (i.e., Section 3.2.1 and Section 3.2.2 in [3]).

Bibliography

[1] 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.

[2] 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.

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

[4] P. Alvaro and S. Tymon, "Abstracting the geniuses away from failure testing", *Communication ACM*, vol. 61, no. 1, pp. 54–61, 2018.

[5] 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.

[6] 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.

[7] 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*, 2020, pp. 1597–1612.

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

[9] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques", *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–39, 2018.

[10] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers", *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 73–85, 2006.

[11] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of c programs", in *proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, pp. 203–213.

[12] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces", in *proceedings of the 8th international SPIN workshop on Model checking of software*, 2002, pp. 103–122.

[13] 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, 2006, pp. 343–358.

[14] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: BFT systems made robust", in *Proceedings of the 2021 Conference on Principles of Distributed Systems*, ser. LIPIcs, vol. 217, 2021, 7:1–7:29.

[15] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill, "Model checking of c and c++ with divine 4", in *proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis*, 2017, pp. 201–207.

[16] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to runtime verification", in *Lectures on Runtime Verification*, 2018, pp. 1–33.

[17] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst, "Visualizing distributed system executions", *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 2, 9:1–9:38, 2020.

[18] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification", in *proceedings of the 23rd International Conference on Computer-Aided Verification*, 2011, pp. 184–190.

[19] M. Böhme, E. Bodden, T. Bultan, C. Cadar, Y. Liu, and G. Scanniello, "Software security analysis in 2030 and beyond: A research roadmap", *ACM Transactions on Software Engineering and Methodology*, 2024.

[20] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections", *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2020.

[21] 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.

[22] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[23] 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*, 2022, pp. 1621–1633.

[24] 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.

[25] S. Bubeck, V. Chadrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4", 2023.

[26] 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, pp. 317–329.

[27] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs", in *proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 209–224.

[28] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt", in *proceedings of the 25th European Conference on Object-Oriented Programming*, 2011, pp. 609–633.

[29] F. Chen and G. Roşu, "Mop: An efficient and generic runtime verification framework", in *proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, 2007, pp. 569–588.

[30] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multi-threaded programs", in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 2325–2342.

[31] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawk-eye: Towards a desired directed greybox fuzzer", in *proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2095–2108.

[32] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code", *arXiv preprint arXiv:2107.03374*, 2021.

[33] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems", in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.

[34] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, *et al.*, "Palm: Scaling language modeling with pathways", *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[35] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking", in *proceedings of 14th International Conference on Computer-Aided Verification*, 2002, pp. 359–364.

[36] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement", in *Proceedings of the 12th International Computer Aided Verification*, 2000, pp. 154–169.

[37] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking", *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003.

[38] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs", in *proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 168–176.

[39] E. M. Clarke, "Model checking", in *Proceedings of the 17th Foundations of Software Technology and Theoretical Computer Science*, 1997, pp. 54–56.

[40] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic", *Lecture Notes in Computer Science*, vol. 131, pp. 55–71, 1981.

[41] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting finite-state models from java source code", in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 439–448.

[42] 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, pp. 391–402.

[43] J. De Ruiter and E. Poll, "Protocol state fuzzing of tls implementations", in *proceedings of the 24th USENIX Conference on Security Symposium*, 2015, pp. 193–206.

[44] N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma, "Runtime monitoring with union-find structures", in *proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2016, pp. 868–884.

[45] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer, *et al.*, "Uncovering bugs in distributed storage systems during testing (not in {production!})", in *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, 2016, pp. 249–262.

[46] 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, pp. 423–435.

[47] 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.

[48] A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia, "Compositional programming and testing of dynamic distributed systems", *Proceedings of the ACM on Programming Languages*, vol. 2, 159:1–159:30, 2018.

[49] C. Dragoi, C. Enea, B. K. Ozkan, R. Majumdar, and F. Niksic, "Testing consensus implementations using communication closure", *Proceedings of ACM on Programming Languages*, vol. 4, 210:1–210:29, 2020.

[50] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery", in *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, 2020, pp. 151–163.

[51] R. Dutra, R. Gopinath, and A. Zeller, "Formatfuzzer: Effective fuzzing of binary file formats", *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–29, 2023.

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

[53] E. A. Emerson and J. Y. Halpern, ""sometimes" and "not never" revisited: On branching versus linear time temporal logic", *Journal of the ACM*, vol. 33, no. 1, pp. 151–178, 1986.

[54] Y. Falcone, S. Krstić, G. Reger, and D. Traytel, "A taxonomy for classifying runtime verification tools", *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 2, pp. 255–284, 2021.

[55] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models", in *2023 IEEE/ACM 45th International Conference on Software Engineering*, 2023, pp. 1469–1481.

[56] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference", in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 337–350.

[57] 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.

[58] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of dtls implementations using protocol state fuzzing", in *proceedings of the 29th USENIX Conference on Security Symposium Security*, 2020, pp. 2523–2540.

[59] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations", in *proceedings of 28th International Conference on Computer-Aided Verification*, 2016, pp. 454–471.

[60] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of ssh implementations", in *proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017, pp. 142–151.

[61] X. Gao, G. J. Duck, and A. Roychoudhury, "Scalable fuzzing of program binaries with e9afl", in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, 2021, pp. 1247–1251.

[62] P. Godefroid, "Model checking for programming languages using verisoft", in *proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997, pp. 174–186.

[63] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing", in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[64] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing." In *Proceedings of the 2008 Network and Distributed System Security Symposium*, vol. 8, 2008, pp. 151–166.

[65] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction", in *proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 265–278.

[66] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework", in *proceedings of 27th International Conference on Computer-Aided Verification*, 2015, pp. 343–361.

[67] D. Harel and A. Pnueli, "On the development of reactive systems", in *Logics and models of concurrent systems*, 1984, pp. 477–498.

[68] J. Hatcliff, M. B. Dwyer, and H. Zheng, "Slicing software for model construction", *Higher-order and symbolic computation*, vol. 13, no. 4, pp. 315–353, 2000.

[69] M. Heizmann, J. Hoenicke, and A. Podelski, "Software model checking for people who love automata", in *proceedings of 25th International Conference on Computer-Aided Verification*, 2013, pp. 36–52.

[70] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects", *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

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

[72] G. J. Holzmann, "The model checker spin", *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[73] E. Hoque, O. Chowdhury, S. Y. Chau, C. Nita-Rotaru, and N. Li, "Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs", in *proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017, pp. 627–638.

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

[75] J. Hu, "Membership rollback issue in canonical raft", GitHub issue available at https://github.com/canonical/raft/issues/250, 2021.

[76] M. Huth and M. Ryan, "Logic in Computer Science: Modelling and reasoning about systems", 2004.

[77] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib", in *proceedings of the 27th International Conference on Computer-Aided Verification*, 2015, pp. 487–495.

[78] 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.

[79] R. Jhala and R. Majumdar, "Software model checking", *ACM Computing Surveys*, vol. 41, no. 4, 21:1–21:54, 2009.

[80] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "Javamop: Efficient parametric runtime monitoring framework", in *proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 1427–1430.

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

[82] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie: Fuzzing windows applications with harness synthesis and fast cloning", in *Proceedings of the 2021 Network and Distributed System Security Symposium*, 2021.

[83] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk, "Ltsmin: High-performance language-independent model checking", in *proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 692–707.

[84] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code", in *proceedings of the 4th USENIX conference on Networked systems design and implementation*, 2007, pp. 243–256.

[85] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat, "Mace: Language support for building distributed systems", in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 179–188.

[86] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "Pgfuzz: Policy-guided fuzzing for robotic vehicles", in *proceedings of 2021 Network and Distributed Systems Security Symposium*, 2021.

[87] K. Kingsbury, "Elle: Finding isolation violations in real-world databases", in *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*, 2021, p. 7.

[88] K. Kingsbury, "Jepsen", Accessed on 10 April 2023, 2023. [Online]. Available: https://jepsen.io/.

[89] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing", in *proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[90] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Communication ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[91] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem", *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

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

[93] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems", in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 399–414.

[94] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java api specifications", in *proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 602–613.

[95] 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, pp. 919–931.

[96] M. Leucker and C. Schallhart, "A brief account of runtime verification", *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

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

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

[99] 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.

[100] J. F. Lukman *et al.*, "FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems", in *Proceedings of the 14th European Conference on Computer Systems*, 2019, 20:1–20:16.

[101] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?", vol. 2, 2018.

[102] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference", in *Proceedings of the 42nd International Conference on Software Engineering*, 2020, pp. 1024–1036.

[103] Z. Manna and A. Pnueli, "Temporal verification of reactive systems: safety", Springer Science & Business Media, 2012.

[104] F. Mattern, "Virtual time and global states of distributed systems", in *Proc. Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[105] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time Temporal Logic guided Greybox Fuzzing", in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1343–1355.

[106] R. Meng, G. J. Duck, and A. Roychoudhury, "Program Environment Fuzzing", in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2024, pp. 720–734.

[107] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large Language Model guided Protocol Fuzzing", in *Proceedings of the 31st Annual Network and Distributed System Security Symposium*, 2024.

[108] R. Meng, V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet five years later: On coverage-guided protocol fuzzing", *IEEE Transactions on Software Engineering*, vol. 51, no. 4, pp. 960–974, 2025.

[109] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Greybox Fuzzing of Distributed Systems", in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2023, pp. 1615–1629.

[110] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing", in *proceedings of the 1st international workshop on Random testing*, 2006, pp. 46–54.

[111] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code", *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 75–88, 2002.

[112] M. Musuvathi, S. Qadeer, T. Ball, M. Musuvathi, S. Qadeer, and T. Ball, "Chess: A systematic testing tool for concurrent software", Microsoft Research, Tech. Rep. MSR-TR-2007-149, 2007.

[113] "Namazu", Accessed on 10 April 2023, 2023. [Online]. Available: https://github.com/osrg/namazu.

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

[115] 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*, 2021, pp. 662–665.

[116] G. Neumann, M. Harman, and S. Poulding, "Transformed vargha-delaney effect size", in *Proceedings of Search-Based Software Engineering: 7th International Symposium*, 2015, pp. 318–324.

[117] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability", in *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017, pp. 377–389.

[118] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm", in *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014, pp. 305–319.

[119] OpenAI, "Gpt-4 technical report", 2023. arXiv: 2303.08774 [cs.CL].

[120] OpenSSL, "Heartbleed Vulnerability", https://en.wikipedia.org/wiki/Heartbleed.

[121] B. K. Ozkan, R. Majumdar, F. Niksic, M. T. Befrouei, and G. Weissenbacher, "Randomized testing of distributed systems with probabilistic guarantees", *Proceedings of ACM on Programming Languages*, vol. 2, 160:1–160:28, 2018.

[122] B. K. Ozkan, R. Majumdar, and S. Oraee, "Trace aware random testing for distributed systems", *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–29, 2019.

[123] 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.

[124] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions", *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[125] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" In *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, 2023, pp. 2785–2799.

[126] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: A greybox fuzzer for network protocols", in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, 2020, pp. 460–465.

[127] 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.

[128] S. Poeplau and A. Francillon, "Symbolic execution with {symcc}: Don't interpret, compile!" In *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 181–198.

[129] "Promela", http://spinroot.com/spin/Man/promela.html.

[130] 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*, vol. 32, no. 160, pp. 1–26, 2023.

[131] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar", in *proceedings of the 1982 International Symposium on programming*, 1982, pp. 337–351.

[132] 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.

[133] Z. Rakamarić and M. Emmi, "Smack: Decoupling source language details from verifier implementations", in *proceedings of the 26th International Conference on Computer-Aided Verification*, 2014, pp. 106–113.

[134] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing", in *Proceedings of the 24th Network and Distributed System Security Symposium*, 2017.

[135] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing", in *proceedings of 2017 Network and Distributed Systems Security Symposium*, vol. 17, 2017, pp. 1–14.

[136] G. Reger, H. C. Cruz, and D. Rydeheard, "Marq: Monitoring at runtime with qea", in *proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015, pp. 596–610.

[137] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences", in *Proceedings of the 21st annual ACM symposium on Theory of computing*, 1989, pp. 411–420.

[138] D. Schemmel, J. Büning, O. S. Dustmann, T. Noll, and K. Wehrle, "Symbolic liveness analysis of real-world software", in *proceedings of 30th International Conference on Computer-Aided Verification*, 2018, pp. 447–466.

[139] M. Schordan, J. Hückelheim, P.-H. Lin, and H. Menon, "Verifying the floating-point computation equivalence of manually and automatically differentiated code", in *proceedings of the 1st International Workshop on Software Correctness for HPC Applications*, 2017, pp. 34–41.

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

[141] 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.

[142] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c", *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[143] "Spot", https://spot.lrde.epita.fr/.

[144] D. Steinhöfel and A. Zeller, "Input invariants", in *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 583–594.

[145] 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.

[146] 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.

[147] O. Tkachuk, M. Dwyer, and C. Pasareanu, "Automated environment generation for software model checking", in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, 2003, pp. 116–127.

[148] F. Vaandrager, "Model learning", *Communications of the ACM*, vol. 60, no. 2, pp. 86–95, 2017.

[149] M. Y. Vardi, "Automata-theoretic model checking revisited", in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2007, pp. 137–150.

[150] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification", in *proceedings of the 1st Symposium on Logic in Computer Science*, 1986.

[151] M. Y. Vardi and P. Wolper, "Reasoning about infinite computations", *Information and Computation*, vol. 115, pp. 1–37, 1 1994.

[152] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong", *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[153] 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.

[154] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs", *Automated software engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[155] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities", in *proceedings of the 42nd International Conference on Software Engineering*, 2020, pp. 999–1010.

[156] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, A. X. Liu, and R. Beyah, "Mpinspector: A systematic and automatic approach for evaluating the security of iot messaging protocols", in *proceedings of the 30th USENIX Security Symposium*, 2021, pp. 4205–4222.

[157] 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.

[158] Z. Wang, B. Liblit, and T. W. Reps, "Tofu: Target-orienter fuzzer", *CoRR*, vol. abs/2004.14375, 2020.

[159] C. J. Watkins, "Learning from delayed rewards", Ph.D. dissertation, King's College, Cambridge United Kingdom, 1989.

[160] C. J. Watkins and P. Dayan, "Q-learning", *Machine learning*, vol. 8, pp. 279–292, 1992.

[161] 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.

[162] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: transparent model checking of unmodified distributed systems", in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009, pp. 213–228.

[163] N. Yang, K. Aslam, R. Schiffelers, L. Lensink, D. Hendriks, L. Cleophas, and A. Serebrenik, "Improving model inference in industry by combining active and passive learning", in *proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*, 2019, pp. 253–263.

[164] H. Yu, Z. Chen, J. Wang, Z. Su, and W. Dong, "Symbolic verification of regular properties", in *proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 871–881.

[165] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm, "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems", in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association, 2014, pp. 249–265.

[166] X. Yuan and J. Yang, "Effective concurrency testing for distributed systems", in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1141–1156.

[167] M. Zalewski, "AFL", https://lcamtuf.coredump.cx/afl/.

[168] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu, "Regular property guided dynamic symbolic execution", in *proceedings of the 37th IEEE International Conference on Software Engineering*, 2015, pp. 643–653.

[169] J. Zhou, A. Miller, R. Sears, W. Wilson, A. Grieser, and B. Muppana, "FoundationDB: A Distributed Unbundled Transactional Key Value Store", in *Proceedings of the 2021 ACM SIGMOD/PODS International Conference on Management of Data*, 2021, pp. 2653–2666.