UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Enhancing Fuzzing Using Stochastic Modeling and Generative AI

Permalink
https://escholarship.org/uc/item/9b60b2fc

Author
Hu, Jie

Publication Date
2024

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org
Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Enhancing Fuzzing Using Stochastic Modeling and Generative AI

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jie Hu

September 2024

Dissertation Committee:

Prof. Heng Yin, Chairperson
Prof. Chengyu Song
Prof. Zhijia Zhao
Prof. Qian Zhang

Copyright by
Jie Hu
2024

The Dissertation of Jie Hu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Heng Yin. Without his invaluable help, I would not have reached this point. His extensive knowledge, guidance, patience, encouragement, and support have been crucial throughout my journey as a Ph.D student.

Additionally, I would like to thank the other members of my dissertation committee: Prof. Chengyu Song, Prof. Zhijia Zhao and Prof. Qian Zhang, for their insightful questions and comments.

I also extend my heartfelt thanks and big hugs to all my fellow labmates and friends: Yue, Jinghan, Wei, Lian, Zixiang, Zhenxiao, Sheng, Zhaoqi, Xiao, and Xiaolin, for their assistance with my research, their emotional support during the challenging times, and most importantly, the fun moments we shared together.

Lastly, I would like to thank my parents, Gongwei Hu and Li Chen, for shaping me who I am through their unwavering support and unconditional love. I also send lots of kisses to my cat, Charlie, for being a sweet companion during the sleepless nights before paper deadlines.

This dissertation includes previously published material entitled "MARCO: A Stochastic Asynchronous Concolic Explorer" published in the 46th IEEE/ACM International Conference on Software Engineering. 2024, and "Augmenting Greybox Fuzzing with Generative AI" released on Arxiv in 2023, and one material under submission entitled "How Well can LLMs Generate Fuzzing Inputs".

To my parents for all the love and support.

ABSTRACT OF THE DISSERTATION

Enhancing Fuzzing Using Stochastic Modeling and Generative AI

by

Jie Hu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2024
Prof. Heng Yin, Chairperson

Fuzzing, or fuzz testing, is an automated program testing technique aimed at uncovering security flaws in software. This method generates and injects crafted inputs into a Program-Under-Test (PUT) to monitor for abnormal runtime behavior, with the goal of identifying potential defects and vulnerabilities. A generic fuzzing framework involves two essential components that influence its testing performance: 1) a **Scheduler** that determines the current optimal strategy for program state space exploration, and 2) a **Mutator** that produces concrete inputs based on the scheduler's strategy.

In this thesis, we propose an enhanced fuzzing framework leveraging stochastic modeling and generative AI (genAI).

Firstly, we introduce an intelligent scheduler that models program execution traces as Markov Chain, with transitions between branches stochastically represented. A reinforcement learning algorithm allows the scheduler to refine its exploration strategy for more efficient and comprehensive testing of the PUT, by learning from outcomes of past explorations. We integrate this scheduler in a whitebox fuzzer/concolic executor, Marco, and evaluate it against state-of-the-art concolic executors on real-world programs, demonstrating Marco's superior performance.

Secondly, we propose to enhance the mutator using genAI. To fully reveal the potential of uncustomized, out-of-the-box large language models (LLMs) for producing high quality inputs for fuzzing, we conduct a comprehensive study of eight state-of-the-art (SOTA) LLMs, encompassing both large and small models from three LLM families. This study establishes a baseline for the fuzzing capabilities of uncustomized LLMs and provides insights for developing more effective collaboration strategies between conventional and LLM-based mutators. We also showcase the performance of an LLM-empowered greybox fuzzer, ChatFuzz, against state-of-the-art greybox fuzzers, demonstrating that ChatFuzz significantly improves coverage findings, and is comparable to or better than the baseline approach for vulnerability detection.

Finally, motivated to tackle the path divergence issue, where the input produced by the mutator diverges from the desired path selected by the scheduler, and inspired by LLMs' potential as intelligent mutators, we propose a path-corrective LLM-based mutator to further enhance fuzzing. Specifically, we identify path-divergent inputs and use specialized prompts to instruct the LLM to generate corrected inputs that follow the desired paths. We present encouraging preliminary results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's fast-paced and digitally interconnected world, the reliability of software systems has become more critical than ever. With industries and infrastructures relying heavily on these systems, improving their robustness against potential security flaws has become a top priority. One of the most popular techniques for finding security bugs in software is program fuzz testing, i.e., fuzzing.

Fuzzing involves generating random pieces of data as program inputs and inputting them to observe the behavior of the program under test (PUT). The primary goal is to trigger bugs within the program under test (PUT), and the first crucial step is generating an input that can reach the vulnerable code. Consequently, the key challenge in fuzzing is the automatic generation of high-quality inputs that can effectively cover the program's code regions.

In a typical fuzzing framework, two essential components—the **Scheduler** and the **Mutator**—significantly influence the quality of the generated seeds. During program fuzz

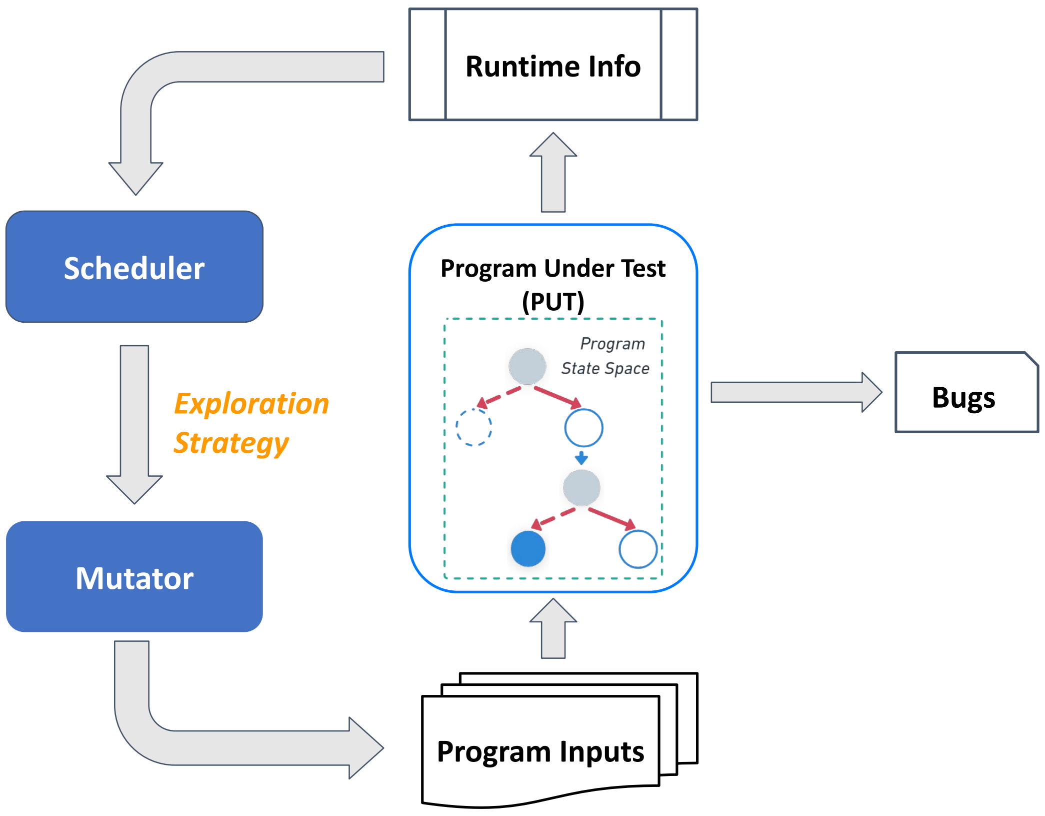


Figure 1.1: Fuzzing Overview

testing, the scheduler determines the optimal exploration strategy for the program state space based on runtime information from previous executions. Following this, the mutator generates a series of new program inputs based on the chosen strategy to test the program under test (PUT), as is demonstrated in Figure 1.1.

Despite notable progress in optimizing fuzzing techniques, several challenges continue to limit their effectiveness.

Whitebox fuzzing leverages symbolic tracing and constraint solving to generate new inputs for program testing [10] and can be implemented through concolic execution. A concolic execution engine performs symbolic tracing along a concrete path, exercised by a concrete input, while collecting the path constraints of the untaken branches at each symbolic branch point. By solving these path constraints, new program inputs are generated to explore previously untaken branches. However, the number of branches and states increases exponentially due to the existence of loops and function calls, which commonly exist in almost all real-world programs. The high cost of solving path constraints and the existence of path divergence make it impractical to thoroughly explore all branches and states within reasonable time and resource limits. The *State Explosion* problem can be mitigated by designing a more efficient scheduler for the fuzzing framework.

According to the optimal exploration strategy set by the **Scheduler**, the **Mutator** component start to craft new program inputs for the testing. Depending on how new seeds are generated, a mutator can follow either a random or directional approach. A random mutator is commonly adopted by greybox or blackbox fuzzer, where a new seed is generated by creating random variation of an existing one through random mutation, or constructed from scratch based on knowledge of the grammar. Due to the random nature of such generation, the fuzzer has little to no control over the path to be exercised by the new seeds. The effectiveness of such random mutations in uncovering program states comprehensively is questionable. In whitebox fuzzing, the mutator adopts a directional approach, where new inputs are generated by solving path constraints collected from specific branches. The resulting inputs are expected to follow paths through these targeted branches. However, when the new input fails to traverse the predicted path, a phenomenon known as *Path Divergence* occurs, which impairs the soundness and completeness of whitebox fuzzing.

To improve the performance of fuzzer, many recent works have adopted machine learning techniques [29, 68, 22, 9]. Recent breakthroughs in attention-based large language models (LLM) [64] have inspired us to explore the capability of LLM for fuzzing.

1.1 Thesis Statement

To sum up, although fuzzing has demonstrated its power in exposing security flaws in computer systems, serious limitations still exist and hinder the effectiveness of fuzzers in exploring deep program states. Leveraging the power of stochastic modeling and generative AI has shown great potential for improving the quality of seeds. We present three projects in this thesis that leverage stochastic modeling and generative AI to address the limitations in the *Scheduler* and *Mutator*, thereby enhancing the performance of fuzzing:

Firstly, we present Marco, where we conducted a study to uncover significant limitations in state-of-the-art branch-flipping policies and proposed an intelligent branch scheduler to address these issues. Specifically, we model the symbolic execution trace as a Markov Chain and introduce a novel metric called the reachability score. This metric assesses each branch based on its potential to uncover new code coverage, incorporating reinforcement learning techniques to progressively refine the evaluation of each branch's reachability, thereby facilitating the generation of higher-quality test cases for more effective testing.

Secondly, we conduct a systematic study across a diverse range of large language models (LLMs) to gain insights into their fuzzing capabilities. Based on the findings from this study, we provide concrete guidelines for developing an LLM-empowered greybox fuzzing framework. Such guidelines are designed to be adaptable to future LLM models and variants, ensuring its continued relevance as the technology evolves. Leveraging the insights from this study, we implement a generative-AI-augmented greybox fuzzer and introduce our prototype, ChatFuzz. Our evaluations of the prototype demonstrate its superior performance compared to the baseline approach.

Thirdly, we aim to mitigate path divergence to further improve whitebox fuzzing. We propose a path-corrective, LLM-based mutator that assists the concolic executor by generating new mutations that correct the path-divergent seeds produced by the concolic executor, thereby enhancing the overall performance of concolic execution. We integrate our proposed mutator in a concolic executor, Marco, to present a prototype Polaris.

Chapter 2

Background

2.1 Whitebox Fuzzing

2.1.1 Symbolic Execution

Symbolic Execution (SE) is an automated program testing technique that aims to maximize code coverage by generating specific inputs to satisfy every condition check that is dependent on the input within the program under test. With SE, the program is executed with symbolic expressions instead of concrete values. An SE engine maintains 1) the mapping between program variables and symbolic expressions, and 2) a set of path predicates imposed by the sequence of branches visited along the execution path.

Two types of SE are extensively researched: 1) online symbolic execution and 2) concolic execution. Online symbolic execution engines, such as KLEE [13] and S2E [23], explore the program space via state forking: when encountering a branch point (whose direction is dependent on the input), an SE engine will fork a new state to explore the opposite branch direction (if it is feasible). As a result, the number of states grows exponentially, leading to the state explosion problem. To tackle the state explosion problem, some recent works [44, 34] resorted to machine learning. Legion [44] leverages Monte Carlo Tree Search (MCTS) to model the state exploration as a sequential decision-making process on the tree-structured program space. Symbolic execution is performed lazily and only when a state is deemed promising. Learch [34] trains a regression model on a set of training programs to learn the state selection policy based on a set of state-describing features. Then, the trained model is used to test unseen target programs.

2.1.2 Concolic Execution

Unlike SE, concolic execution (CE) explores the program space iteratively. Given an input, a CE engine (e.g., QSYM [75], SymCC [51], and SymSan [18]) executes the program concretely and simultaneously collects symbolic constraints along its concrete execution path. When a symbolic branch point (whose direction depends on the input) is encountered, the CE engine collects the constraint of the current branch condition to dictate which branch direction is taken by the concrete execution. Additionally, based on a branch-flipping policy, the CE engine may decide to generate a new input that can traverse the untaken branch direction. To do so, the CE engine constructs a constraint set that includes the negated current branch condition and a number of preceding branch conditions and queries an SMT (Satisfiability Modulo Theories) solver for a solution. Then a new input is generated by replacing parts of the original input with the values suggested by the solution. After the CE finishes processing the current input, it will pick and process one of the newly generated inputs. Obviously, the branch-flipping policy is essential for concolic execution.

2.1.3 Branch Flipping Policies

The most naïve branch-flipping policy would be "flip all". As the name suggests, this policy tries to flip all possible branches. This policy can ultimately achieve the highest code coverage, given unlimited computing resources and time. No one has ever adopted this policy because computing resources and time are never unlimited, and many branches are either redundant or unworthy to be flipped.

A more realistic branch-flipping policy is to flip every branch executed through a unique execution path prefix. More specifically, this path prefix consists of a list of symbolic branches along the execution path, while concrete branches are ignored. For this reason, we refer to this policy as "PP policy". However, even with this policy, the number of branches to be flipped can still be enormous. This is because a program often contains loops and function calls, and one branch that appears in different loop iterations and different calling contexts will be flipped repeatedly due to its unique path prefix in each loop iteration and each calling context. Yun et al. observed that constraints repetitively generated by the same code are useless for finding new code coverage in real-world software [75].

Based on this observation, existing state-of-the-art CE engines (e.g., QSYM [75], SymCC [51], and SymSan [18]) adopt a more restrictive branch-flipping policy, which was first introduced in QSYM. This policy looks at branch bigrams. It will flip the current branch if its bigram (i.e., the pair of the previous symbolic branch and the current one) is new. We refer to this policy as the "BR policy" because it focuses on branches rather than paths. Compared to the PP policy, the BR policy will flip significantly fewer branches because only the last symbolic branch is included in the "context" of the current branch instead of all preceding symbolic branches.

Some CE engines explore the branch selection heuristics with respect to the branch locality. In particular, SAGE [28] executes inputs in descending order of their code coverage and only flips branches that are located below the point where the current execution trace branches off from its parent trace to avoid redundant exploration. To efficiently explore uncovered branches, CREST [12] proposes a control flow graph (CFG) directed searching algorithm to prioritize branches in close proximity to uncovered branches through the statically constructed control flow graph and call graph. Specifically, each branch is evaluated by a scalar value obtained by adding up 1) the length of the shortest path to its nearest uncovered branch and 2) the number of flipping attempts devoted to it. CREST then flips branches in ascending order of this scalar value.

2.2 Greybox Fuzzing

2.2.1 Coverage-guided Greybox Fuzzing

Grey-box fuzzing begins with a seed input, executing the program and iteratively generating new inputs by mutating the previous ones. New inputs are added to the queue if they improve a specified guidance metric such as branch coverage. American Fuzzy Lop is one of the most widely used fuzzing tools [76]. Traditional coverage-guided fuzz testing faces challenges in efficiency and effectiveness due to a vast space of inputs and unbounded program paths. Lemieux et al. tackle this by identifying rarely executed branches with AFL-generated inputs and devising custom mutations to prioritize the exploration of these branches [41]. As a result, it requires fewer fuzzing loops and achieves higher coverage in less time. Other approaches incorporate symbolic execution in fuzzing to guide careful selection and mutation of the inputs, invoking unique program paths [62, 16]. Padhye et al. introduce Zest [48], which incorporates the semantic validity of input mutations by mapping bit-level changes to valid structural modifications, reducing the search space of inputs.

The effectiveness of fuzz testing heavily relies on the quality of initial seeds. Yet, commonly used seeds tend to traverse similar "high-frequency" paths. To expand path coverage without significantly increasing the number of tests, researchers have designed strategies for seed selections. AFLFast [8] models coverage-based greybox fuzzing as a Markov chain, and assigns different selection probabilities for different seeds. EcoFuzz [74] improves AFLFast's Markov chain model and presents a variant of the Adversarial Multi-Armed Bandit model. EcoFuzz sets three states of the seeds set and develops a unique adaptive scheduling algorithm.

2.2.2 Grammar-Based Test Generation

One challenge in grey-box fuzz testing is generating valid inputs, especially for highly-structured inputs and object-oriented programs. This has led to research efforts to minimize unfruitful fuzzing iterations by generating legal inputs for the target program using input grammars. For example, CodeAlchemist [32] is a code generation engine that can systematically generate both syntactically and semantically correct JavaScript code snippets. Token-Level AFL [56] mutates JavaScript programs at a token level. Tokens from a dictionary are used when mutating programs to replace, insert, or overwrite existing tokens. Le et al. propose a grammar-based fuzzing approach called Saffron that relies on a user-defined grammar [39]. During fuzzing, if an input generated by the grammar leads to a program failure, Saffron reconstructs the grammar according to newly learned input specifications of the program. Wang et al. leverage a user-provided grammar, but instead of arbitrary mutations, they introduce grammar-specific mutations to diversify test inputs for tightly formatted input domains such as XML and JSON [67]. Gopinath et al. highlight that the state-of-art grammar-aware fuzzer *dharma* [1] is still two orders of magnitude slower than a random fuzzer and suggest guidelines for efficient grammar-aware fuzzing [31]. In their follow-up work, they present an approach to infer an input grammar from the interactions between an input parser and input data [30].

All existing grammar-based techniques require developers and users to understand the input grammar or manually create data generators. Additionally, fuzzing techniques developed for one type of grammar may not readily translate to other grammar types. In this work, we leverage large language models to automatically infer grammar from seed inputs and generate new valid test cases, mitigating the need for manual grammar specification.

2.2.3 Seed Scheduling

Many techniques have been proposed to improve fuzzing [17, 43, 69, 77, 33, 27, 7, 78, 47, 66], with one key optimization being seed selection [54, 35]. AFLfast [8] prioritizes less explored paths by allocating more resources to them. Vuzzer [53] emphasizes test cases that are more likely to expose vulnerabilities. Entropic [6] uses information-theoretic entropy to schedule seeds, optimizing for coverage gains and bug discovery. AFL-Hier [65] applies reinforcement learning to schedule seeds clustered by multi-level coverage metrics, while K-scheduler [59] utilizes graph centrality analysis to prioritize seeds with higher potential for reaching new code coverage.

2.3 Artificial Intelligence (AI)

2.3.1 Large Language Models

Pre-trained Large Language Models(LLMs) represents a category of neural networks characterized by a huge amount of parameters. Typically, these models are trained on a large corpus of text data in an autoregressive manner, where they learn to predict the next token in a text sequence. Such extensive pre-training enables them to operate as one-shot or zero-shot learners [11]. In other words, these models can undertake a variety of tasks when given only one single example of the task or natural language-based instructions. The instructions, alongside any supplementary input data, provided to LLMs are commonly referred to as prompts.

Previous works leveraging LLMs have employed various optimization methods, including finetuning and prompt engineering, to tailor the LLMs for specific tasks. Finetuning involves updating the model parameters of a pretrained LLM on a task-specific dataset, specifically collected for the subsequent tasks. Prompt engineering, on the other hand, does not involve model parameter updating. It focuses on developing effective natural language prompts to instruct the LLM towards desired outputs. In this paper, our focus remains on prompt engineering without modifying the internal working of the LLMs. In other words, we investigate the baseline performance of LLMs as an advanced program input generator without modifying the internal parameters.

2.3.2 AI-based Fuzz Testing

Previous research has explored the application of AI techniques in program and test case generation [58, 57, 70]. For example, NEUZZ [58] introduces a gradient-descent based approach to fuzz testing by creating a smooth surrogate function to approximate the target program's discrete branching behavior. AthenaTest [63] trains local transformer-based networks to generate test inputs from a corpus of focal methods and test inputs. Jigsaw [37] is a program synthesis tool based on LLMs and validates their correctness using existing test cases, which diverges from the focus of this paper.

The use of the prompting LLMs has also gained prominence in recent research efforts. Bareiß et al. employ Codex by providing prompts containing one method-test pair and the method to be tested, enabling the generation of test oracles and test cases [5]. ChatUniTest [72] uses ChatGPT to generate Junit test cases. FuzzGPT [24] leverages LLMs to generate unusual programs seeking to trigger abnormal behavior in deep learning library APIs. CodaMosa [40] uses Codex as a black-box tool to generate tests without requiring explicit training, and incorporate these tests into a search algorithm.

In ChatAFL [46], LLM is prompted to extract the machine-readable grammar of the target protocol, which is used for structure-aware mutation. Throughout the fuzzing process, LLM is leveraged to diversify the input corpus and to generate appended messaged to existing sequences, enabling state transition into deeper program space when the fuzzer reaches a coverage plateau and gets stuck. In ChatAFL, the LLM has to have comprehensive understanding of the subject protocol for it to work. During the grammar extraction stage, the only target-related information included in the prompt is the name of the protocol, highlighting ChatAFL's reliance on LLM's extensive knowledge of the target protocol's interval working to generate accurate grammar. The scalability of ChatAFL to protocols with complex state space has not been thoroughly investigated.

Fuzz4All [71] utilizes a powerful LLM, GPT4, to distill a concise and informative prompt for fuzzing. This involves extracting information from the documentation of the System Under Test (SUT), example code snippets, or specifications. Subsequently, it leverages a generation LLM, StarCoder, to use the previously generated prompt for generating new test cases. In essence, to operate effectively as a seed mutator, the LLM is provided with information regarding the target program's functionality as well.

2.3.3 Human-in-the-loop Fuzzing

Prior works have explored human-in-the-loop approach for fuzzing [38, 3, 60]. In HaCRS [60], humans are provided information related to the target application's behavior and interact with the application via a text-based window to figure out how to trigger certain behavior of the target. Evaluation result suggests that even non-expert human can significantly enhance the bug detection ability of the vulnerability detection tool. IJON [3] allows human analyst annotate the target program to guide the fuzzer to explore deep program states. Our approach incorporates LLM to assist fuzzer. The LLM is pretrained on enormous amount of data and can grasp and preserve the format of the sample input while generating new variations.

Chapter 3

Marco: A Stochastic Asynchronous Concolic Explorer

Concolic execution (CE), which conducts concrete and symbolic execution of the program under test (PUT) simultaneously, is a program testing technique used for code exploration and vulnerability detection. Unlike dynamic symbolic execution (DSE), which explores the program space by using symbolic inputs [13], concolic execution is performed with a concrete input exercising a concolic path consisting of branches that are dependent on a subset of input bytes. Each input-dependent branch in the concolic path has two directions: 1) a visited direction that is traversed by the concrete execution; 2) an unvisited direction that can potentially lead to a new path. Concolic execution effectively explores the program space by generating new inputs that traverse these unvisited directions of input-dependent branches.

Although powerful, concolic execution is known to be costly. As a result, many efforts have been made to improve the runtime efficiency of CE over the past few years, in terms of both constraint collection and constraint solving [75, 51, 52, 18, 19].

In contrast, another essential component in concolic execution, branch-flipping policy, has not yet received enough attention. A branch-flipping policy dictates which symbolic branch needs to be flipped to generate a new testcase traversing the flipped branch. State-of-the-art (SOTA) CE engines employ a very restrictive branch-flipping policy – to flip only a very small fraction of symbolic branches that are most likely to reach new code coverage – in order to suppress testcase/path explosion problems, where the number of generated testcases quickly surpasses CE's processing capacity.

In this project, we conduct the first study on this branch-flipping policy and have a few unique observations. On the one hand, we show that this policy is too strict, and misses many good branches that could lead to much higher code coverage. On the other hand, this policy is not as effective as expected since only a small fraction (on average 27%) of branches selected by this policy can actually lead to new code coverage. Consequently, we observe that this rigid and nearsighted branch-flipping policy significantly undermines the effectiveness of CE.

Moreover, we show that the path divergence problem [4] (i.e., the testcase generated by CE does not follow the expected path) can be as high as 50% in practice and is a norm rather than an exception due to the imperfect design and implementation of CE. Therefore, we argue that a good branch-flipping policy needs to model the path divergence on each symbolic branch when selecting the next branch to flip.

To overcome the limitations, we propose a global-view new-coverage directed branch scheduling algorithm for concolic execution. To find out which symbolic branch is the best to flip, we estimate the potential of each symbolic branch (i.e., how likely we can reach new code coverage by flipping this branch) and select the branch with the highest potential. Specifically, we model the concolic execution as a Markov process: each branch transition is a probabilistic event, and an execution path is a sequence of branch transitions, and thus a sequence of probabilistic events. To obtain a global view of all testcases, we observe the executions of all testcases and construct a stochastic Concolic State Transition Graph (CSTG) to characterize transition probabilities between states and estimate the probability of a given branch reaching any unvisited states. We refer to this probability as reachability score. This reachability score is further dampened by the path divergence rate observed on this branch.

To select the best branch (i.e., one with the highest reachability score) to flip, our branch selection must be asynchronous. When encountering a symbolic branch, the existing CE engines decide synchronously whether to flip it based on the historical information collected up to this point. This decision, however, might not be globally optimal because a seemingly good branch to flip might have already been traversed by the remaining execution of the current testcase or the remaining testcases that have not been processed yet. Therefore, we propose to process all testcases to maintain an up-to-date global view (in the form of CSTG) and then asynchronously select the best branch to flip. To do so, we develop an efficient concolic state saving and restoring mechanism. We save the symbolic expression table and branch dependency information for quick reloading after the highest potential state is identified.

To evaluate the efficacy of this idea, we implement a prototype called MARCO[1], atop SymSan [18]. We evaluated MARCO on 16 real-world programs and 71 programs from the DARPA Cyber Grand Challenge (CGC) binary set to demonstrate that MARCO, on average, increases edge coverage by 13.03%. For 3 out of the 11 programs where MARCO finds more coverage, it also covers all edge coverage found by SymSan. We further evaluate its bug detection efficiency on 14 programs from Unifuzz [42]. The result shows that our approach can find 33.52% more unique bugs than the SOTA CE engine SymSan. Furthermore, MARCO can uniquely find more than twice of bugs than SymSan does. On 5 of the tested programs, MARCO finds more unique bugs in 12h than any of the seven fuzzers evaluated in UniFuzz (excluding QSYM, which is configured as a hybrid fuzzer) can find in 24h experimental runs.

We evaluate the state-of-the-art branch-flipping policy and reveal several important yet unreported limitations. We propose a stochastic and asynchronous branch scheduling algorithm that is able to effectively pick the most promising branch for new input generation. We implement a prototype MARCO and evaluate its efficacy on 16 real-world applications. The experimental results demonstrate that MARCO can constantly outperform the SOTA in terms of coverage finding and bug detection. We open-source the implementation of our prototype at `https://zenodo.org/record/8339481`.

3.1 Motivation

To understand the effectiveness of different branch-flipping policies, we conducted a measurement study. We equipped SymCC [51], one of the SOTA CE engines, with both

[1]Named after Marco Polo, and is also short for **Mar**kov **Co**ncolic Explorer.

PP and BR policies. We selected four programs in binutils (`objdump`, `size`, `nm-new`, and `readelf`) and assembled an input corpus of 1000 seeds for each of them. We made the following four observations.

Table 3.1: Code Coverage w/ Single-Pass Exploration

Program	**Code Coverage**	
	BR	**PP**
objdump	3571	4032(+13%)
size	2128	2192(+3%)
nm-new	1995	2040(+2%)
readelf	2461	3527(+**43%**)

(1) The BR policy is so overly strict that it filters out many promising branches. We investigate if the branches discarded by the BR policy are indeed useless for reaching higher code coverage. To answer this question, for each program, we compared the code coverage after SymCC processed the same input corpus and attempted to flip the branches according to the BR and PP policies, respectively. To simplify the evaluation, we did not allow SymCC to further process testcases generated from the initial input corpus. Table 3.1 lists the results. We can see that for all four programs, the PP policy achieved higher code coverage than the BR policy. For `readelf`, the PP policy achieved a whopping 43% higher code coverage. This evaluation shows that the BR policy can miss promising branches that could lead to higher code coverage[2].

(2) The strict BR policy often leads to early termination. Observation 1 tells us that the BR policy filters out promising branches that directly lead to new code coverage. A promising branch may indirectly lead to new code coverage after several generations of

[2]Edge coverage measured by SanitizerCoverage tool.

Table 3.2: Code Coverage w/ Continuous Exploration

Program	**Total Cov.**	**BR**	
		Coverage	**Time-to-Term.**
objdump	82442	6252(7.58%)	2.68h
size	57871	3777(6.53%)	2.48h
nm-new	58378	3996(6.85%)	4.33h
readelf	31622	4394(13.90%)	**0.52h**

testcases that are derived from a testcase traversing this branch. Ideally, a good branch-flipping policy would recognize this kind of branch and make continuous progress by iteratively processing newly generated testcases. Therefore, we would like to see how well a CE engine performs when it continuously processes newly generated testcases. Table 3.2 presents the results of this continuous exploration under the BR policy. We can see that the CE engine terminated within five hours[3] for all four programs, because it exhausted its attempts to flip all available branches in all initial inputs and generated testcases. Moreover, the final code coverage only covers 6.53% to 13.90% of the total coverage[2].

Table 3.3: Quality of Generated Testcases

Program	**BR**	
	New-cov Testcases	**Total Testcases**
objdump	231(10.83%)	2,132
size	387(26.51%)	1,460
nm-new	493(45.23%)	1,090
readelf	873(27.25%)	3,204

(3) Branches selected by the BR policy are of low quality. As described above, the BR policy uses branch bigrams to select promising branches to flip. We would expect that most of these selected branches could lead to new code coverage. Table 3.3 illustrates our

[3]All experiments conducted in this paper are measured in terms of wall time.

findings on the quality of the new inputs generated from these selected branches. In fact, on average, only 27.46% of the generated inputs from the branches selected by the BR policy can lead to new coverage. In other words, the majority (72.54%) of the flipping and solving efforts do not immediately translate into code coverage gain. One major reason why *BR policy* cannot select high-quality branches is that the branch-flipping decision is only made based on the testcases that have been previously processed and the current testcase that is processed up to this point. It does not have a chance to examine the remaining execution of the current testcase or the remaining testcases to make a globally optimal decision.

Table 3.4: Path Divergence Rate

Program	**PD Count**	**Total Solving (excluding unsat)**	**PD Rate**
objdump	4628	16926	27.34%
size	5304	10728	49.44%
nm-new	4668	16975	27.50%
readelf	1551	11614	13.35%
Overall	**16151**	**56243**	**28.72%**

(4) Path divergence (PD) rate of concolic execution is exceedingly high that many constraint solving efforts go wasted. We observe that oftentimes, a generated testcase does not traverse the intended unvisited path. This problem is referred to as path divergence problem [28]. Table 3.4 lists our findings with respect to path divergence. We can see that path divergence is very common (as high as almost 50% for `size`, and on average 28.72%). We also observe that the path divergence issue is program-specific and branch-specific. Many branches do not have path divergence at all, while other branches constantly lead to path divergence. Unfortunately, current branch-flipping

policies do not take this into account, leading to the low performance of CE.

Based on the observations, we are motivated to design a new concolic execution scheme that can overcome the aforementioned limitations for more efficient testing.

3.2 Methodology

In this project, we introduce Marco, a novel stochastic and asynchronous concolic explorer. Specifically, to tackle the first two limitations, unlike SymSan, Marco keeps all path constraints from a unique path prefix and incorporates extra information, including calling context and branch direction, into branch definition to retain more meaningful branches. To address the third limitation, our system implements a reachability-guided branch scheduler that can accurately assess the potential of finding new code coverage for every branch. The scheduler then conducts asynchronous solving to make sure our decisions are globally optimal. Furthermore, to overcome the PD problem, Marco models the PD rate for each branch and takes it into consideration when making scheduling decisions.

3.2.1 Overview

As shown in Figure 3.1, Marco comprises three major components: 1) the asynchronous concolic execution engine, 2) the CSTG constructor, and 3) the reachability-guided branch scheduler.

At the beginning of the testing process, the asynchronous concolic execution engine receives an initial seed input and a binary program as input. It then performs concrete and symbolic execution simultaneously, without any constraint solving, to collect concolic traces. These traces comprise symbolic branches encountered, along with the path constraint information needed for branch flipping.

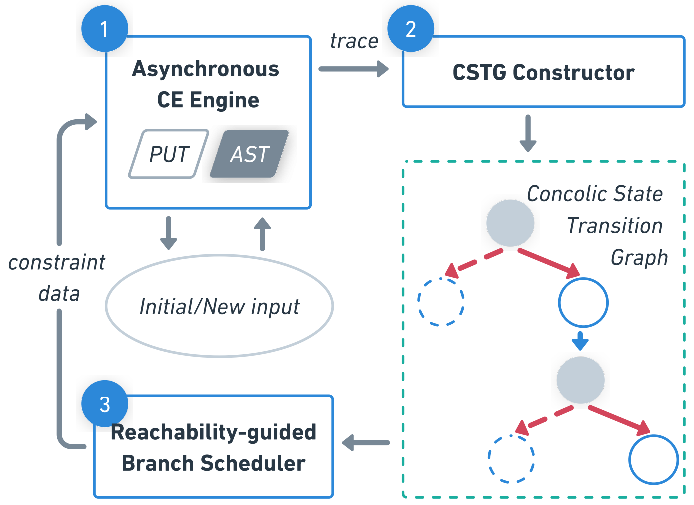


Figure 3.1: Marco Overview

The resulting trace is then passed to the CSTG constructor, which incrementally constructs a CSTG using branch points and branches as nodes, and branch point-to-branch transitions, as well as branch-to-branch point transitions as edges. The reachability-guided state scheduler assesses the potential of each node in the CSTG, calculates a reachability score for each node, and ranks them based on their scores. The highest-ranked node has the greatest potential to lead to new code coverage. The asynchronous CE engine will be invoked to solve a path constraint from the top-ranked node for new testcase generation. Marco then executes the testcase to collect traces and repeat the process.

3.2.2 A Running Example

To better explain our design, we will use an example program from [15], illustrated in Listing 3.1. The program takes two symbolic inputs, `x` and `y`, as input parameters for the function `testme()`. This function contains two symbolic branches located at Line 7 and 8 respectively. The directions taken at these two branches depend on the values of the symbolic inputs.

Listing 3.1: A Running Example

```
 1  int twice (int v) {
 2      return 2*v;
 3  }
 4
 5  void testme (int x, int y) {
 6      z = twice (y);
 7      if (z == x) {
 8          if (x > y+10) { ERROR; }
 9      }
10  }
11
12  int main() {
13      x = sym_input();
14      y = sym_input();
15      testme(x, y);
16      return 0;
17  }
```

3.2.3 Asynchronous Concolic Execution Engine

Unlike synchronous CE engines [75, 51, 52, 18] that perform symbolic tracing and branch flipping simultaneously, Marco takes an asynchronous approach. Specifically, it decouples branch flipping logic (which includes path condition collection and new testcase generation) from the symbolic tracing logic and defers it until after all branch points uncovered are assessed, and a global optimal branch choice is made. It is worth mentioning that although some prior works (e.g., SAGE and CREST) collect execution traces and then replay them offline for branch-flipping, the branch selection is made while processing the current trace. In other words, their branch-flipping policy adopts only a local view as compared to Marco, which will evaluate all branches to make a global optimal selection.

Specifically, the asynchronous CE engine alternates between two modes: 1) the symbolic tracing mode, where it executes the target program with existing testcases to collect data for educated branch prioritization, and 2) the path exploration mode, where it flips a selected branch to find a new path.

Symbolic Tracing Mode

In this mode, the CE engine takes one Program Under Test (PUT) and one testcase as input and produces a concolic path and an AST table. Since our implementation is based on SymSan [18], the AST table stores all the necessary information for reconstructing symbolic expressions.

A concolic path consists of a list of symbolic branches that follow the execution path and some auxiliary information. Below are related definitions:

Branch Point. A branch point *bp* is defined as:

$$bp = (addr, ctx) \ , \tag{3.1}$$

where *addr* is the address of the branching instruction, and *ctx* represents the calling context, which is calculated as a hash of all the call sites on the call stack. The context-sensitive definition of branch point allows Marco to differentiate a branching instruction under different calling contexts and characterize program exploration status more accurately.

For the running example, we have two branch points {L7, main → testme} and {L8, main → testme} at Line 7 and 8 respectively. For brevity, we refer to them as L7 and L8 in the following discussion.

Branch. A branch *brc* is defined as:

$$brc = (bp, dir) \ , \tag{3.2}$$

where *bp* is a branch point defined by Definition (3.1), and *dir* is the direction taken from the branch point. Each branch point has two branches. We use T and F to denote "then" and "else" branches respectively. In the running example, we have four branches denoted as L7T, L7F, L8T, and L8F.

For each symbolic branch, we need to collect essential information about its path constraints. In traditional symbolic execution, the path constraints include all preceding symbolic branches. However, this strategy is often overly strict: generating a new input that follows the exact same path and visits the untaken branch is often impossible [23]. However, there may exist a new input that follows a slightly different path and successfully visits the desirable branch. EXE [14] presents constraint independence optimization which divides path constraints into subsets which are dependent on disjoint sets of input bytes to solve them separately. This idea is then adopted by QSYM [75] and SymSan [18] for concolic execution. Specifically, when negating a branch, SymSan includes any preceding branch that shares data-flow dependencies with the current branch or another preceding branch already included. The resulting set of branches are referred to as nested branches in SymSan. Since we perform concolic execution asynchronously, we prefer not to collect branch constraints right away. Instead, we just record their nested branches.

Nested Branch Set. We define Nested Branch Set NBS for a branch brc as a set of branches in a recursive manner: if a branch brc_i has a data-flow dependency with the target branch brc, then $brc_i \in NBS(brc)$; and if $\exists brc_j \in NBC(brc)$ and brc_i has a data-flow dependency with brc_j, then $brc_i \in NBS(brc)$.

Concolic Path. A Concolic Path CP is defined as a list of 2-tuples:

$$\begin{aligned} CP = & [(brc_0, NBS(brc_0)), (brc_1, NBS(brc_1)), \\ & ..., (brc_n, NBS(brc_n))] \ , \end{aligned} \tag{3.3}$$

where brc_i is the i-th symbolic branch encountered in the execution trace, and $NBS(brc_i)$ is the nested branch set of brc_i.

For the running example, there are three unique concolic paths including: {(L7F,∅)}, {(L7T, ∅), (L8F, {L7T})}, {(L7T, ∅), (L8T, {L7T})}.

Loop Pruning Optimization. Furthermore, we employ optimization to speed up the concolic path collection. Real-world programs often have many loops. Symbolic branches in loops will repeatedly appear in the concolic paths. It takes time to collect their nested branch sets, even though it is much faster than collecting the nested branch constraints. It is also unlikely to iterate through all these sets in order to generate new testcases in the later stage. Therefore, we decide to prune the nested branch sets early on. More specifically, during the execution, we trace the visit count of each encountered symbolic branch. For a branch whose visit count does not evaluate to the power of 2, we do not generate its NBS. In other words, its NBS is $\emptyset$.

In summary, in symbolic tracing mode, the CE engine traces all the testcases in the queue to collect the concolic paths and AST tables. Then it switches into path exploration mode.

Path Exploration Mode

In this mode, the CE engine invokes the reachability-guided branch scheduler (discussed in subsection 3.2.5) to find a global optimal branch choice. With the constraint data of the chosen branch, the CE engine will assemble the path constraint set for traversing this branch and solve it to generate a new testcase.

The constraint data of the chosen branch consists of 1) brc_n, the chosen branch; 2) NBS_n, nested branch set of brc_n; and 3) the AST table of the execution where the chosen branch is encountered.

We start by initializing the PC, the path constraint set as $\emptyset$. Then we query the AST table for the branch predicate of brc_n and add it into PC. If NBS_n is not empty, we query the AST table for each branch in NBS_n for their branch predicates and add them into PC. Then we reuse the solving strategy proposed in QSYM [75]. Specifically, if PC is not satisfiable, we resort to optimistic solving, which will only solve the branch predicate of the target branch and disregard any predicates collected from NBS_n. If optimistic solving is not viable either, the branch scheduler will be prompted again to generate another set of constraint data until a new seed is generated.

3.2.4 CSTG Constructor

After collecting concolic paths in the asynchronous CE engine, we seek to construct CSTG, which further enables the reachability-guided branch scheduler. The graph is a directed heterogeneous graph defined as follows.

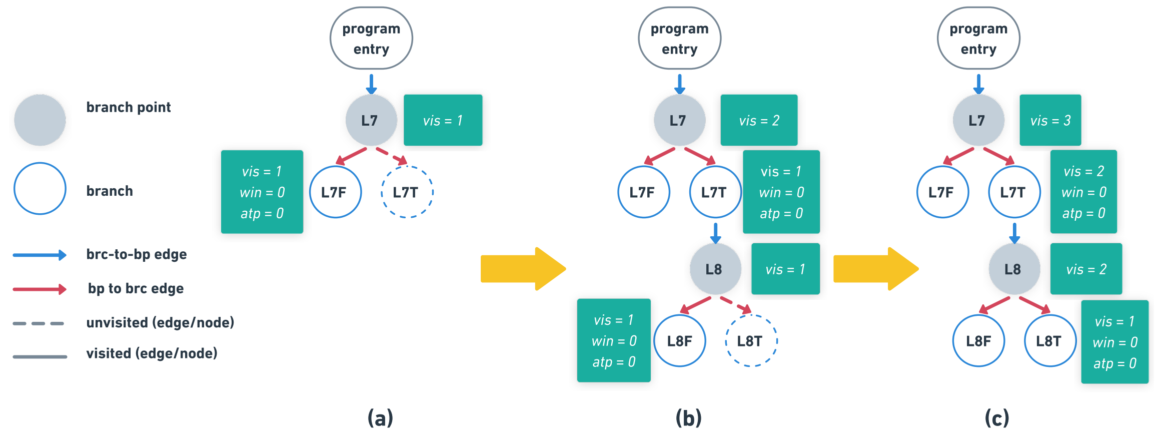


Figure 3.2: CSTG Construction of Example Program

Concolic State Transition Graph (CSTG). A CSTG is defined over a set of CPs as:

$$G = (V, E) \ , \tag{3.4}$$

where V is a set of branch points and branches, and E is a set of vertex transitions. In addition, a virtual root vertex denotes the program entry point. Each vertex $v \in V$ is associated with a set of attributes including: 1) $v.vis$: the number of concrete visits at v;

2) $v.atp$: the number of branch flipping attempts at v; 3) $v.win$: the number of successful branch flipping attempts at v; and 4) $v.pcq$: the queue of path constraint sets for generating new testcase that potentially will traverse v. Note that attributes 2) to 4) only apply to vertices representing branches instead of branch points. Each edge $e \in E$ is associated with a concrete visit count $e.vis$.

Algorithm 1 illustrates how MARCO constructs the CSTG incrementally. Initially, the graph contains one root node R as the program entry, and the edge set is empty. The procedure takes as input the graph G and a new concolic path CP as defined in (3.3). The algorithm then performs a preprocessing step to retain a set of visited branches along with their concrete path prefixes and remove from CP any branch that is visited through the observed path prefix. When a new branch (according to the Definition 3.2) is observed for the first time, we insert three nodes (one for its branch point *bp* and two for the taken and untaken branches brc_0 and brc_1, and three edges into the current graph (Ln.6-10). If the branch has been observed and thus has already existed in the graph, we simply retrieve the existing three nodes (one branch point and two branches) from the graph (Ln.12-14). Then, the algorithm calls $updateNode$ to update the nodes' attributes defined in subsection 3.2.4 as needed (Ln.16-17). Specifically, for *bp* and brc_0, we update visit count $v.vis$. If brc_0 matches $lastChosen$, we update its win count $v.win$. For brc_1, we update the path constraint queue $v.pcq$ to include the new path constraint collected from the current execution path to potentially force execution down brc_1. Further, if the currently taken branch brc_0 is equal to the node picked by the last scheduling round to perform branch flipping on $lastChosen$, it means the testcase generated from the last round (i.e., the current testcase) indeed traverses the selected branch. In this case, it will increase the current branch's win count by one.

In our running example, we consider three concolic paths {(L7F,∅)}, {(L7T, ∅), (L8F, {L7T})}, {(L7T, ∅), (L8T, {L7T})}. Marco gradually constructs CSTG of the example program as illustrated in Figure 3.2 (a), (b), and (c).

Initially, the graph is empty with a root node, which denotes the program entry. For the first concolic path {Ł7F}, since node L7F is a new node, we add three nodes, i.e. L7, L7F, and L7T, and three edges, i.e. (R, L7), (L7, L7F), (L7, L7T), into the graph. We increase the visit counts of node L7, L7F, edge (R, L7), (L7, L7F) from 0 to 1. After processing this concolic path, the graph is presented as Figure 3.2(a). Similarly, Marco then takes the second concolic path {L7T, L8F} as input. As the first branch L7T already exists in the graph, we increase the visit counts of node L7, L7T and edge (L7, L7T) by 1. However, the second branch L8F is not an existing node in the graph. Therefore, we insert three nodes, i.e. L8, L8F and L8T, and three edges (L7T, L8), (L8, L8F), (L8, L8F) into the graph. And we update the visit counts accordingly. After processing this concolic path, the graph is shown as Figure 3.2(b). Moreover, we make similar changes as discussed above for the concolic path {L7T, L8T}. Hence, after processing the three concolic paths, Figure 3.2(c) is the final CSTG, which will then be used for branch scheduling.

3.2.5 Reachability-guided Branch Scheduler

Reachability-based branch scheduler aims to find the branch that bears the highest potential for new code coverage and gives the path constraint data of the top-ranked node to the asynchronous CE engine for input generation.

Essentially, we assess the potential of a branch by the number of reachable yet unvisited branches deeper in the execution paths that traverse the branch. To do so, we generate a reward score for each untaken branch, consider a concolic trace as Markov Chain and compute a transition probability to take the path divergence (PD) rate into consideration, and further accumulate the rewards up to calculate a node reachability score that estimates the potential of every branch in the CSTG, in order to pick the best one for further exploration. However, one technical challenge is that the transition probability between nodes and the estimated reward of nodes are unknown at the beginning of the testing. Here in this section, we discuss how MARCO tackles this challenge.

Edge Transition Probability Calculation. The transition probability of an edge captures how likely an execution will branch to the end node from the start node. As discussed in subsection 3.2.4, there are two types of edges in MARCO: 1) the *bp*-to-*brc* edges and 2) the *brc*-to-*bp* edges. And we calculate their transition probability differently.

The transition probability of a *bp*-to-*brc* edge is associated with the success rate of generating a testcase traversing *brc* by solving a path constraint associated with *brc*, i.e., the opposite of the path divergence rate of this edge. The total solving attempt count at *brc* is *brc.atp*, and the success count is *brc.win*. Intuitively, the estimated transition probability is $brc.win/brc.atp$. The estimation is relatively accurate when the attempt count at *brc* is high enough. But this assumption does not always hold, especially at the early stage of testing and for the less-explored code regions. For better estimation of the transition probability and to balance between exploration and exploitation, we resort to *Thompson Sampling* (TS) [55]. The key idea of TS is to sample the success rate of an action over the *Beta Distribution* defined by the outcomes of the past trials. The Beta distribution is defined by two positive parameters α, denoting the win count, and β, denoting the loss count. It becomes more and more concentrated around the empirical success rate $\alpha/(\alpha+\beta)$ as the number of total trials $(\alpha+\beta)$ grows. For a *bp*-to-*brc* edge, α is *brc.win* and β is (*brc.atp* - *brc.win*). The transition probability of a *bp*-to-*brc* edge is calculated by Equation 3.5.

$$p(bp, brc) = \tau(y.win, y.atp - y.win) \ , \tag{3.5}$$

where τ denotes Thompson Sampling. Note that, each *bp* has two *bp*-to-*brc* edges, each leading to one viable branch. We normalize the transition probabilities of these two edges to ensure they sum up to one.

The *brc*-to-*bp* edge transition differs from that of *bp*-to-*brc* edge in the following aspects: 1) CE engine cannot actively steer execution from a *brc* node to one particular *bp* node through path constraint solving; 2) one *bp* node has two outgoing edges, each leading to one viable branch, while a *brc* node potentially has zero to multiple succeeding *bp* nodes; 3) each *brc* has only one parent node which is its branch point while each *bp* can potentially have multiple preceding *brc* nodes. Therefore, Equation 3.5 does not apply to computing the transition probability for a *brc*-to-*bp* edge.

The transition probability of an edge leading from *brc* to *bp* can be estimated as the success rate of transitioning from *brc* to *bp*. In this case, each visit at edge $e_{brc,bp}$ is considered a win. The total amount of trials for visiting this edge includes the visit count and attempt count at *brc*. In other words, each time *brc* is visited or attempted, but the subsequent execution does not lead from *brc* to *bp* is considered a losing attempt. Similarly, when the total trial count is low, the accuracy of the estimation can be low. Again, we leverage TS for the transition probability computation for *brc*-to-*bp* edge with α being $e_{brc,bp}.vis$ and β being $brc.vis + brc.atp - e_{brc,bp}.vis$ as shown in Equation 3.6.

$$p(brc, bp) = \tau(e_{brc,bp}.vis, brc.vis + brc.atp - e_{brc,bp}.vis) \tag{3.6}$$

Again, we normalize the transition probabilities of the edges leading from the same *brc* node and ensure they sum up to one.

In summary, we leverage Thompson Sampling to dynamically optimize the estimation of transition probabilities of the two types of edges in CSTG which allows us to balance between exploration and exploitation.

Node Reachability Score Calculation. We compute a node reachability score for each node in CSTG, which indicates the nodes' potential for leading to new code coverage in future testing. We then use it to guide the path prioritization in concolic execution.

The reachability score of the *brc* node should capture two aspects: 1) potential new code coverage reachable from *brc* and 2) the difficulty of generating a new testcase that visits the *brc* node. We measure a node's reachable new code coverage as a numerical value denoted as *Coverage Score* and compute it by Equation 3.7 (for leaf nodes) and Equation 3.8 (for interior nodes).

$$N.score = \tau(0, N.vis + N.atp) \tag{3.7}$$

The coverage score of a leaf node is calculated by Equation 3.7. In particular, for an unvisited leaf node, the coverage score is affected by the number of solving attempts devoted to it. When the number of attempts grows but the node remains unvisited, it means that this node could be too hard to reach. Therefore, our limited resources are better off being relocated to other nodes. For a visited leaf node, apart from the attempt count, the visit count also affects its coverage score. Each visit to a node without steering the execution into a deeper state is considered a failed attempt. Consequently, the exploration should try to avoid such nodes. As the visit count and the attempt count grow, the coverage score of a visited leaf node will decrease.

We compute an interior node's coverage score by Equation 3.8:

$$N.score = \sum_{i=0}^{n} p(N, M_i) * M_i.score \ , \tag{3.8}$$

where M_i ($i \in [0, n]$) denotes a child node of N. Essentially, the coverage score of an interior node is affected by two major factors. First, a node that is adjacent to a large number of unvisited nodes is in general of higher potential than a node that has only a small number of unvisited neighbor nodes. Hence, the number of a node's unvisited successors in CSTG can strongly indicate its potential for new code coverage. Second, given any path in a program, the number of inputs that go through the child node is strictly less than or equal to the number of inputs that go through the parent node. Subsequently, the distance between a node and its unvisited successors also plays an essential role in estimating the potential for new coverage.

The coverage score of each node in CSTG is updated periodically to reflect the most recent changes. Apparently, CSTG can be a cyclic graph which imposes a challenge for efficiently updating each one of the nodes for an updated coverage score. We periodically perform the whole graph score updates by first performing a post-order traversal over the graph to extract all the nodes into an ordered list. Then we traverse the list to update each node's coverage score. The reachability score for each *brc* node in the graph is computed by Equation 3.9.

$$brc.rs = p(bp, brc) * brc.score \tag{3.9}$$

Essentially, for a branch *brc* with a high path divergence rate (i.e. low in-edge transition probability), it is hard to generate a testcase traversing that branch and it renders the coverage score in vain. We then prioritize the branch nodes in CSTG for scheduling by their reachability score.

Branch Prioritization. After calculating reachability scores, the node (i.e., branch) with a better potential of reaching new code will have a higher score and be promoted in the scheduling. The path constraint associated with this top-ranked node is sent to the Path Constraint Solver for new input generation. In case of an unsatisfiable path constraint, the scheduler is prompted again until a new testcase is successfully generated and the testing continues.

3.3 Evaluation

We evaluate the efficacy of our proposed approach by answering the following research questions:

- **RQ1: Effectiveness of end-to-end concolic execution.** Can our model improve the performance of end-to-end concolic execution?
- **RQ2: Effectiveness of design choices.** What are the unique contributions of each

design choice in MARCO?

- **RQ3: Vulnerability detection.** Can MARCO be more effective when detecting vulnerabilities?

3.3.1 Evaluation Plan

To better answer the aforementioned research questions, we use the following configurations:

- **SymSan [18]**, the SOTA CE engine, which adopts the traditional synchronous solving (i.e., the constraint solving is conducted at the time when the branch is encountered) with the same native branch flipping policy as QSYM [75]. This baseline is to directly compare with MARCO.
- **SymSan-pp**, a variant of SymSan that adopts a *PP* branch-flipping policy, where each branch is defined by its path prefix. The testcases are executed in a First-In-First-Out (FIFO) order, with all branches flipped by the visit order. This baseline is to show that simply selecting more branches to flip will not improve CE performance.
- **MARCO-rdm**, a variant of MARCO that defines each branch by its path prefix and picks a random branch from the last visited program path to flip and generate a new testcase. This configuration is to demonstrate the effectiveness of our branch scheduling strategy.
- **MARCO-cfg**, a configuration that applies the CFG-directed searching algorithm of CREST [12] on our dynamically generated CSTG. The assessment of each branch is determined by the branch count between the branch itself and the nearest unvisited branch, as well as the flipping attempt count. This configuration is used to show the effectiveness of our branch scheduling strategy.

- **Marco-uv**, a variant of Marco which only allows the scheduler to pick from unvisited nodes. This configuration is used to show the necessity of flipping the visited branches.
- **Marco-MC**, a variant of Marco with Markov Chain modeling but no Thompson Sampling. This configuration evaluates the importance of Thompson Sampling in Marco.
- **Marco**, our full-fledged system.

To answer RQ1, we compare the code coverage metric of the full-fledged model against SymSan. The experiment is conducted on real-world programs listed in Table 3.5. For RQ2, we compare the code coverage per path constraint solving for all the configurations listed to showcase the effectiveness of each design choice. Finally, to answer RQ3, we run both Marco and SymSan on the UniFuzz dataset, and compare the number of unique bugs found by them.

Table 3.5: Details of Real-world Applications Evaluated

No.	Program	Version	No.	Program	Version
1	nm-new	2.33.1	16	libtiff	2e822691
2	readelf	2.33.1	17	tcpdump	4.8.1 + libpcap 1.8.1
3	objdump	2.33.1	18	flvmeta	1.2.1
4	size	2.33.1	19	tiffsplit	libtiff 3.9.7
5	libpng	1.2.56	20	jhead	3.00
6	libxml2	2.9.2	21	imginfo	jasper 2.0.12
7	file	5.42	22	jq	1.5
8	vorbis	c1c2831f	23	lame	lame 3.99.5
9	curl	2481dbe	24	wav2swf	swftools 0.9.2
10	lcms	430f916	25	mujs	1.0.2
11	woff2	9476664	26	sqlite3	3.8.9
12	libjpeg-turbo	b0971e47	27	mp3gain	1.5.2-r2
13	sqlite3	c78cbf2	28	mp42aac	Bento3 1.5.1-628
14	tcpdump	4.99.1	29	cflow	1.6
15	freetype	cd02d359a	30	infotocap	ncurses 6.1

Dataset. We collect a dataset consisting of 30 popular real-world programs as shown in Table 3.5, as well as 71 programs from the DARPA Cyber Grand Challenge (CGC) dataset. To answer RQ1 and RQ2, we conduct experiments on the CGC programs, as well as programs No. 1 to 16 (Binutils and Fuzzbench [2] binaries). To answer RQ3, we further leverage programs No. 17 to 30, which are the subset of the MARCO compatible Unibench dataset proposed in UniFuzz [42]. We configure the experiment to align with the original setup in UniFuzz, including each program's execution option and the initial seed corpus used.

Experiment Setup. All evaluation was done on a workstation with two-socket, 48-core, 96-thread Intel Xeon Platinum 8168 processors. The workstation has 768G memory. The operating system is Ubuntu 18.04 with kernel 5.4.0.

3.3.2 RQ1: Effectiveness of MARCO

To demonstrate the effectiveness of MARCO in terms of exploring new code coverage, we measure the edge coverage during testing and compare our full-fledged model with SymSan. Each configuration is repeated 10 times to reduce randomness.

We collect the edge coverage at the end of each 24h trial and measure the coverage improvement ratio of MARCO over the baseline SymSan. For each program, we further investigate the relative code coverage between SymSan and MARCO with the formula proposed in QSYM [75]. For code coverage A (MARCO) and B (SymSan), we can quantify the coverage difference using:

$$d(A,B) = \begin{cases} \frac{|A-B|-|B-A|}{(A\cup B)-(A\cap B)} & \text{if A} \neq \text{B} \\ \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

With the coverage difference score $d(A, B)$, we can infer the number of unique edges that A covered, out of the total edge coverage that either A or B can uniquely explore. A positive score means A (Marco) finds more unique coverage than B (SymSan). The value will be 1 if A (Marco) not only finds more coverage than B (SymSan) but also covers all edge coverage explored by B (SymSan).

In our experiment, we evaluate the performance of Marco on 16 real-world programs (programs No.1 to 16 in Table 3.5). On average, Marco is able to cover 13.03% more edges, with a maximum improvement on `readelf` for 88.56%. This indicates the effectiveness of our approach in improving the effectiveness of concolic testing for real-world programs. Out of the 16 tested programs, Marco finds more edge coverage than SymSan on 11 programs (68.75%). Moreover, Marco dominates the coverage findings on three targets (`file`, `lcms`, and `sqlite3`), where it also covers all the edges found by SymSan.

We further evaluate the performance of Marco on 87 programs (71 DARPA CGC binaries + 16 real-world programs) and compute the coverage difference score between Marco and SymSan. Inspired by [75], we visualize the results in Figure 3.3: the blue color indicates that Marco finds more edge coverage than SymSan, and the red color indicates that SymSan finds more. The results indicate that Marco can do better than SymSan on 49 programs, and worse on 17 programs.

Further investigation (Table 3.6) shows that SymSan would terminate within 5

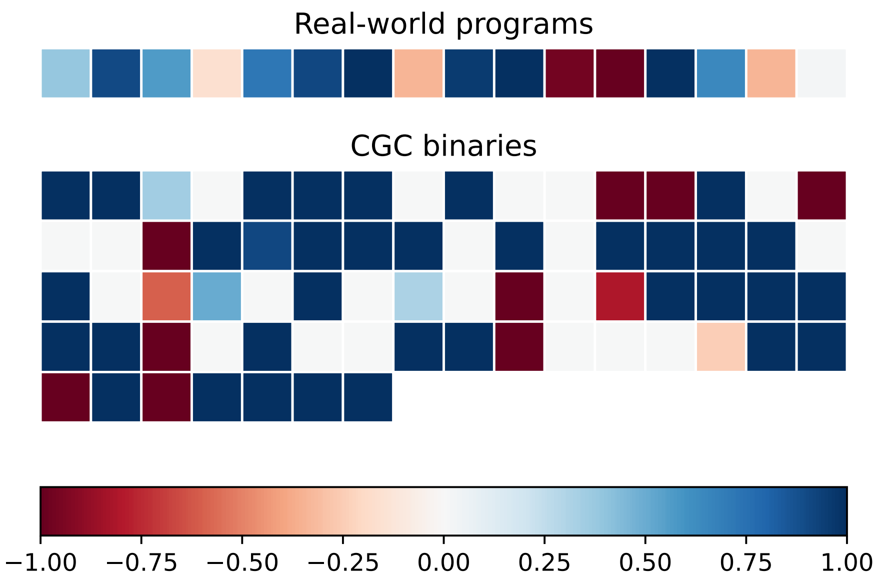


Figure 3.3: Coverage Difference Score of Real-world Programs and CGC Binaries

hours on 75% (12/16) of the real-world programs, even though there still exist many edges unexplored. This is due to the overly strict branch definition and ill-advised branch-flipping strategy adopted in SymSan.

Table 3.6: Average Termination Time for SymSan

Program	**Term. Time(h)**	**Program**	**Term. Time(h)**
nm-new	4.33±1.40	curl	0.10±0.03
readelf	0.52±0.05	lcms	0.06±0.01
objdump	2.68±0.54	woff2	>24
size	2.48±0.31	libjpeg-turbo	15.43±2.90
libpng	0.05±0.01	sqlite3	0.02±0.00
libxml2	1.57±0.16	tcpdump	0.75±0.81
file	0.27±0.01	freetype	10.36±0.79
vorbis	8.66±0.79	libtiff	1.93±0.26

To evaluate the scalability of Marco, we investigate the graph size growth and the memory cost for each of the 16 real-world programs during testing. The results show that the number of nodes in CSTG grows sub-linearly during the 24h trials. At the end of each trial, the minimum, maximum, average, and median values of the node counts are 0.26k, 118.98k, 13.89k, and 3.10k correspondingly. We then record the amount of memory taken by storing the AST tables and see that the disk usage grows linearly. At the end of the trial, the minimum, maximum, average, and median values of memory usage are 7.88G, 63.19G, 23.90G, and 20.66G.

3.3.3 RQ2: Effectiveness of Design Choices

As discussed earlier, SymSan terminates very early in 75% of the tested programs, meaning only a limited number of solving attempts have been made. In RQ2, we allocate the same amount of solving attempts for the other configurations and assess their new code coverage. By doing so, we can demonstrate the effectiveness of our design choices in improving the branch prioritization scheme.

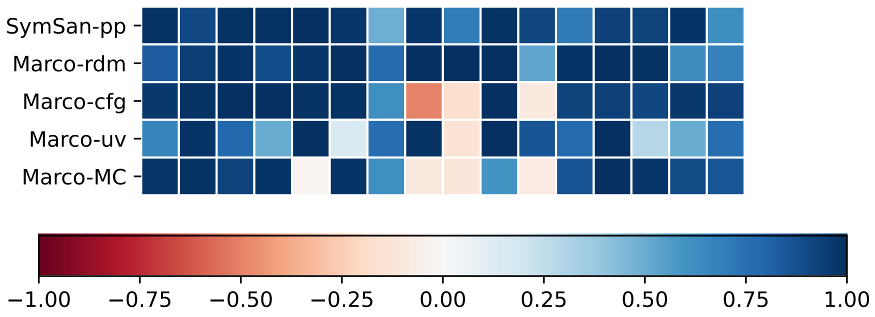


Figure 3.4: Coverage Difference Score Within Solving Budget

We look into the edge coverage for SymSan-pp, Marco-rdm, Marco-cfg, Marco-uv, and Marco-MC with the 16 real-world programs and compute the coverage difference scores compared with Marco as defined in Equation 3.10. The experimental results are displayed in Figure 3.4. Each row depicts the coverage difference score of A (Marco) and

B (the baseline labeled to the left of the row). The blue color indicates that MARCO finds more edge coverage than the corresponding baseline, and the red color suggests otherwise. The results exhibit a few major conclusions:

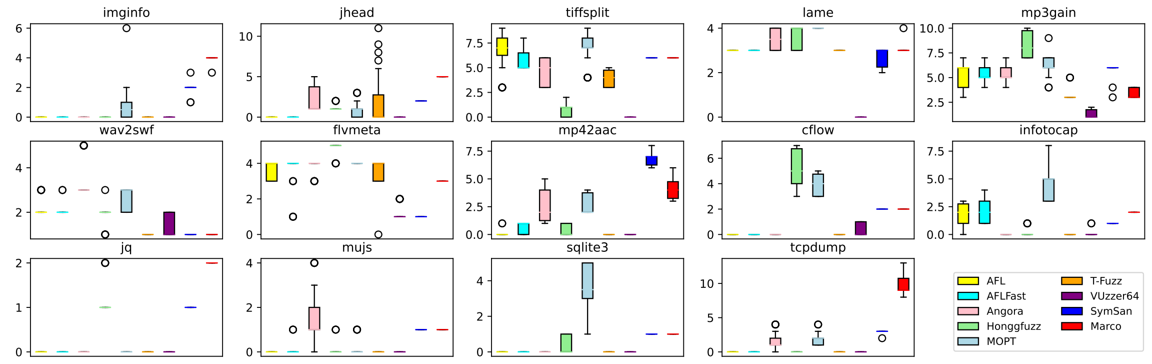


Figure 3.5: Number of Unique Bugs Detected

Firstly, MARCO outperforms SymSan-pp and MARCO-rdm on all 16 programs. On average, MARCO covers 55.99% and 86.64% more code than SymSan-pp and MARCO-rdm. This result explicitly demonstrates that the novel branch prioritization strategy in MARCO, other than a simple FIFO or random selection, is extremely useful when it comes to code exploration.

Secondly, MARCO outperforms MARCO-cfg on 13 out of 16 tested programs. For the other three programs (`vorbis`, `curl`, and `woff2`) where MARCO-cfg finds more coverage, the differences are slight (<1%). MARCO is able to find 83.92% more code coverage than MARCO-cfg. This result indicates that our branch flipping strategy is better than the CFG-directed approach.

Thirdly, compared with MARCO-uv, MARCO manages to find more coverage on 15 programs out of 16, with only one exception `curl`. On average, Marco finds 29.22% more code coverage than Marco-uv. This shows that it is indeed a good strategy to deem both visited and unvisited nodes as candidates for path constraint solving.

Lastly, Marco outperforms Marco-MC on 12 out of 16 tested programs with an average coverage improvement ratio of 57.21%, indicating that modeling edge transition and reachability score with Thompson Sampling to balance between exploration and exploitation is crucial to making effective branch prioritization decisions.

We further evaluate the significance of difference comparing Marco and the other configurations in Figure 3.4 across the 16 tested programs on their code coverage findings using p-values from the `Mann-Whitney U-Test`. We use p-value < 0.05 as the threshold for statistical significance. We observed p-values above 0.05 in only two programs, `curl` (0.07) and `woff2` (0.48), when comparing Marco and Marco-cfg. For the rest of the results, the p-values are below 0.001 for majority of the cases. The result suggests significant difference between Marco and the other configurations.

3.3.4 RQ3: Vulnerability Detection

Lastly, we showcase the capability of vulnerability detection for Marco by using the UniFuzz dataset, which consists of 14 programs. Specifically, we run both Marco and SymSan 5 times for 24 hours and compare the average number of unique bugs detected. According to the results, Marco is able to find 33.52% more bugs (47.8 v.s. 35.8) than SymSan. Among them, Marco can uniquely identify 2.41 times the bug count of SymSan (20.5 v.s. 8.5). More concretely, Marco finds more unique bugs than Symsan on 7 programs, less on 2, and the same amount on 5. These numbers show that Marco has its unique advantages when finding vulnerabilities compared with state-of-the-art CE engines.

We further cross-check our results with that reported in UniFuzz paper[4] in UniFuzz [42] paper for 7 fuzzers (AFL [76], AFLFast [8], Angora [20], HonggFuzz [73], MOPT [45], T-Fuzz [50] as well as VUzzer64 [53]) in 24h. We draw the box plot of all 8 baselines in Figure 3.5. According to the result, Marco is able to find more unique bugs in 12h than any of the 7 fuzzers can find in the 24h trial on 5 (`imginfo`, `jhead`, `mp42aac`, `jq` and `tcpdump`) out of the 14 tested programs. Marco ranks the second place on `mujs` and `sqlite3`, second to Angora and MOPT respectively. We further explore the statistical rankings among Marco and the 7 fuzzers by their average unique bug detection counts for each program. Marco beats 6 fuzzers and is second to MOPT only. This demonstrates that Marco can find bugs very efficiently.

[4]We contacted the authors for the original experiment data but didn't get a response by the time of submission. Therefore we repopulated the bug detection result based on the data reported in their supplementary result: `https://github.com/unifuzz/supplementary_results/blob/master/UNIFUZZ_Supplementary_Paper.pdf`

Algorithm 1 The CSTG Construction Algorithm

```
1: $lastChosen \leftarrow$ state chosen from last scheduling round
2: procedure GraphUpdate(G, CP, lastChosen)
3:     lastnode = R
4:     while ! CP.empty() do
5:         (addr, ctx, dir) = CP.pop()
6:         if !G.findNode(addr, ctx) then
7:             $bp$ = G.newNode(addr, ctx)
8:             $brc_0$ = G.newNode(addr, ctx, dir)
9:             $brc_1$ = G.newNode(addr, ctx, !dir)
10:            G.newEdge(<lastnode, $bp$>,<$bp, dp_0$>,<$bp, dp_1$>)
11:        else
12:            $bp$ = G.getNode(addr, ctx)
13:            $brc_0$ = G.getNode(addr, ctx, dir)
14:            $brc_1$ = G.getNode(addr, ctx, !dir)
15:        end if
16:        for $s \in [bp, brc_0, brc_1]$ do
17:            G.updateNode(s, lastChosen)
18:        end for
19:        for $e \in$ [<lastnode, $bp$>,<$bp, brc_0$>] do
20:            $e.vis{+}{+}$
21:        end for
22:        lastnode = $brc_0$
23:    end while
24: end procedure
```

Chapter 4

How Well can LLMs Generate Fuzzing Inputs?

Fuzz testing has emerged as a powerful technique for uncovering security flaws in software systems. During the fuzzing process, we execute the Program Under Test (PUT) with large amount of testcases and monitor its runtime behavior to find vulnerabilities. In particular, greybox fuzzing proves to be very effective. A greybox fuzzer (e.g., AFL [76]) chooses a seed from the seed queue, generates new inputs by performing a set of manually-crafted mutations on it, and then executes the instrumented PUT with these new inputs. If a new input causes new code coverage, it will be kept as a new seed and put in the seed queue.

Recently, Large Language Models (LLMs) have demonstrated its success in many fields, including natural language processing, automatic code generation, vulnerability detection, automatic program repair. A natural question to ask here is "<u>can LLMs help</u>

with fuzzing?". Indeed, some recent research efforts have already incorporated LLMs into fuzzing. Specifically, ChatAFL [46], a protocol fuzzer, queries LLM (`gpt-3.5-turbo`) to extract machine-readable form of the PUT grammar to help fuzzer perform structure-aware mutation.

Apart from the grammar extraction, ChatAFL only uses ChatGPT for limited queries for testcase generation: to diversify the initial corpus before commencing fuzzing campaigns and to escape coverage plateau when the fuzzer counterpart gets stuck. Fuzz4All [71], on the other hand, targets systems taking in programming or formatted inputs and conducts much more frequent LLM queries to generate testcases for testing. As a result, a smaller LLM, `StarCoder`, is selected as the generation LLM for the interest of efficiency. To provide the generation LLM with better context and instruction, a larger and more capable LLM, `gpt-4-0613`, is employed as the distillation LLM. It synthesizes prompts for the generation LLM by summarizing the usage and functionality of the PUT.

Despite these initial successes in adopting LLMs in fuzzing, the research community lacks a good understanding of out-of-the-box LLMs' fuzzing performance in general. That is, without making any customizations or optimizations (such as fine-tuning, in-context learning, distillation, and prompt optimization), how well can an out-of-the-box LLM work for fuzzing? The evaluation of the out-of-the-box LLMs' fuzzing performance can establish a baseline for any followup research in this direction. To the best of our knowledge, this is the first measurement study of this kind.

More specifically, we evaluated two online models (ChatGPT 3.5 and ChatGPT 4.0) and four locally-deployed models (Llama2 and Code Llama chat and completion models), and found answers to the following questions. First of all, should an LLM generate a testcase from scratch or mutate an existing testcase? We observe that all six models produce better-quality testcases by mutating an existing one than generating one from scratch, with respect to more valid and diverse testcases and higher code coverage.

Second, among these models, which model produces best mutations? Mutations are considered good, if they are not only syntactically distinct, but also semantically diverse (i.e., exploring unique program execution space). Our study suggests that the two online models (ChatGPT 3.5 and ChatGPT 4.0) significantly outperform the four local models, and ChatGPT 3.5 is the best despite that ChatGPT 4.0 is more recent.

Third, which model is the most cost-effective, in terms of time and monetary cost? A common belief is that larger online models like ChatGPT are more expensive than those smaller local models, and thus less cost-effective. Our study disapproves it: under the same time or monetary budget, ChatGPT models can achieve much higher code coverage than those local models. Moreover, ChatGPT 3.5 is more cost-effective than ChatGPT 4.0, in terms of both time and monetary cost.

At last, can LLMs completely replace existing manually-crafted mutators? The answer is "No". Our study shows that despite some overlaps, different LLMs and AFL++ tend to explore separate regions in the program execution space.

In this paper, we design three research questions to drive a systematic study on LLMs' fuzzing capabilities, and present our findings. This study across a wide range of models allows us to understand how fuzzing performance can be influenced by model size, family and computational needs and be able to derive insights that remain relevant as newer variants emerge. Based on the study conducted across these models, we provided concrete insights and guidelines for LLM-augmented fuzzing in the discussion section, such as PUT (Program-under-test) selection, effective prompt engineering, and LLM model selection which also disproved the assumptions made in prior works. These insights and guidelines offer a resilient framework that can be adapted to future LLM models and variants, ensuring their relevance even as the technology progresses.

4.1 Systematic Study

1) Subjects: To conduct a systematic study on the capability of LLMs for program input generation, we choose eight representative LLMs as shown in Table 4.1. These LLMs belong with three LLM family, ChatGPT, Llama 2 and Starcoder. The three ChatGPT models are of larger size (175 billion parameters and above) and closed source, therefore can only be accessed remotely via API calls. The Llama 2 and Starcoder models are relatively small compared with ChatGPT models. The Llama 2 models come in three model sizes: 7, 13 and 70 billion parameters. We choose the 7-billion model to fit our GPU infrastructure (NVIDIA Tesla V100 SXM2 16GB). For Starcoder we choose the 7b model for the same reason. To comprehensively evaluate our study subjects, we further include AFL++ [26] (the State-of-the-art greybox fuzzer), and Gramatron [61] (a grammar-aware fuzzer, dubbed as `GRAM`) to provide a perspective on how well LLM mutators perform compared to conventional fuzzers.

2) Benchmark Programs: We conduct experiments on four program, of which three (`jq`, `php`, `mujs`) takes in text-based inputs that adhere to the syntax rules of JSON, PHP, JavaScript

Table 4.1: Subject LLMs

Mode	**Model Endpoint**	**Abbreviation**	**Size**	**Environment**
instruct/chat	gpt-3.5-turbo-1106	GPT3.5	large	remote via API
	gpt-4-1106-preview	GPT4.0		
	gpt-4o	GPT4O		
	CodeLlama-7b-instruct	CL-in	small	local model inference
	llama-2-7b-chat	LL-in		
completion	CodeLlama-7b	CL		
	llama-2-7b	LL		
	starcoder2-7b	STAR		

source code; and one (`objdump`) takes in non-text-based inputs in ELF format. Note that the subject LLMs only accept and produce text and do not support non-text-based inputs or outputs. To query the LLMs to generate ELF format seeds, we first encode the sample ELF input using the command `xxd` into text, then decode the output text from the LLMs with the command `cut -d’ ’ -f2-9 output.txt | xxd -r -p > output.bin` into binary form.

Table 4.2: Benchmarks

Program	**Input Format**	**Syntax Check Command**
jq	json	jq . @@
php	PHP	xmllint –noout @@
mujs	JavaScript	node @@
objdump	ELF	readelf -a @@

For each program, we prepare a set of 20 unique seeds of which 10 are syntactically valid and 10 are of broken syntax. The commands used to validate the testcases’ syntax validity is shown in Table 4.2, where @@ denotes the testcase file name.

3) Experiment Setup: To utilize the ChatGPT models, we have registered for the OpenAI API service. For the four Llama 2 models and one Starcoder model, each trial is pinned on one GPU, the computing instances are allocated using the Google Cloud Platform.

Please note that the OpenAI API service imposes different rate limits for different usage tiers. Our experiment for the three ChatGPT models was conducted using Tier 5, which offers the highest rate limit among all usage tiers[1]. Additionally, the four Llama 2 models and one Starcoder model were run on a VM instance with NVIDIA V100 GPU (NVIDIA Tesla V100 SXM2 16GB) using the Google Cloud Platform, with costs calculated at an hourly rate of \$3.67.

4) Research Questions: Here we explain the purpose for each research question as well as the corresponding experiments.

RQ1: *How to formulate the prompt for an LLM to act as an effective program input generator?* Prompting is a critical step in utilizing LLMs, as it provides context and guidance to the model. However, there are no established empirical guidelines for optimizing prompts for program testing tasks. For this RQ, we conduct experiments involving three prompt designs to find the effective prompt that directs the LLM to generate better program inputs for automated fuzz testing. The information provided in the prompt significantly influences the quality of generated testcases. In particular, depending on whether or not a sample testcase is included in the prompt, the LLM can operate either in *mutation-based* mode or *generative* mode. By specifying the expected format of the program input in the prompt, we can direct the LLM to produce syntactically valid seeds to mitigate the risk of early rejection in the execution path.

RQ2: *Can LLMs produce diverse seed mutations?* For this RQ, we examine coverage efficiency of subject LLMs compared with conventional fuzzers. For this research

[1] `https://platform.openai.com/docs/guides/rate-limits/usage-tiers?context=tier-one`

question, we examine the coverage efficiency of LLM mutators compared with conventional mutators in terms of unique coverage findings, as well as their explored region diversity and sparsity. Our goal is to determine whether LLM mutators can entirely replace conventional greybox fuzzers, and if not, which LLM mutator contributes the most to the fuzzer. To that end, we further investigate the explored-region diversity of LLM-generated mutations to identify which LLM has the most potential for collaborating with a greybox fuzzer.

RQ3: *Are LLMs cost effective for fuzzing?* One practical consideration when utilizing LLMs for program analysis tasks, such as fuzzing, is the associated cost. Is the performance gain achieved by adopting LLM methods justified by the resources expended? In this RQ, we explore the coverage findings relative to the time and monetary costs involved for the tested LLMs, and help the community determine whether or not LLM should be used, and if so which one offers the most cost-effective solution.

4.2 Findings

In this section, we analyze the experiment results and present our findings and answer each research question.

4.2.1 RQ1: Prompt Design

To fully harness the potential of leveraging Large Language Models (LLM) for program input generation or mutation to enhance fuzzing, a crucial step is to synthesize an effective prompt to guide the model. Previous works, such as Fuzz4all and ChatAFL, have demonstrated the efficacy of LLMs in understanding the internal workings of Programs

Under Test (PUTs) and utilizing this understanding to generate superior test cases. In this paper, we aim to explore the potential of LLMs for fuzzing without prior knowledge of the target program's runtime logic.

Table 4.3: Prompt Designs

Type	**Program Input Info**	
	Sample Input	**Format**
LLM_S	✓	✗
LLM_F	✗	✓
LLM_FS	✓	✓

To ensure our evaluation is not biased towards any particular LLM and to generalize our findings, we assess three types of prompts across the eight subject LLMs. In Table 4.3, we present the three prompt designs evaluated. In particular:

- With `LLM_S` prompt, the LLM operates in a mutational-based manner, where a sample input is included in the prompt. The model is instructed to generate variations of this sample input.
- With `LLM_F` prompt, the LLM operates in a generative manner, where the expected program input format is included in the prompt. The model is instructed to generate testcases that adhere to corresponding syntax rules.
- With `LLM_FS` prompt, both a sample input and the input format requirement is included to instruct the LLM. The LLM operates in a mutational-based manner with additional information specifying the expected format.

To determine the optimal prompt design and understand how the inclusion of sample input and/or input format affect the quality of generated testcases, we conduct experiments on the four real-world programs listed in Table 4.2. We start with a set of 20 initial seeds, comprising ten adhering to the specific format and ten not. Each of the eight LLMs is in default parameter configuration and prompted with the three prompts mentioned earlier to generate 100 new testcases per initial seed (for `LLM_F` the prompt does not include any sample seed from the initial corpus but the number of new testcases generated is the same). Such mutation budget is determined empirically as we observe that as the mutation budget grows, the responses from LLMs start to repeat since the identical prompt is being used repeatedly. In order for the unique mutation ratio among LLMs to become apparent, we empirically set the mutation budget to 100 consistently across all LLMs for the experiments.

We then evaluate the quality of generated testcases based on the following metrics:

- *Edge code coverage*: This measures the extent of code covered by the testcases, which is crucial for assessing the effectiveness of LLM for fuzzing. The edge coverage information is collected using the `afl-cmin` tool from AFL++ toolbox.
- *Ratio of unique testcases*: This reflects a mutator's ability to generate diverse mutations from the initial input is essential for LLM to act as a seed mutator. A seed is uniquely identified by its MD5 checksum value.
- *Ratio of valid testcases*: This reflects the proportion of generated seeds that is format-conformed. We use the commands listed in Table 4.2 to verify the validity of seeds for corresponding format.

`LLM_F` vs. `LLM_S` (Generative vs. Mutational-based approach)

By evaluating the quality of mutations generated from each LLM adopting `LLM_F` and `LLM_S`, we try to determine whether the generative approach or the mutation-based approach is more effective for using LLMs as program input generators. To that end, we examine the code coverage achieved using the two prompt designs and present the result in Table 4.4.

Table 4.4: Coverage Achieved by LLM_F and LLM_S

LLM Mutator	**jq**		**php**		**mujs**		**objdump**	
	-F	-S	-F	-S	-F	-S	-F	-S
GPT3.5	5873	**5909**	**21558**	16041	4638	**11223**	2488	**2902**
GPT4.0	**7646**	6693	**28585**	17284	8395	**12088**	**2080**	2016
GPT4O	**7554**	6859	**29719**	19322	5945	**15081**	2077	**3068**
CL-in	4587	**5316**	15486	**18571**	6935	**8344**	1140	**2102**
LL-in	4675	**5558**	8906	**21053**	6977	**10176**	714	**784**
CL	3343	**7421**	**31358**	22852	3275	**10269**	1327	**1599**
LL	4011	**7106**	15768	**17976**	5051	**12296**	706	**2199**
STAR	**10092**	9876	32054	**34385**	14745	**24727**	717	**1068**

According to the results, the LLM mutator adopting the `LLM_S` design achieved an average improvement of 26.34%, 6.29%, 103.81% and 54.55% over those using the `LLM_F` design across the four targets. These results suggest that LLMs employing a mutational-based approach generally perform better than those operating in a black-box generative manner.

We further investigated `LLM_S`'s coverage improvement within the large LLM set and small LLM set respectively. According to the results, with large LLMs using the `LLM_S` prompt, the models on average find 7.02% and 33.37% less code coverage than those using the `LLM_F` design for programs `jq` and `php`. However, for the other two targets, the `LLM_S` models find 113.22% and 20.42% more coverage. As for small LLMs, the `LLM_S` prompt consistently leads to higher code coverage compared to the `LLM_F` prompt.

`LLM_FS` vs. `LLM_F` (Impact of sample inclusion)

To further investigate the impact of including a sample input in the prompt, we investigate the two prompt designs: `LLM_FS` and `LLM_F` in terms of the code coverage findings.

Table 4.5: Coverage Achieved by LLM_F and LLM_FS

LLM	**jq**		**php**		**mujs**		**objdump**	
Mutator	-F	-FS	-F	-FS	-F	-FS	-F	-FS
GPT3.5	5873	**6599**	21558	**22039**	4638	**12064**	2488	**3054**
GPT4.0	7646	**7663**	**28585**	22639	8395	**15069**	2080	**2874**
GPT4O	7554	**8150**	**29719**	24225	5945	**18915**	2077	**3239**
CL-in	4587	**6676**	15486	**19534**	6935	**13610**	**1140**	1033
LL-in	4675	**6010**	8906	**19335**	6977	**10359**	714	**869**
CL	3343	**7559**	**31358**	24121	3275	**10469**	**1327**	1035
LL	4011	**7189**	15768	**22556**	5051	**13157**	706	**2207**
STAR	**10092**	9972	32054	**33942**	14745	**24292**	717	**1186**

Table 4.6: Seed Uniq Ratio(%) w/&o Sample Seed

LLM	**jq**		**php**		**mujs**		**objdump**	
Mutator	-F	-FS	-F	-FS	-F	-FS	-F	-FS
GPT3.5	**37.65**	32.93	**82.60**	73.10	**98.70**	55.05	**98.30**	40.60
GPT4.0	**89.32**	42.37	**99.95**	72.25	**99.95**	78.10	82.60	**97.45**
GPT4O	**99.80**	48.75	**100.00**	65.30	**100.00**	82.85	**98.90**	82.60
CL-in	0.25	**8.60**	1.75	**13.45**	7.65	**27.70**	**1.40**	1.35
LL-in	0.75	**52.15**	0.35	**58.45**	8.65	**80.70**	15.20	**25.75**
CL	**53.45**	30.65	**40.80**	16.80	**61.50**	16.85	31.05	**39.55**
LL	30.60	**36.30**	20.05	**22.40**	11.95	**25.25**	26.70	**27.25**
STAR	**99.20**	92.35	97.90	**98.60**	98.00	**98.70**	94.65	**97.65**

As shown in Table 4.5, the LLM mutator adopting the `LLM_FS` design achieved an average improvement of 37.34%, 16.51%, 130.92%, 48.15% over those using the `LLM_F` design across the four targets. Across the five small LLMs, the `LLM_FS` design fairly consistently outperforms `LLM_F` design by achieving an average coverage improvement of 55.65%, 33.82%, 117.92%, 53.67% on the four targets respectively. Across the large LLMs, `LLM_FS` design gained 6.82%, 152.59% and 38.96% coverage improvement on `jq, mujs` and `objdump` while ahieved 12.35% less coverage for `php` as compared with `LLM_F` design. The results indicate that in general `LLM_FS` design leads to better code coverage than `LLM_F` design across both large and small LLM set.

With `LLM_F` design, the LLM mutators are repeatedly queried using the identical prompt which potentially could lead to higher duplicated responses and thusly low unique seed ratio in the produced mutations. To validate this hypothesis, we investigated the ratio of unique mutations for each LLM mutator adopting the two prompt designs respectively. As demonstrated in Table 4.6, for large LLMs, incorporating a sample seed in the prompt actually results in a lower unique seed ratio across the four targets. As for small LLMs, the model adopting `LLM_FS` design is able to generate more unique mutations than its `LLM_F` counterpart. The disparity between the large and small LLM set could be attributed to the fact that large LLMs are trained on larger dataset and more intelligent, thus being more resilient to duplicated prompts than small LLMs.

Table 4.7: Coverage Achieved by LLM_S and LLM_FS

LLM Mutator	**jq**		**php**		**mujs**		**objdump**	
	-S	-FS	-S	-FS	-S	-FS	-S	-FS
GPT3.5	5909	**6599**	16041	**22039**	11223	**12064**	2902	**3054**
GPT4.0	6693	**7663**	17284	**22639**	12088	**15069**	2016	**2874**
GPT4O	6859	**8150**	19322	**24225**	15081	**18915**	3068	**3239**
CL-in	5316	**6676**	18571	**19534**	8344	**13610**	**2102**	1033
LL-in	5558	**6010**	**21053**	19335	10176	**10359**	784	**869**
CL	7421	**7559**	22852	**24121**	10269	**10469**	**1599**	1035
LL	7106	**7189**	17976	**22556**	12296	**13157**	2199	**2207**
STAR	9876	**9972**	**34385**	33942	**24727**	24292	1068	**1186**

`LLM_FS` vs. `LLM_S` (Impact of format inclusion)

To investigate the impact of incoporating the expected format of the generated program inputs, we compare the code coverage findings achieved by `LLM_FS` and `LLM_S` designs respectively. According to the result in Table 4.7, the LLM mutator adopting the `LLM_FS` design achieved an average improvement of 10.34%, 15.06%, 16.21% for the three targets executing text-based inputs, while for `objdump` where ELF format inputs are expected, the code coverage dropped slightly for 1.31%. In particular, for the three large LLMs, the `LLM_FS` consistently performed better than `LLM_S` by 15.00 to 31.25%. As for the five small LLMs, the improvement of `LLM_FS` achieved 5.35 to 14.42% improvement over `LLM_S` design while on `objdump`, the code coverage dropped for 12.77%.

Table 4.8: Seed Valid Ratio(%) w/&o Format Info

LLM Mutator	**jq**		**php**		**mujs**		**objdump**	
	-S	-FS	-S	-FS	-S	-FS	-S	-FS
GPT3.5	48.02	**92.63**	89.35	**97.90**	50.75	**64.20**	**48.80**	40.40
GPT4.0	58.28	**88.35**	92.49	**99.90**	56.35	**68.25**	27.00	**95.80**
GPT4O	51.85	**83.55**	98.00	**99.60**	55.80	**69.25**	44.40	**62.20**
CL-in	49.65	**87.40**	**74.40**	63.70	55.00	**59.05**	**40.40**	0.20
LL-in	**39.15**	20.15	71.80	**75.30**	**48.25**	14.90	0.00	0.00
CL	27.90	**30.10**	**88.70**	71.15	29.80	**46.30**	**0.45**	0.05
LL	12.70	**29.65**	**91.35**	90.40	24.10	**34.95**	10.40	**12.00**
STAR	31.05	**38.25**	**71.00**	68.30	**36.50**	35.30	0.00	**0.05**

The assumption is that incorporating format information in the prompt could enhance the chance of generating format-conformed seeds, and therefore enabling the exploration to deeper program states. We examined the ratio of valid seeds in the produced mutations for each LLM mutator. As is shown in Table 4.8, across the three programs executing text-based inputs, the LLM mutators adopting `LLM_FS` is able to produce higher ratio of valid seeds than those adopting `LLM_S`. As for `objdump`, the small LLMs in general produce less valid ELF inputs than larger LLMs.

In summary, LLMs perform better in mutation-based mode compared to generative mode, especially when the expected format of the mutations is included in the prompt.

4.2.2 RQ2: Diversity of LLM-generated Mutations

Here, we delve deeper into the mutations generated by LLMs and contrast them with those generated by the mutation engine of the conventional greybox fuzzers, AFL++ and Gramatron. We examine and compare the mutation quality across the subject LLM mutators as well as the conventional greybox fuzzers.

Table 4.9: Rare Coverage Ratio(%)

Program	**GPT3.5**	**GPT4.0**	**CL-in**	**LL-in**	**CL**	**LL**	**AFL++**
jq	16.55	7.19	0.00	0.00	0.00	0.00	76.26
php	13.52	15.74	0.00	0.06	0.03	2.13	68.52
mujs	26.39	46.67	0.00	0.00	0.00	0.00	23.97
xml	75.33	0.12	0.00	0.00	0.03	0.00	24.52
Average	32.95	17.43	0.00	0.02	0.02	0.53	48.32

Code coverage is classified as “rare” if it can only be triggered exclusively by one of the seven mutators. We aggregate the code coverage findings across all mutators in the 8-hour trials for each target program and identify the total rare code coverage for each target program. Then, we compute the uniquely identified code coverage for each baseline and calculate the ratio of rare code coverage against the total amount of rare coverage for each baseline. The results are presented in Table 4.9.

The results suggest that larger LLMs uncover significantly more rare coverage (on average 32.95% for `GPT3.5` and 17.43% for `GPT4.0`) compared to smaller models, which on average are below 1%. However, a big proportion of the rare coverage, 48.32%, can only be detected using conventional mutators of AFL++, indicating that none of the tested LLMs can fully dominate the coverage findings or replace the conventional mutator. Moreover, there is potential for achieving even more code coverage by leveraging a collaborative approach between large LLMs and greybox fuzzers.

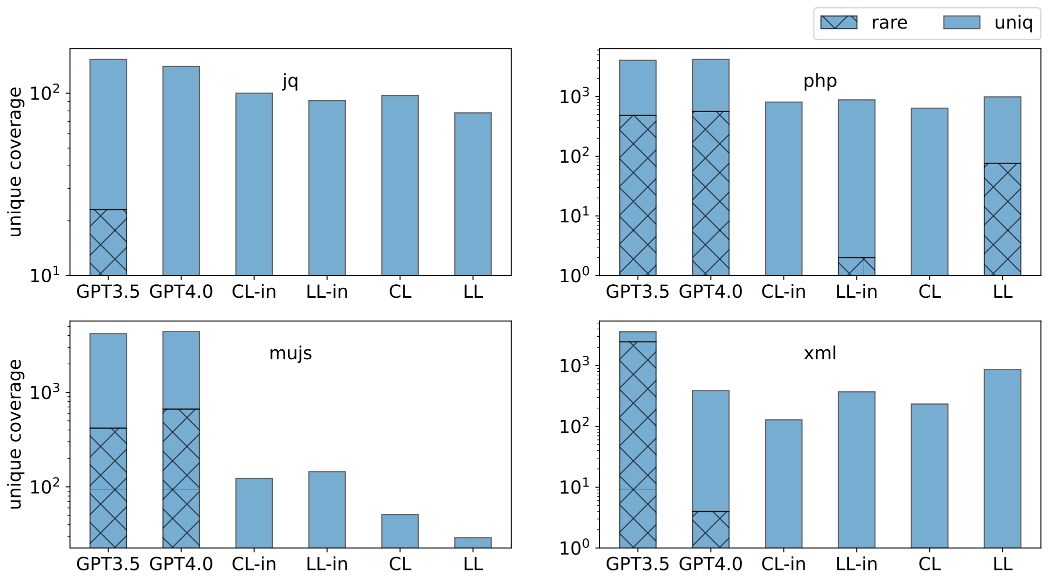


Figure 4.1: Unique Code Coverage

To determine which LLM collaborates more effectively with AFL++, we conduct further investigation to identify each LLM's unique coverage findings not detected by AFL++. We illustrate each LLM's unique code coverage (labeled as "uniq"), as well as the rare code coverage (labeled as "rare") in Figure 4.1. Please note that in this figure a log scale was applied to the y-axis to improve visualization clarity.

Based on the results, the `GPT3.5` model generally outperforms the other five LLMs and achieves the most unique code coverage that AFL++ cannot detect. Following closely behind is the GPT4.0 model. In contrast, the four smaller LLMs detects significantly less unique coverage compared to the larger models. These findings suggest that larger LLMs may hold greater potential as collaborative components for conventional greybox fuzzers like AFL++.

Substitution or Coordination?

Firstly, we examine the code coverage findings achieved by each of the ten mutators and investigate the Pairwise Unique Coverage across the ten mutators on the four targets. Note that for Gramatron, it does not support json or ELF format inputs.

Table 4.10: Edge Coverage Achieved in 4h

Mutator	**jq**	**php**	**mujs**	**objdump**
GPT3.5	5500	18713	12042	1817
GPT4.0	5133	18456	11669	1712
GPT4O	5207	20071	14067	1803
CL-in	4232	12892	5086	559
LL-in	4175	13151	5664	555
CL	4227	12391	5271	563
LL	4194	12922	5379	571
STAR	5125	15358	7493	574
AFL++	5587	18735	8892	6029
GRAM	-	18859	10092	-

According to Table 4.10, for `objdump` which takes non-text-based inputs, AFL++ demonstrates significant superiority over the three large LLMs, which in turn outperform the five small LLMs. For the other three targets which take text-based inputs, the large LLMs reached similar or higher code coverage than AFL++. The small LLMs in general achieved less code coverage than large LLM, while `STAR` performs better than the other four Llama 2 models. As for the grammar-aware fuzzer `GRAM`, it is able to outperform AFL++ in both `php` and `mujs` and ranked second place in `php` while performs worse than three large LLMs in `mujs`.

Table 4.11: Dominating/Dominated Mutator Count (obj: objdump)

Mutator	**# dominating**					**# dominated by**				
	jq	**php**	**mujs**	**obj**	**AVG**	**jq**	**php**	**mujs**	**obj**	**AVG**
GPT3.5	2	0	2	4	2.00	0	0	0	0	0.00
GPT4.0	1	0	0	4	1.25	0	0	0	1	0.25
GPT4O	4	1	3	5	3.25	0	0	0	0	0.00
CL-in	0	0	0	0	0.00	3	1	2	6	3.00
LL-in	0	0	0	0	0.00	3	0	0	7	2.50
CL	0	0	0	2	0.50	1	0	2	5	2.00
LL	0	0	0	3	0.75	1	0	1	4	1.50
STAR	1	0	0	1	0.50	0	0	0	1	0.25
AFL++	0	0	0	5	1.25	0	0	0	0	0.00
GRAM	-	0	0	-	0.00	-	0	0	-	0.00

We further investigate the dominant relationship between mutators to determine if any mutator can fully substitute another by covering all of its discovered edges. One mutator A *dominates* the coverage finding of another mutator B if A is able to cover all the coverage of B. As is shown in Table 4.11, for each mutator we examine the average number of other mutators that it dominates across four targets and report the rank as follows: `GPT4O` (3.25) > `GPT3.5` (2.00) > `GPT4.0` = `AFL++` (1.25) > `LL` (0.75) > `STAR` = `CL` (0.50) > `CL-in` = `LL-in` = `GRAM` (0.00). In particular, `AFL++` is able to dominate the coverage findings of all five small LLMs but none of the three large LLMs for `objdump`. This indicates the weakness of small LLMs for generating high quality seeds in non-text-based format.

The results also suggest that the four Llama 2 models are dominated by the highest number of mutators: `CL-in` (3.00) > `LL-in` (2.50) > `CL` (2.00) > `LL` (1.50), followed by the `STAR` and `GPT4.0` models at 0.25 each. The two conventional fuzzers, `AFL++` and `GRAM`, along with the other two large LLMs, `GPT3.5` and `GPT4.0`, are not dominated by any other mutators.

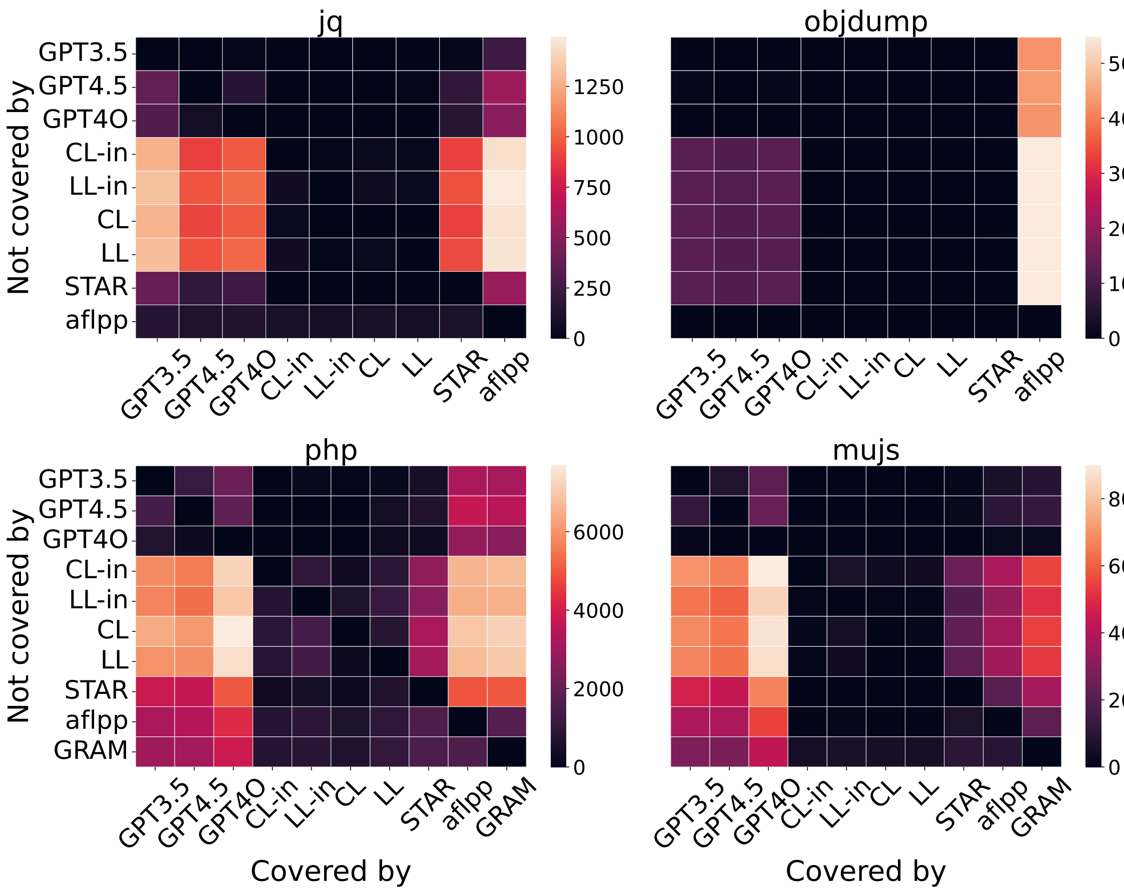


Figure 4.2: Pairwise Unique Coverage

In summary, none of the LLMs can consistently outperform AFL++ and therefore cannot substitute conventional fuzzer. The coordination mode is potentially a better choice.

To investigate the coordination potential between the tested mutators, we examine the Pairwise Unique Coverage across the ten mutators and present the result in Figure 4.2. Each colored box indicates the number of unique edges covered by the column mutator but not the row mutator. A lighter-colored box indicates a higher potential for improving the coverage of the row mutator by incorporating the column mutator. As depicted in the figure, incorporating large LLMs has the high potential to improve the coverage efficiency of small LLMs across the three targets expecting text-based inputs. Furthermore, for `php` and `mujs`, large LLMs find unique coverage that the two conventional fuzzers cannot trigger. As for `objdump`, it appears that `AFL++` finds the most unique coverage compared to any of the LLM mutators. Out of the five small LLMs, the four Llama 2 models in general finds way less unique edges across the four targets, while `STAR` finds significant amount of unique edges overlooked by the four Llama 2 models for program `jq` and `php`. As for the conventional fuzzers, AFL++ finds more unique edges over small LLMs rather than large LLMs across four targets.

In summary, it is promising to incorporate large LLM mutators in collaboration with conventional fuzzers for better coverage efficiency, especially for testing programs that expect text-based inputs.

Mutation Diversity in Program Space.

For a program input generator, it is crucial that the generated inputs explore sparse and distinct code regions. And the more different the explored region is compared to a conventional fuzzer, the higher the potential for the two mutators to cooperate for better coverage efficiency. Here, we explore the diversity of mutations generated from the

LLM mutator based on their trajectories into the program state space. In this evaluation, we included two conventional fuzzers, `AFL++` and `GRAM`, in the comparison to discuss the diversity of the LLM explored region against that of conventional fuzzers. To ensure a fair comparison, both conventional fuzzers were modified to retain all mutations without filtering. For each target program, we collected 100 mutations derived from each mutator, all originating from the same initial seed. We used `afl-showmap` to collect the execution trace of each mutation in bitmap form. Then, we performed T-SNE transformation on the set of bitmaps of each mutator to generate the visualization in Figure 4.3.

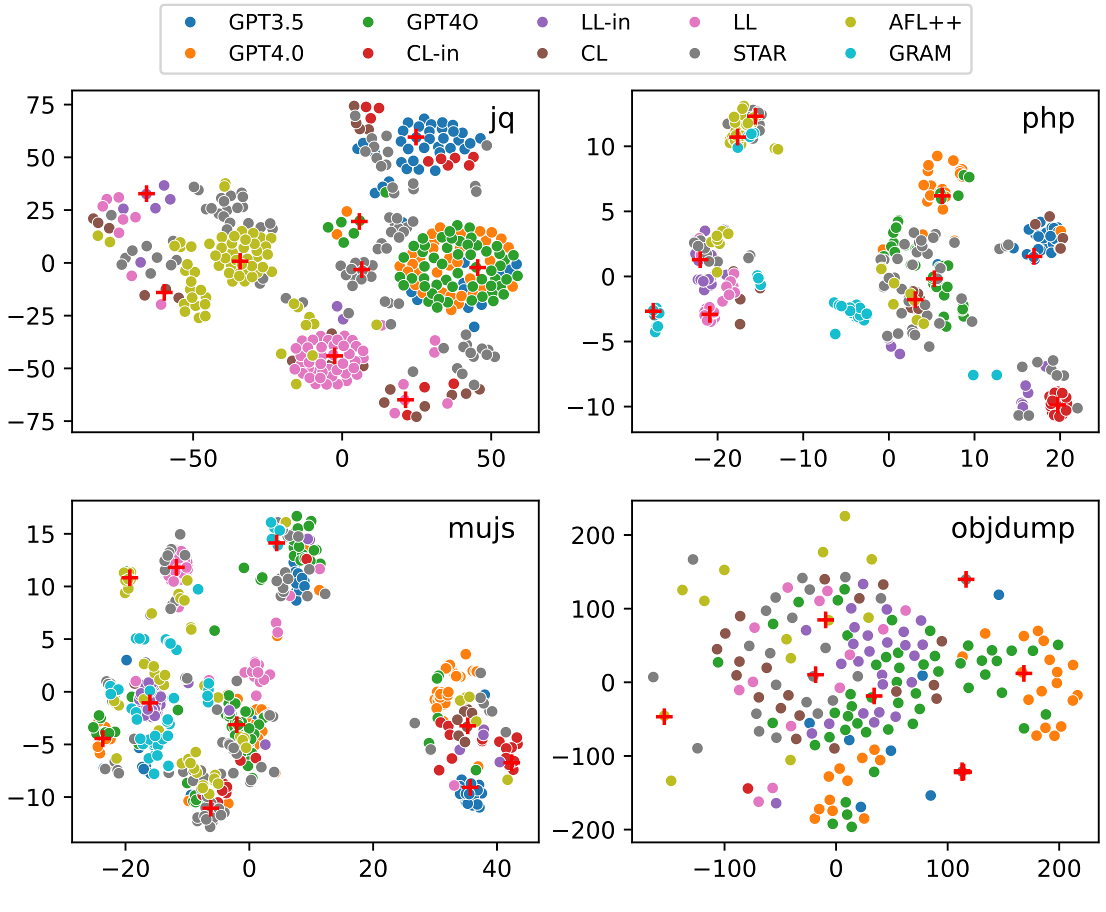


Figure 4.3: Mutation Diversity in Program Space

In Figure 4.3, each dot represents one executed path in the program space. The distance between two dots reflects the difference between two paths. A sparse set of dots indicates that the corresponding mutator is exploring a diverse set of paths. As depicted in the figure, each mutator's explored paths can form one to several small clusters (e.g., `STAR` for `jq`). Some mutators appear to explore similar regions as another mutators (e.g., `GPT4.0` and `GPT4O` for `jq`). For `objdump`, the figure appears less crowded than the other three because of the high amount of duplicated paths. Intuitively, a sparser distribution of dots suggests that an LLM mutator may perform better as a greybox fuzzer compared to denser distributions. Moreover, if its dots are distant from those of AFL++, it could collaborate more effectively with AFL++ to achieve higher coverage findings.

Table 4.12: Distance to AFL++ Centroid

Mutator	**jq**	**php**	**mujs**	**objdump**	**Stat. Rank**
GPT3.5	83.33	35.81	58.47	278.57	2.25
GPT4.0	44.22	24.32	15.87	327.63	4.50
GPT4O	79.76	25.43	22.21	190.15	4.50
CL-in	85.97	42.68	64.14	276.82	2.00
LL-in	44.86	10.38	12.32	328.55	5.00
CL	29.43	24.22	56.35	147.12	6.00
LL	54.93	14.00	7.58	277.48	5.50
STAR	40.99	2.63	25.49	195.44	6.25
GRAM	-	16.61	23.94	-	-

For a quantitative analysis of the result and to help interpret the figure, we perform Kernel Density Estimation (KDE) [49] to identify the centroid for each mutator's coverage bitmap set, using the distribution density of data points in the embedding space. The centroid of each mutator's bitmap set represents the center of the program space explored by the corresponding mutations. Assuming that the data points around each centroid are evenly distributed, if the distance between two centroids is large, the corresponding mutators are exploring different regions of the program execution space. Conversely, if the distance between two centroids is small, the two mutators are likely to be exploring overlapping execution space. To assess the diversity of mutations generated by LLM and those generated from AFL++, we calculate the centroid distance between each LLM and AFL++ and present the results in Table 4.12.

Based on the results, `CL-in` emerge as the top LLM mutator that exhibits difference from `AFL++` in the program execution space, followed by `GPT3.5` and then `GPT4.0` and `GPT4O`. If the distribution of the explored program space of each mutation is evenly distributed, `CL-in` is expected to make the most unique contribution to AFL++, followed by `GPT3.5` and then `GPT4.0` and `GPT4O`.

Table 4.13: KDE Centroid Density

Mutator	**jq**	**php**	**mujs**	**objdump**	**Stat. Rank**
GPT3.5	8.15	6.47	10.24	15.91	4.75
GPT4.0	5.24	9.82	7.14	15.42	3.25
GPT4O	5.51	10.07	6.96	6.54	3.00
CL-in	12.71	13.84	9.86	21.88	7.00
LL-in	14.68	11.84	17.02	15.52	7.00
CL	15.61	12.90	18.41	9.40	7.00
LL	11.34	9.86	14.59	10.55	5.25
STAR	6.54	4.41	4.49	5.46	1.50
AFL++	9.43	14.29	6.25	26.34	6.25
GRAM	-	13.32	11.69	-	-

Additionally, for each mutator, it is crucial that its generated mutations explore a sparse code space. If the execution trajectories are too dense, the mutator may not efficiently explore new code coverage but instead focus on a narrow set of duplicated paths. To assess the coverage diversity of each mutator, we present the density score of the corresponding centroid in Table 4.13.

The result indicates that `STAR` overall has the highest rank with respect to its density score in exploring four targets. It is followed by three large LLMs. However, for `CL-in` which explores regions most distant from AFL++, its mutations are densely distributed around the center of its explored region.

Based on the statistical rankings of the eight LLM mutator in Table 4.12 and Table 4.13, small LLMs either explore paths in close proximity with AFL++ (e.g., `STAR`, `CL` and `LL`), or their explored regions are too dense (e.g., `CL-in`, `LL-in`, `CL`), making them unsuitable as collaborator mutators for greybox fuzzers.

In summary, an effective LLM mutator should explore unique code regions that are distant from the center of AFL++'s explored region. Additionally, its mutations should cover a diverse set of paths rather than being too focused around a central point. Overall, large LLMs tend to perform better than small LLMs.

In summary, while LLM mutators cannot entirely replace AFL++ mutators, they have demonstrated great potential as collaborative components for conventional fuzzers, especially large LLMs. Furthermore, LLMs are capable of exploring diverse and distinct code regions.

4.2.3 RQ3: Cost Effectiveness

Here, we examine the cost-effectiveness of the eight subject LLMs by assessing their code coverage results under the same time or monetary constraints. Initially, given a target program and its corresponding initial corpus, all seeds undergo a filtering process based on their code coverage. Seeds that do not trigger any new code coverage are excluded from the seed pool. Subsequently, each subject LLM is prompted with the `LLM_FS` design. As discussed earlier, in this mode a sample input along with the desired data format of the new mutations are included in the prompt. Such sample input is randomly selected from the filtered seed pool. The LLM generates new mutations, which are then filtered to retain only those that trigger new code coverage. This iterative process continues until the allotted 4-hour time budget is exhausted. We then assess the code coverage findings over time as well as the monetary cost and present the result in Figure 4.4.

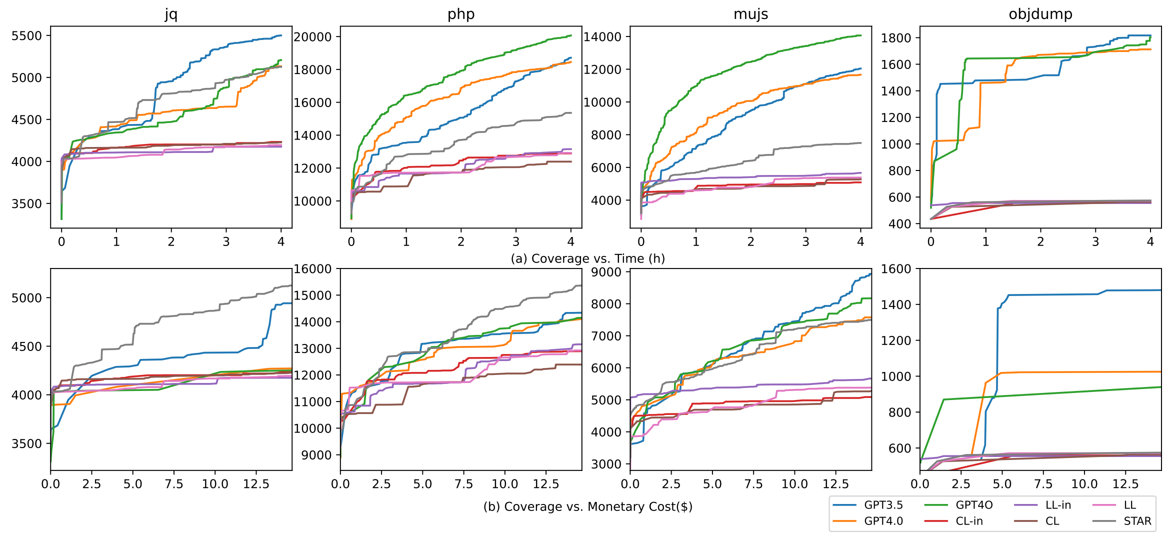


Figure 4.4: Cost Effectiveness

Coverage Over Time

As depicted in Figure 4.4 (a), the three large LLMs ranked top-three across all four targets by finding the most coverage by the end of 4h, when the four Llama 2 models all reached coverage plateau. And STAR model consistently outperforms the other four small LLMs across the four targets. Such result suggests that large LLMs exhibit greater coverage efficiency compared to smaller models and Starcoder2 model consistently outperforms Llama 2 models. Surprisingly, despite GPT4.0 being perceived as more powerful than GPT3.5, it actually performs worse than GPT3.5 as a seed mutator.

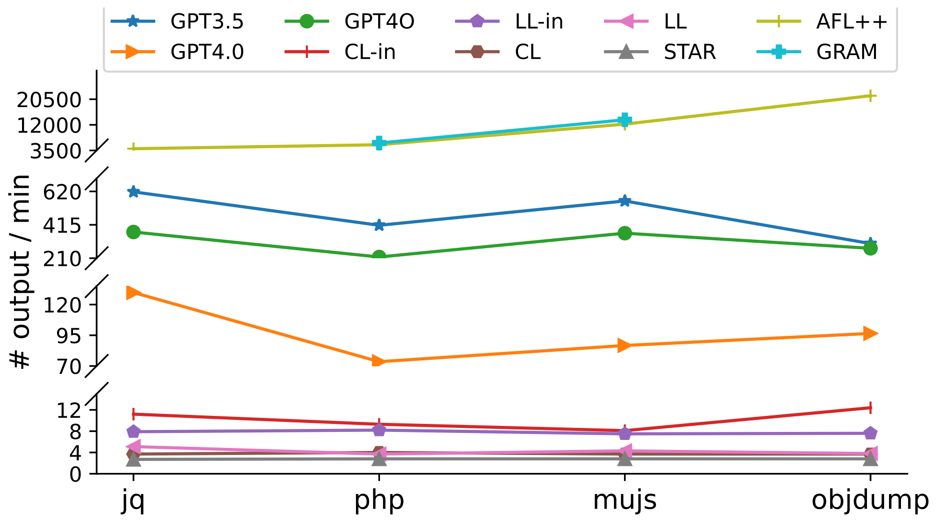


Figure 4.5: Mutation Generation Rate

For further investigate this disparity in coverage gain, we examined the mutation generation efficiency across the eight LLM mutators and compare them against the two conventional fuzzers, AFL++ and GRAM and present the findings in Figure 4.5. According to the results, conventional fuzzers, AFL++ and GRAM, have a much higher mutation generation rate compared to LLM mutators. Within the LLM set, GPT3.5 has the highest rate, followed by GPT4O, and then GPT4.0. Additionally, the large LLMs demonstrate a significantly higher rate than the small LLMs, all of which are below 13 mutations per minute.

For each mutator, we examined the number of new coverage seeds discovered across the 4-hour trials for each target program and reported the seed count as well as its ratio over all mutations generated in Table 4.14. Recall that conventional fuzzers have the highest mutation generation rate, followed by large LLMs, which then followed by small LLMs. However, in terms of new coverage ratio, small LLMs are the best, followed by large LLMs and then conventional fuzzers.

Table 4.14: New Coverage Seeds (obj: objdump)

Baseline	**# newcov seeds**					**newcov ratio (%)**				
	jq	**php**	**mujs**	**obj**	**AVG**	**jq**	**php**	**mujs**	**obj**	**AVG**
GPT3.5	457	1033	1515	97	775.50	0.31	1.04	1.12	0.13	0.65
GPT4.0	207	753	950	63	493.25	0.66	4.27	4.57	0.27	2.44
GPT4O	312	1342	1936	80	917.50	0.35	2.57	2.22	0.12	1.32
CL-in	69	132	122	3	81.50	2.56	5.91	6.29	0.10	3.72
LL-in	69	100	114	4	71.75	3.65	5.07	6.30	0.22	3.81
CL	66	74	57	5	50.50	7.50	7.72	6.48	0.57	5.57
LL	55	60	82	6	50.75	4.53	6.79	8.02	0.65	5.00
STAR	125	203	240	7	143.75	19.00	30.43	36.09	1.06	21.65
AFL++	663	2818	1683	1538	1675.50	0.07	0.22	0.06	0.03	0.10
GRAM	-	2537	1544	-	2040.50	-	0.18	0.05	-	0.12

The results suggest that the high mutation generation rate does not proportionally translate to the number of new coverage seeds discovered by each mutator. On the contrary, small LLMs generate the highest ratio of new code coverage seeds, despite having the lowest seed generation rate. In particular, for `STAR`, the new coverage ratio reached as high as 36% for `php`. For `objdump`, the ratio dropped to 1.06% but is still the highest across eight LLM mutators and higher than `AFL++`. Furthermore, conventional fuzzers have the highest generation rate but the lowest new coverage seed ratio across four targets. This indicates that the high latency of small LLMs largely affected its efficiency in discovering new coverage seeds.

In summary, LLM mutators have a higher probability of producing new coverage seeds but are severely limited by their mutation rate. However, with the ongoing development of hardware, the constraints on generation rate for LLMs could potentially be lifted, paving the way for their substitution of conventional fuzzers in the future.

Coverage vs. Monetary Cost

Another practical factor to consider when utilizing LLM for software engineering tasks is the monetary cost, particularly in fuzzing, where queries to the LLM are made as frequently as possible to achieve higher throughput and generate more seeds within a given unit of time. Common belief is that larger and more powerful LLMs, such as ChatGPT, are too expensive when frequent queries are necessary, while smaller models that can be run on local machines cost significantly less.

Several precedent works have based their design choices on this assumption and resort to smaller LLMs for tasks requiring more frequent model queries. For instance, Fuzz4All adopts the larger and more powerful LLM, GPT-4.0, to summarize PUT documentations and generate informative prompts. These prompts are then used to frequently query a smaller LLM, StarCoder, for seed generation in testing. Similarly in ChatAFL, seed generation queries to the LLM, GPT-3.5, are actively reduced to only when the fuzzer gets stuck. These design choices aim to reduce costs by minimizing queries to larger LLM.

To verify this assumption, we assessed the code coverage findings achieved by each mutator with respect to the monetary cost($) spent. We visualize the detailed coverage growth in relation to monetary cost in Figure 4.4 (b), with the plot adjusted to reflect the lowest cost per program for visual clarity. As is depicted in Figure 4.4 (b), large LLMs consistently make greater coverage gain under the same monetary budget than the four Llama 2 models. On two targets (`mujs`, `objdump`), large LLMs outperformed `STAR` as well. Such results suggest that large LLM in general is more cost effective than small LLMs, especially for programs that take non-text inputs.

In summary, under the same time and monetary budget, large LLMs are generally more cost-effective than small LLMs and can scale beyond generating text-based inputs.

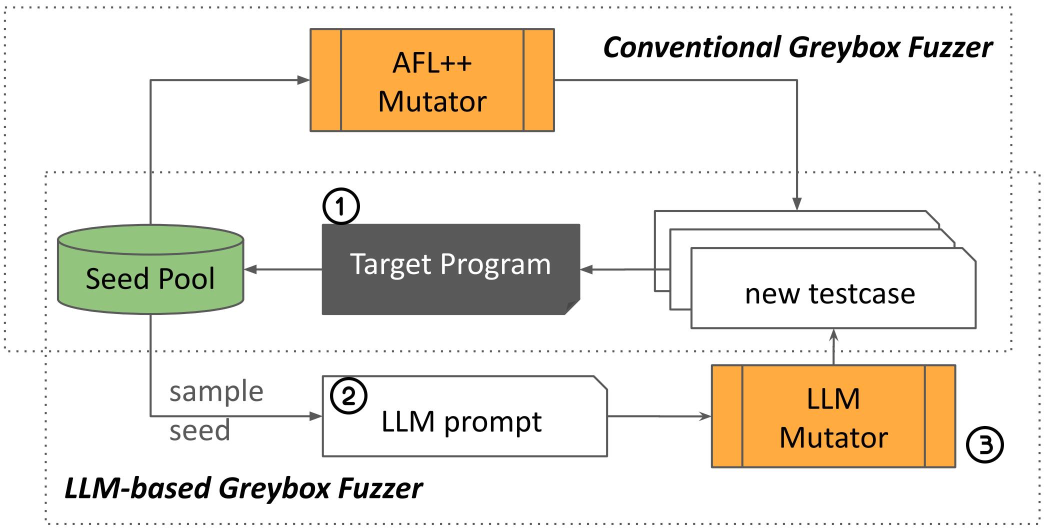


Figure 4.6: LLM-augmented Greybox Fuzzer

4.3 Integration with Traditional Greybox Fuzzer

We construct a GenAI-augmented greybox fuzzer by incorporating a `gpt-3.5-turbo` LLM as assistant mutator for the state-of-the-art greybox fuzzer, AFL++. We present our prototype, ChatFuzz, and evaluate its effectiveness in coverage finding and vulnerability detection compared with baseline approaches. The optimal parameter configuration of the LLM mutator is empirically decided and presented in Table 4.15.

Table 4.15: LLM Mutator Configuration

Model	`max_token`	`n`	`temperature`	**Prompt**	
				format	**input**
AI_CP	256	20	1.25	✓	✓
AI_CT	256	20	1.50	✓	✓
AI_noFORM_CP	256	20	1.25	✗	✓
AI_noFORM_CT	256	20	1.50	✗	✓

We evaluate the efficacy of ChatFuzz by addressing the following research questions: 1) Coverage Efficiency: Can ChatFuzz improve code coverage for end-to-end fuzzing? How does its performance compare to that of non-assisted fuzzers and grammar-based fuzzers? 2) Security Application: Is ChatFuzz more effective at detecting vulnerabilities?

Baselines. To better answer the aforementioned research questions, we use the following baseline configurations:

- **AFL++**. The original AFL++ [26] fuzzer in non-deterministic mode. This configuration has one AFL++ 4.03a instance and no chat mutator. This is the non-assisted greybox fuzzing.
- **TOKEN**. This baseline utilizes the AFL++ fuzzer with a grammar-based mutator: Autotokens[2], implementing the concept introduced in [56]. This is the greybox fuzzer assisted by grammar-based mutator.

[2]`https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators/autotokens`

- **ChatFuzz**. The full-fledged ChatFuzz model integrating *AI_ CP* mutator with AFL++. In this configuration, AFL++ fuzzer instance and the chat mutator run as two separate processes. The chat mutator randomly picks one seed from the fuzzer queue as a sample input to prompt *AI_ CP* model for new variations with the parameter configurations presented in Table 4.15. The expected format of the seeds is included in the prompt. The AI model's response will undergo parsing to eliminate comments or leading words, ensuring that only the formatted input is retained. Fuzzer instance periodically inspects the new seeds generated from AI model to import those that trigger new edge coverage into the fuzzer queue. This and the following three configurations are AI-assisted greybox fuzzers.

- **ChatFuzz-C**. This model substitutes the *AI_ CP* model with *AI_ CT* model in ChatFuzz. Parameter configuration of the *AI_ CT* model is presented in Table 4.15. The *AI_ CT* model has significantly higher latency than the *AI_ CP* model but is able to generate fewer duplicates and more valid seeds.

- **ChatFuzz-F**. This model integrates AFL++ with *AI_ noFORM_ CP*. Compared with the full-fledged ChatFuzz, this baseline removes input format information from the model prompt. In other words, this is the format-agnostic setting of ChatFuzz.

- **ChatFuzz-CF**. This model integrates AFL++ with *AI_ noFORM_ CT*. With this model,we remove the input format information from the prompt of ChatFuzz-C model's AI mutator. This is the format-agnostic setting of the ChatFuzz-C model.

Dataset. Currently, ChatFuzz supports target programs that take in text-based format-

Table 4.16: Benchmarks

Type	Program	Version	Input Format
data	jq	jq-1.5	json
	php	php-fuzz-parser_0dbedb	PHP
	xml	libxml2-v2.9.2	xml
	jsoncpp_fuzzer	jsoncpp	json
code	mujs	mujs-1.0.2	js
	ossfuzz	sqlite3_c78cbf2	SQL
	cflow	cflow-1.6	C
	lua	lua_dbdc74d	lua
text	curl_fuzzer_http	curl_fuzzer_9a48d43	HTTP response
	openssl_x509	openssl-3.0.7	DER certificate
	base64	LAVA-M	.b64 file
	md5sum	LAVA-M	md5 checksum

Table 4.17: Unsupported Targets

Benchmark	Targets	Unsupported Format	Benchmark	Targets	Unsupported Format
Fuzzbench [2]	libjpeg-turbo-07-2017	image	Unibench [42]	exiv2	image
	libpng-1.2.56	image		gdk-pixbuf-pixdata	image
	bloaty_fuzz_target	ELF, Mach-O, etc.		imginfo	image
	freetype2-2017	TTF, OTF, WOFF		jhead	image
	harfbuzz-1.3.2	TTF, OTF, TTC		tiffsplit	image
	lcms-2017-03-21	ICC profile		lame	audio
	libpcap_fuzz_both	PCAP		mp3gain	audio
	mbedtls_fuzz_dtlsclient	other		wav2swf	audio
	openthread-2019-12-23	other		ffmpeg	video
	proj4-2017-08-14	other		flvmeta	video
	re2-2014-12-09	other		mp42aac	video
	systemd_fuzz-link-parser	other		nm	binary
	vorbis-2017-12-11	OGG		objdump	binary
	woff2-2016-05-06	WOFF		tcpdump	network
	zlib_zlib_uncompress_fuzzer	Zlib compressed		infotocap	terminfo file
LAVA-M [25]	who	utmp file		pdftotext	pdf
	uniq	other			

ted inputs. We inspect 45 programs from three benchmarks heavily evaluated in precedent works: Unibench, Fuzzbench and LAVA-M and list 12 programs that ChatFuzz can generate realistic inputs for in Table 4.16. Specifically, we classify the 12 targets into three categories based on the format of the input seeds: 1) formatted data file; 2) source code in different programming languages; and 3) text with no explicit syntax rules. We further show the reasons why the other 33 programs are not included in the evaluation in Table 4.17.

Experiment Setup. All experiments were conducted on a workstation with two-socket, 48-

core, 96-thread Intel Xeon Platinum 8168 processors. The workstation has 768G memory. The operating system is Ubuntu 18.04 with kernel 5.4.0. Note that each fuzzing trial is assigned one dedicated core to ensure fair comparison.

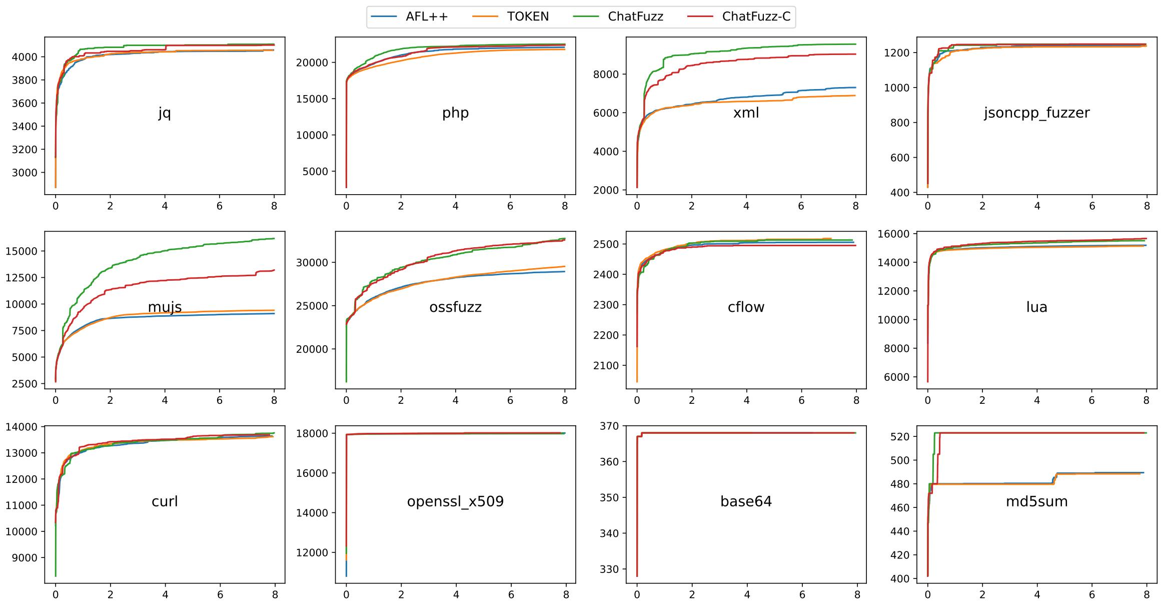


Figure 4.7: End-to-End Fuzzing Coverage Growth

4.3.1 RQ4: Coverage Efficiency

To demonstrate the coverage efficiency of ChatFuzz, we measure the edge coverage achieved by running AFL++, TOKEN, ChatFuzz and ChatFuzz-C on 12 programs for 8h. To reduce randomness, we obtain the average code coverage across five trials for AFL++ and TOKEN. The coverage growth in 8h is shown in Figure 4.7. Programs that accept data files as input are listed in the first row, those requiring source code files are in the second row, and the programs in the last row do not have explicit syntax rules for their input. We further report the coverage improvement of TOKEN, ChatFuzz and ChatFuzz-C over AFL++ and the unique coverage triggered by the corresponding assistant mutators in Table 4.18.

Table 4.18: Coverage Analysis

Program	**8h Code Coverage**				**Improvement over AFL++ (%)**			**Uniq. cov. triggered by assist. mutator**		
	AFL++	TOKEN	ChatFuzz	ChatFuzz-C	TOKEN	ChatFuzz	ChatFuzz-C	TOKEN	ChatFuzz	ChatFuzz-C
jq	4059	4057	**4109**	4101	-0.05	**1.24**	1.04	**8**	6	0
php	22089	21784	**22509**	22390	-1.38	**1.90**	1.36	126	**361**	167
xml	7304	6894	**9551**	9035	-5.61	**30.76**	23.70	**118**	28	1
jsoncpp	1240	1238	**1248**	**1248**	-0.19	**0.61**	**0.61**	**4**	0	0
mujs	9099	9406	**16173**	13205	3.37	**77.74**	45.12	345	**1027**	359
ossfuzz	28940	29535	**32751**	32582	2.06	**13.17**	12.59	**933**	889	761
cflow	2506	**2519**	2513	2495	**0.54**	0.30	-0.42	**26**	0	0
lua	15184	15132	15507	**15657**	-0.34	2.13	**3.12**	71	**353**	277
curl	13642	13624	**13764**	13687	-0.13	**0.89**	0.33	**4**	0	0
openssl_x509	18010	18001	17976	**18016**	-0.05	-0.19	**0.03**	0	0	0
base64	368	368	368	368	0.00	0.00	0.00	0	0	0
md5sum	489	488	**523**	**523**	-0.20	**6.87**	**6.87**	0	0	0
Average	10244	10254	**11416**	11109	0.10	**11.44**	8.44	136	**222**	130

Across the 12 target programs, ChatFuzz achieves a 11.44% increase in code coverage compared to AFL++, while ChatFuzz-C exhibits an 8.44% improvement. TOKEN demonstrates similar performance to AFL++. The improvement varies significantly across different programs. Notably, ChatFuzz and ChatFuzz-C achieved improvements of over 10% compared to AFL++ for programs such as `xml, mujs`, and `ossfuzz`, which takes highly structured data format: XML, Javascript and SQL. And for the four programs with no explicit input format (`curl, openssl_x509, base64, md5sum`), TOKEN, ChatFuzz and ChatFuzz-C were able to make only small to no improvement.

We further investigate the unique code coverage triggered by the assistant mutator of TOKEN, ChatFuzz and ChatFuzz-C. By the end of the fuzzing campaign, no fuzzer-originated seeds triggered this particular section of the code region. Therefore, it measures the unique contribution of the assistant mutator. It's important to note that zero unique code coverage doesn't necessarily mean there was no contribution from the assistant mutator. There are cases where seeds from the assistant mutator triggered an edge early on, and seeds from the fuzzer triggered it afterward. As a result, the fuzzer can explore more code coverage due to this initial discovery. We delve deeper into this aspect in **??**.

Based on the results, Autotokens in TOKEN uniquely triggered 1636 edges for 9 programs, while AI_CP in ChatFuzz uniquely triggered 2664 edges for 6 programs, and AI_CT in ChatFuzz-C managed to uniquely discover 1565 edges for 5 programs. The result indicates that grammar-based assistant is able to find unique coverage for more targets while AI-assistant is able to find more unique code coverage.

In summary, AI-assisted fuzzing can significantly enhance code coverage compared to both non-assisted fuzzer and the grammar-based approach.

Table 4.19: Bug Detection

Program	**# of crashes**					**# of unique bugs**				
	AFL++	**ChatFuzz**				**AFL++**	**ChatFuzz**			
			-F	**-C**	**-CF**			**-F**	**-C**	**-CF**
mujs	1.0	3.20	4.40	**5.20**	3.00	0.5	1.2	**5.0**	2.6	1.2
cflow	244.3	241.20	248.20	**264.60**	238.40	2.7	3.0	2.8	**3.2**	3.0
base64	**81.1**	80.40	80.50	80.00	81.00	45.8	46.4	45.8	44.4	**46.6**
md5sum	144.1	140.80	**144.50**	141.60	116.80	**54.6**	52.0	*53.2*	51.8	*52.4*
Average	117.6	116.40	119.40	122.85	109.80	25.9	25.7	**26.7**	25.5	25.8
Rank	2.8	3.8	**2.0**	2.5	4.0	3.5	2.8	2.5	3.3	**2.3**

4.3.2 RQ5: Security Application

In this section, we evaluate the bug detection ability of AFL++, and ChatFuzz in four different configurations. Specifically, we inspect the program crashing inputs for six programs including four from UniBench benchmark and two from LAVA-M dataset. In Table 4.19, we present the number of unique bugs detected in 24h by each baseline respectively. To reduce randomness, we report the average unique bug detection count across five trials for ChatFuzz and its variations. For AFL++, we report the result across ten trials. The results of `jq` and `ossfuzz` are omitted from the table as none of the tested fuzzers was able to trigger any crash.

According to the result, ChatFuzz-C on average finds the highest number of crashing inputs across all baselines, and ChatFuzz-F finds the highest amount of unique bugs. Statistical ranking result suggests that AI-assisted fuzzers outperform AFL++ in finding unique bugs.

In summary, AI-assisted fuzzers outperform non-assisted fuzzer in bug detection.

4.4 Discussion

Previous findings have demonstrated the ability of out-of-the-box LLMs to generate fuzzing inputs and, moreover, the potential for collaboration with conventional greybox fuzzers. However, the efficacy of an LLM-based greybox fuzzer varies depending on several practical factors such as the input type (text-based or non-text-based) of the **target program** (① in Figure 4.6), access to sample inputs and/or its format to formulate the **LLM prompt** (② in Figure 4.6) as well as the particular **LLM** (③ in Figure 4.6) used. Here, we provide a discussion on how to better leverage LLMs for greybox fuzzing.

4.4.1 Target Program

Based on prior findings, when testing programs expecting text-based inputs (e.g., `jq`, `php`, `mujs`), we can opt for an LLM-based greybox fuzzer and potentially achieve higher coverage findings than with a conventional fuzzer. However, when testing programs expecting non-text-based inputs (e.g., `objdump`), we still need to rely on conventional fuzzers.

As is demonstrated in our experiments, ChatFuzz can be utilized for fuzzing programs that execute on text-based formatted input. Other data formats such as image, video, audio or data format that requires particular data loader such as pdf, WOFF or compressed file are not supported. ChatFuzz is limited by the AI model used in the chat mutator. With a generative AI model that supports more data format, for instance DALL·E, ChatFuzz will able to generate testcases in the format of image. For text-based input that does not have explicit syntax rules like hash values, it is hard for generative AI to figure out the underlying rule for the text and therefore the chat mutator could make low to none contribution to the greybox fuzzer.

4.4.2 Prompt Formulation

According to our findings, LLM mutators generally produce mutations with better coverage efficiency when both a sample input and the expected program input format are included in the prompt.

Depending on the goal of the program input generator, the LLM prompt can be formulated differently. In particular, for large LLMs, removing the sample from the prompt can significantly increase the chance of producing a unique seed compared to including a sample. However, for the instruction-mode Llama 2 models, including a sample introduce higher chance of unique mutation.

If the goal is to generate as many syntactically valid mutations as possible, the format information should be included in the prompt. However, if the format is unknown to the testing engine, LLM mutators still have the potential to infer the format from the sample and produce syntactically valid seeds, especially for large LLMs compared to small LLMs.

4.4.3 LLM Choice

Previous works leveraging LLMs for generating fuzzing inputs, such as ChatAFL and Fuzz4all, aim to reduce queries to large LLMs and/or delegate the frequent input-generation queries to small LLMs, in the interest of saving time and monetary costs. This design is based on the assumption that small LLMs, although less powerful than large LLMs, are more cost-effective. Our findings disprove this assumption and demonstrate that large LLMs are more cost-effective, both in terms of time and monetary cost, than small LLMs. Moreover, for large LLMs, newer models do not necessarily guarantee better performance. For example, `GPT3.5` is consistently more effective than `GPT4.0` across the benchmark set, in both time and monetary cost. Therefore, when selecting an LLM for generating fuzzing inputs, size matters more than timeliness. One should aim for larger models but not necessarily the newest ones, while considering the time and monetary budget.

However, we acknowledge that the suboptimal results of small LLMs are partially due to the lack of more powerful GPU instances, where the model query rate can be significantly enhanced and potentially lead to better results. According to our evaluation, small LLMs actually have a better chance of generating new coverage seeds than large LLMs. For instance, on `mujs`, 36.09% of the mutations generated from small LLM `STAR` have triggered new code coverage which is exceedingly high, compared with that of large LLM `GPT4.0` (4.57%) or that of SOTA greybox fuzzer `AFL++` (0.06%). However, such advantage does not translate to better coverage efficiency (`GPT4.0` and `AFL++` reached 1.6x and 1.2x the edge coverage of `STAR` respectively) due to the high model latency. Therefore, researchers with access to powerful GPU machines can opt for small LLMs to generate fuzzing inputs.

4.4.4 Collaboration with Traditional Fuzzer

Our findings have demonstrated the potential of collaborating LLM mutators with conventional greybox fuzzers. This collaboration can expand the testing targets of LLM-based greybox fuzzer from being limited to text-based inputs to include generic programs, while also improving coverage efficiency.

One naive approach is to introduce an LLM mutator to operate in parallel with a greybox fuzzer, while sharing the same seed pool with the fuzzer and relying on the fuzzer to periodically synchronize the LLM-generated mutations. However, the seed scheduling for the LLM mutator could be different from that of the conventional fuzzer. To ensure that each seed is fuzzed for a sufficient duration, the LLM mutator needs to maintain a relatively small and distinct set of seeds. With LLM mutators, what are the characteristics of a good sample input remains an open question, and we will explore this in our future work.

Chapter 5

POLARIS: A Path-Corrective LLM Assistant for Concolic Execution

A whitebox fuzzer utilizes a directional mutator that generates new seeds by solving path constraints to explore a prioritized branch selected by the scheduler. In our earlier work, we introduced and integrated a reachability-guided branch scheduler with a concolic executor, resulting in the prototype system, MARCO. In this approach, the scheduler prioritizes branches based on the estimated amount of new code coverage that could be reached through a path passing that branch. However, there are instances where the newly generated seed from solving constraints does not traverse the expected branch, leading to a phenomenon known as *Path Divergence* [4, 28, 21].

Previous concolic execution engines adopt a filter-based branch-flipping policy that uniquely identifies branches using the bigram of the previous and current branch, flipping only unique branches. This branch filter strategically skips solving constraints that are likely to result in repeated exploration of previously visited branches and thereby avoid repeated path exploration. However, this approach overlooks the issue of path divergence, leading to over 29% redundant path exploration [36]. With Marco, we introduce a path divergence-tolerant branch-flipping policy for concolic execution, prioritizing branches based on a metric that deprioritizes branches prone to high path divergence. Nevertheless, while this policy alleviates the impact of path divergence on concolic execution testing, it does not fully resolve it.

Recent advancements have aimed to improve the efficiency of concolic execution through various techniques, such as dynamic data-flow analysis [18], compiler-based symbolic execution [51], just-in-time gradient descent search [19], and reachability-guided branch scheduling [36]. Despite these developments, path constraint solving remains a major bottleneck in concolic execution performance. Therefore, enhancing the soundness of each constraint-solving attempt and addressing the path divergence issue has become a critical challenge.

5.1 Background and Motivation

In a whitebox fuzzing framework based on concolic execution, the concolic executor runs the target program with a concrete input, guiding the execution down a specific concrete path. Along this path, for each symbolic branch point, the path constraint of the untaken branch is recorded. The branch scheduler then determines whether the branch should be flipped. If so, the corresponding path constraint is sent to an SMT solver. As is shown in Figure 5.1, the SMT solver then evaluates the constraint to determine if a solution exists that satisfies it. If a solution is found, it is provided to the concolic executor to generate a new testcase, driving execution down the previously untaken branch. A preliminary study conducted using Marco for testing *libxml2* reveals that, out of the 4000 path constraints sent from the concolic executor to the SMT solver, 41.9% were satisfiable and led to the generation of new inputs. Further investigation into the new testcases shows that the majority executed the predicted path, accounting for 36.7% of the solving attempts, while 5.3% of the solving attempts resulted in path divergence.

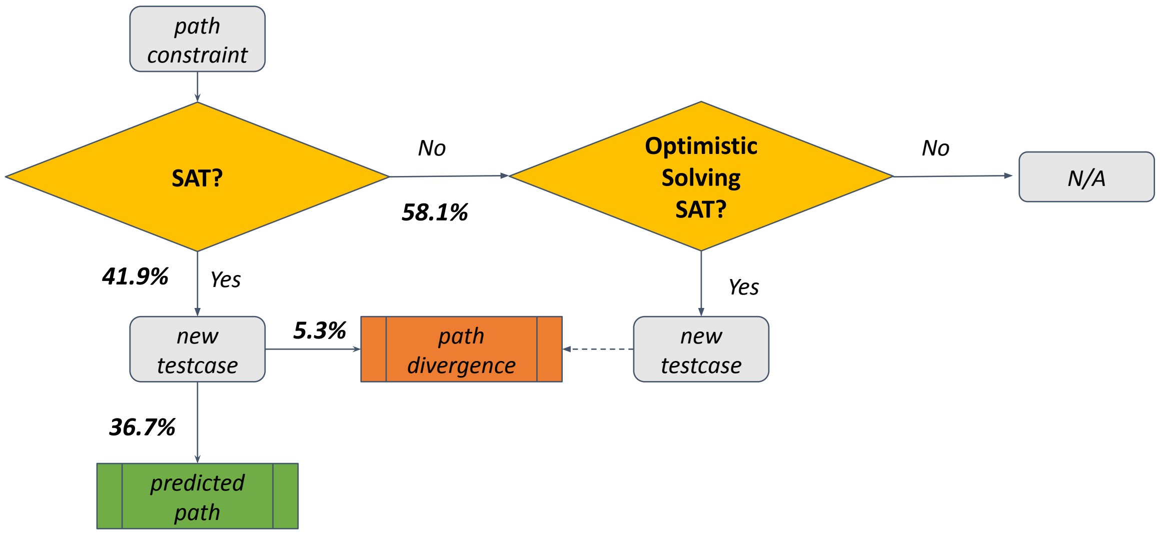


Figure 5.1: Path Divergence Analysis of Marco

Conventionally, if the path constraint is unsatisfiable or a viable solution is not found within a preset time limit, the SMT solver abandons the attempt, resulting in no new test case being generated from that path constraint. Our study on Marco shows that 58.1% of the path constraints are unsatisfiable. However, Marco employs optimistic solving, where we resort to only solving the constraint of the current branch, ignoring the other constraints accumulated from preceding branches in the path sequence. This technique, originally proposed in QSYM [75], has been shown to effectively improve coverage finding and vulnerability detection in concolic execution engines and has become a common practice in later concolic execution engines [18, 36]. However, the testcase generated from optimistic solving is only an optimistic speculation of the true testcase that would traverse the predicted path through the targeted untaken branch, and it may lead to path divergence. In this sense, up to 63.4% of path constraint solutions may not actually traverse the predicted path, with 58.1% of these originating from optimistic solving.

To diagnose the root causes of path divergence, CREST [21] presented a empirical study and identified eight divergent patterns of which the three most prevalent patterns lead to 82% of total path divergence: exceptions, external calls and type casts. In particular, when exceptions occur, execution will be directed to related exception handling routine and thereby lead to an unexpected path. When the program initiate an external call which concolic executor cannot perform symbolic tracing for, the associated constraints will not be included in the generated path constraints. Type cast, which commonly exist in C programs, can lead to inaccurate symbolic tracing which eventually lead to the path constraint being inaccurate. Such prevalent patterns lead to inaccurate path constraint and thereby contribute to the path divergence.

To diagnose the root causes of path divergence, CREST [21] conducted an empirical study and identified eight divergent patterns. Among these, the three most prevalent patterns account for 82% of total path divergence: exceptions, external calls, and type casts.

Specifically, when exceptions occur, execution is redirected to related exception-handling routines, leading to unexpected paths. When the program makes an external call that the concolic executor cannot symbolically trace, the associated constraints are omitted from the generated path constraints. Type casts, common in C programs, can lead to inaccurate symbolic tracing, resulting in incorrect path constraints. These prevalent patterns contribute to inaccurate path constraints and, consequently, to path divergence.

5.2 System Overview

We propose a path-corrective mutator that uses specialized prompts to instruct an LLM-based mutator to produce new testcases that traverse the target branch that concolic execution failed to explore due to path divergence.

The intuition is that each path divergence seed stems from an incomplete path constraint. Given a set of true seeds that successfully visited the target branch, we can identify their shared patterns, which likely represent the missing constraints in the incomplete path constraint. These patterns can then be leveraged to generate a new mutation that corrects the path divergence. Using the set of true seeds and the path divergence seed, a pattern identifier detects partial solutions from both, producing a *seed template* that incorporates key elements from each. This seed template is then passed to a prompt crafter, which generates a specialized prompt to instruct an LLM mutator in creating a corrected seed, ultimately resolving the path divergence for a concolic executor.

The path-corrective mutator can be integrated into a whitebox fuzzing framework capable of detecting path divergence and providing the associated target branch along with

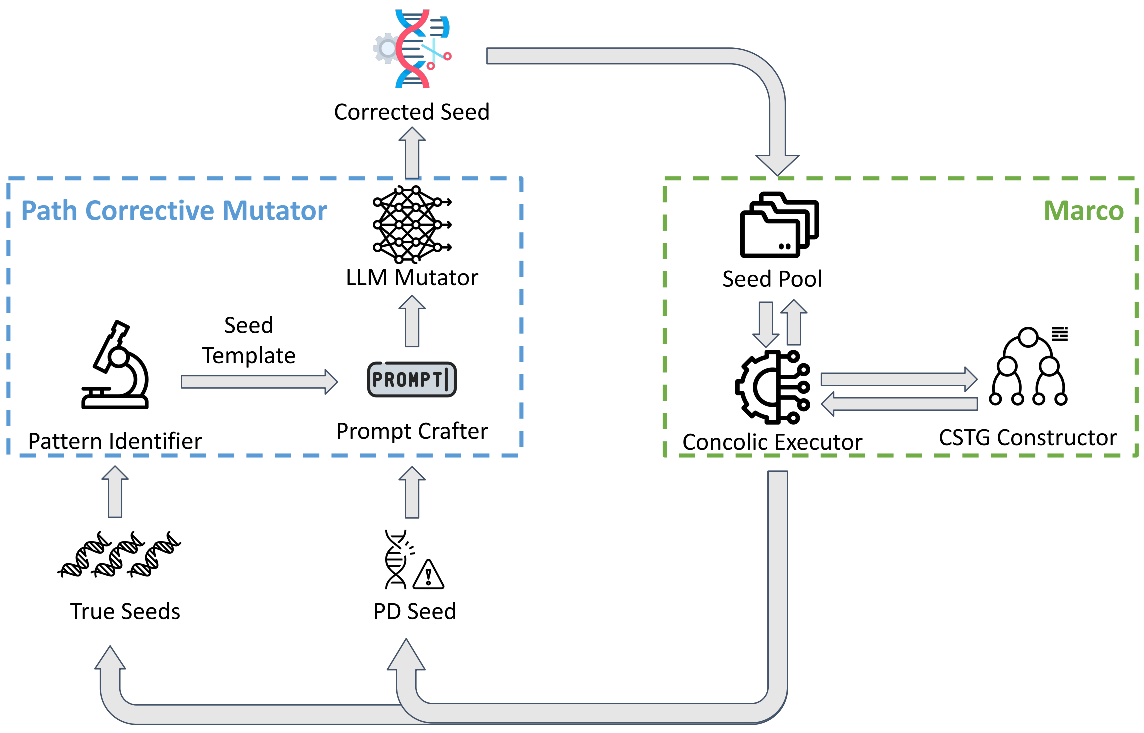


Figure 5.2: Design Overview

any true seeds that successfully visited this branch. Using these inputs, the path-corrective mutator generates corrected seeds, which are then supplied to the concolic executor as additional seeds for testing the program under test (PUT). With MARCO's Concolic State Transition Graph (CSTG), we can easily access each branch's set of executing seeds and identify path divergence when it occurs. Consequently, we integrated our proposed mutator with MARCO to test its effectiveness, resulting in the prototype system, POLARIS. It is important to note that this mutator can be incorporated into any whitebox fuzzer capable of detecting path divergence and providing a list of true seeds for the target branch.

5.2.1 Motivating Case Study

We present a successful path correction case to demonstrate how pattern identification and prompt crafting function in our proposed mutator, offering a motivating example of effectively correcting path divergence.

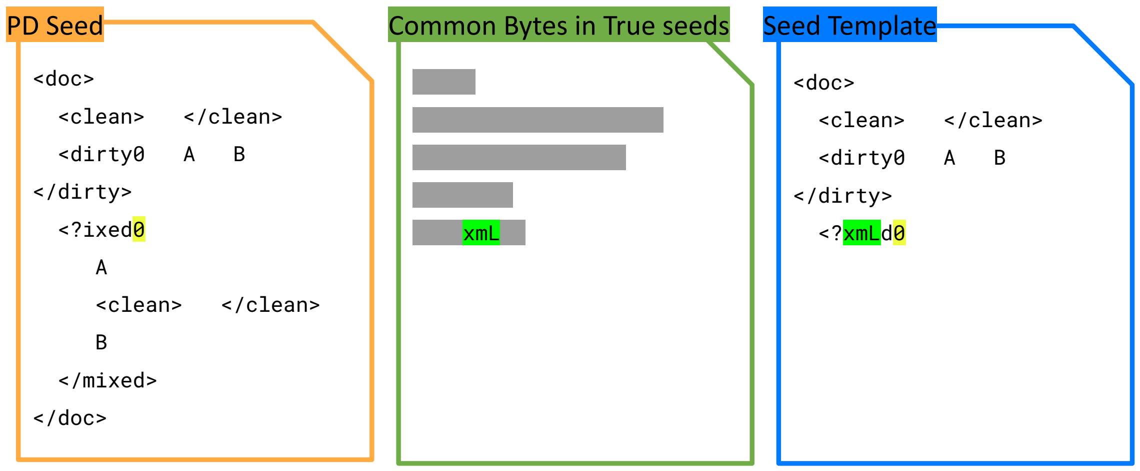
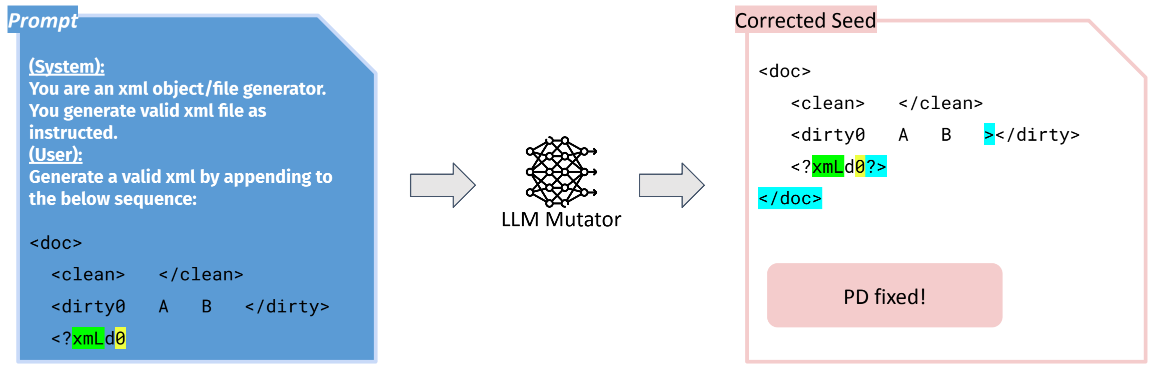


Figure 5.3: Build Seed Template

At the start of path corrective mutation, Marco identifies a path divergence from the PD seeds, as shown in Figure 5.3, along with a set of true seeds that successfully traversed the target branch missed by the PD seed. The pattern identifier first detects the solution that the CE engine used during path constraint solving to generate the PD seed (highlighted in yellow) and then identifies the common bytes shared by the set of true seeds (highlighted in green). These bytes represent a potential solution to the true path constraint set of the target branch. The pattern identifier then overwrites the PD seed with the extracted common bytes and trims any trailing bytes from the PD seed that do not contain partial solutions, constructing the seed template, as is shown in Figure 5.3. This seed template is an optimized speculation of the true seed capable of successfully visiting the target branch.


Prompt
(System):
You are an xml object/file generator. You generate valid xml file as instructed.
(User):
Generate a valid xml by appending to the below sequence:
LLM Mutator
Corrected Seed
PD fixed!

Prompt:

```
<doc>
  <clean>   </clean>
  <dirty0   A   B   </dirty>
  <?xmLd0
```

Corrected Seed:

```
<doc>
   <clean>   </clean>
   <dirty0   A   B   ></dirty>
   <?xmLd0?>
</doc>
```

Figure 5.4: Prompt LLM Mutator to Generate Corrected Seed

Using the seed template, the prompt crafter generates a specialized prompt to guide the LLM mutator in creating a new mutation that corrects the path. As shown in Figure 5.4, the LLM mutator is instructed to function as an XML object generator, appending a sequence of characters to the end of the seed template with the goal of constructing a syntactically valid XML object. The intuition behind this approach is that the new mutation will retain the optimized speculation of the true seed while ensuring valid XML syntax to avoid early rejection before reaching the target branch, thereby increasing the likelihood of successful path correction. Evaluation indicates that the corrected seed presented in Figure 5.4 successfully resolved the path divergence of the PD seed.

5.3 Preliminary Results

We evaluate Polaris against Marco on the program `libxml2` to compare the coverage achieved under the same seed generation budget (8300) and to assess the path correction rate of Polaris.

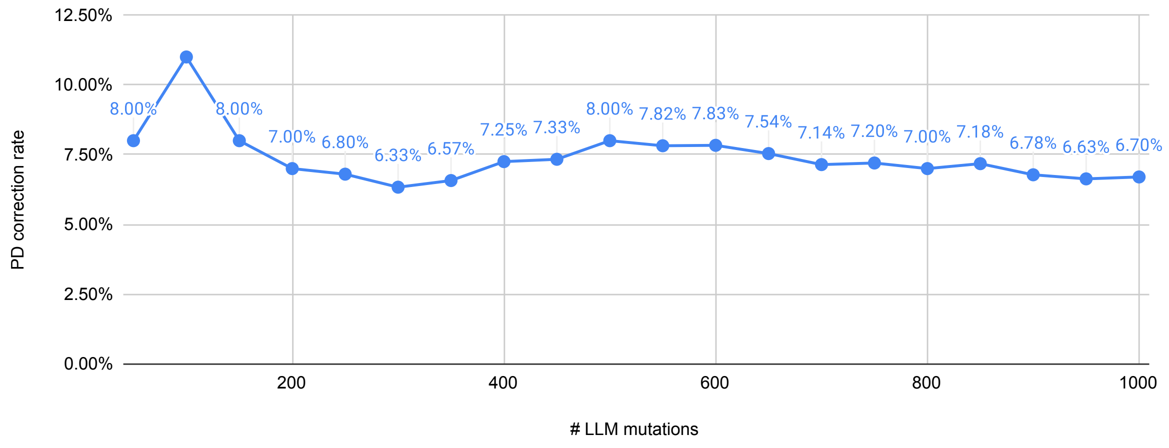


Figure 5.5: Path Correction Ratio

The results show that Marco corrects approximately 7.5% of path divergence occurrences in concolic testing, as demonstrated in Figure 5.5, leading to an 8.3% improvement in coverage, measured by node count in the CSTG, as shown in Figure 5.6. These results demonstrate the potential and highlight the significant opportunity for enhancing the soundness and completeness of concolic testing through the adoption of the path-corrective mutator.

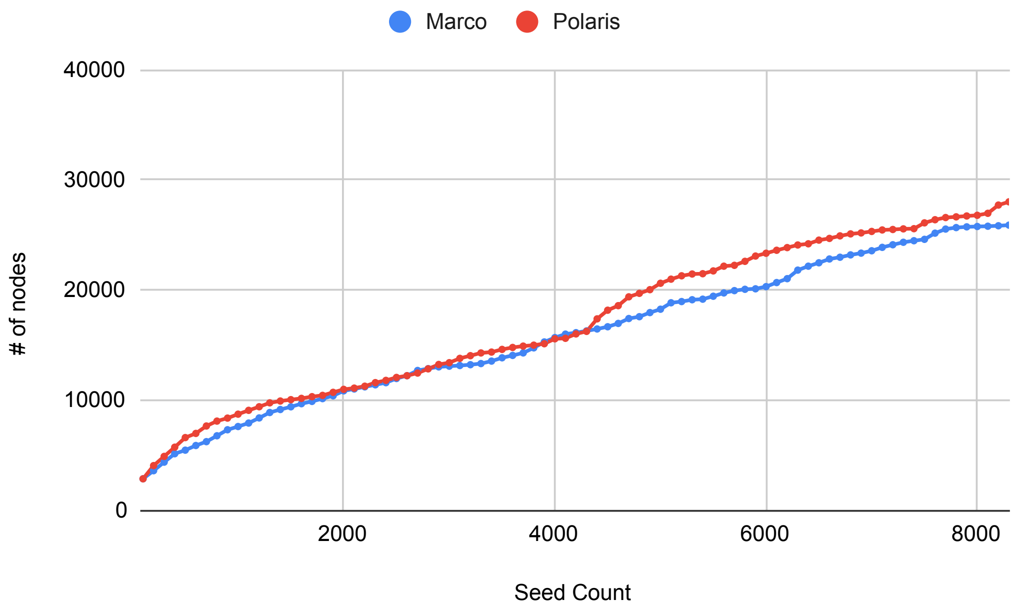


Figure 5.6: Coverage Efficiency

5.4 Discussion

Path divergence remains an open challenge in whitebox fuzzing, impacting the soundness and completeness of testing. Addressing this issue could make a significant contribution to the field and enhance concolic testing performance. The successful path correction case and preliminary results show that the LLM mutator can generate new mutations for path correction, improving the soundness and completeness of concolic testing. While the current results are not yet optimal, we are confident in the potential for future improvements. The key challenge now is to categorize path divergence by its root causes and determine how the LLM can contribute meaningfully to resolving this issue. This will be a focus of our future research.

Chapter 6

Conclusions

In this thesis, we highlight three major limitations in fuzzing—state explosion, random mutation effectiveness, and path divergence—that impair overall performance. We present three contributions aimed at addressing these challenges and enhancing fuzzing efficiency.

First, to tackle the state explosion problem, we introduce a stochastic modeling of the concolic trace as a Markov Chain and propose a novel reachability metric for more accurately assessing each branch's potential for new coverage. We adopt a reinforcement learning-based approach and integrate this scheduling scheme into a whitebox fuzzing framework, introducing the prototype Marco. We evaluate Marco against state-of-the-art concolic executors and greybox fuzzers, demonstrating its superior performance in coverage discovery and vulnerability detection.

Next, we conduct a systematic study of the fuzzing capabilities of out-of-the-box large language models (LLMs). We present insights from the experimental results and offer guidelines for constructing optimized GenAI-augmented mutators for fuzzing. We integrate our proposed mutator into a greybox fuzzer, presenting the prototype ChatFuzz. Evaluations of ChatFuzz show enhanced performance in coverage discovery and vulnerability detection compared to baseline approaches.

Lastly, we identify an important research direction addressing the impact of path divergence on the soundness and completeness of fuzzing, with over 60% of path constraint solutions potentially leading to divergence. To mitigate this issue, we propose a path-corrective solution using an LLM-based mutator, which we integrate with the Marco framework, introducing the prototype Polaris. Preliminary studies demonstrate significant potential and room for improvement in this area.

With these three contributions, we present an enhanced fuzzing framework featuring a *reachability-guided scheduler* and a *GenAI-augmented path-corrective mutator*.

Bibliography

[1] Mozilla security - dharma. `https://github.com/mozillasecurity/dharma`, 2020.

[2] Fuzzbench: Fuzzer benchmarking as a service. `https://google.github.io/fuzzbench`, 2022.

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

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 51(3):1–39, 2018.

[5] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code, 2022.

[6] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 678–689, 2020.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2329–2344, 2017.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1032–1043, 2016.

[9] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In 2018 IEEE Security and Privacy Workshops (SPW), pages 116–122. IEEE, 2018.

[10] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In 2013 35th International Conference on Software Engineering (ICSE), pages 122–131. IEEE, 2013.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[12] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pages 443–446. IEEE, 2008.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In 8th USENIX Symposium on Operating Systems Design and Implementation, volume 8, pages 209–224, 2008.

[14] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC), 12(2):1–38, 2008.

[15] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. Communications of the ACM, 56(2):82–90, 2013.

[16] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP '15, page 725–741, USA, 2015. IEEE Computer Society.

[17] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. arXiv preprint arXiv:2007.15943, pages 2325–2342, 2020.

[18] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. Symsan: Time and space efficient concolic execution via dynamic data-flow analysis. In 31st USENIX Security Symposium (USENIX Security 22), pages 2531–2548, 2022.

[19] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In 2022 IEEE Symposium on Security and Privacy (SP), pages 1531–1531. IEEE Computer Society, 2022.

[20] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In 2018 IEEE Symposium on Security and Privacy (SP), pages 711–725. IEEE, 2018.

[21] Ting Chen, Xiaodong Lin, Jin Huang, Abel Bacchus, and Xiaosong Zhang. An empirical investigation into path divergences for concolic execution using crest. Security and Communication Networks, 8(18):3667–3681, 2015.

[22] Liang Cheng, Yang Zhang, Yi Zhang, Chen Wu, Zhangtan Li, Yu Fu, and Haisheng Li. Optimizing seed inputs in fuzzing with machine learning. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pages 244–245. IEEE, 2019.

[23] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. Acm Sigplan Notices, 46(3):265–278, 2011.

[24] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. arXiv preprint arXiv:2304.02014, 2023.

[25] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In 2016 IEEE symposium on security and privacy (SP), pages 110–121. IEEE, 2016.

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ combining incremental steps of fuzzing research. In Proceedings of the 14th USENIX Conference on Offensive Technologies, pages 10–10, 2020.

[27] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In 2018 IEEE Symposium on Security and Privacy (SP), pages 679–696. IEEE, 2018.

[28] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In Network and Distributed System Security Symposium, NDSS, volume 8, 2008.

[29] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 50–59. IEEE, 2017.

[30] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Inferring input grammars from dynamic control flow, 2019.

[31] Rahul Gopinath and Andreas Zeller. Building fast fuzzers, 2019.

[32] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In NDSS, 2019.

[33] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In Network and Distributed Systems Security (NDSS) Symposium 2018, 2018.

[34] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. Learning to explore paths for symbolic execution. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 2526–2540, 2021.

[35] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 230–243, 2021.

[36] Jie Hu, Yue Duan, and Heng Yin. Marco: A stochastic asynchronous concolic explorer. In Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pages 1–12, 2024.

[37] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In Proceedings of the 44th International Conference on Software Engineering, ICSE '22, page 1219–1231, New York, NY, USA, 2022. Association for Computing Machinery.

[38] Ismet Burak Kadron, Yannic Noller, Rohan Padhye, Tevfik Bultan, Corina S Păsăreanu, and Koushik Sen. Fuzzing, symbolic execution, and expert guidance for better testing. IEEE Software, 2023.

[39] Xuan-Bach D. Le, Corina S. Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. ACM SIGSOFT Software Engineering Notes, 44(4):14, 2019.

[40] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In International conference on software engineering (ICSE), 2023.

[41] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 475–485, 2018.

[42] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In USENIX Security Symposium, pages 2777–2794, 2021.

[43] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In 2022 IEEE Symposium on Security and Privacy (SP), pages 1–17. IEEE, 2022.

[44] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin IP Rubinstein. Legion: Best-first concolic testing. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pages 54–65, 2020.

[45] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. Mopt: Optimized mutation scheduling for fuzzers. In USENIX Security Symposium, pages 1949–1966, 2019.

[46] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS), 2024.

[47] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In Proceedings of the 29th USENIX Conference on Security Symposium, pages 2289–2306, 2020.

[48] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 329–340, 2019.

[49] Emanuel Parzen. On estimation of a probability density function and mode. The annals of mathematical statistics, 33(3):1065–1076, 1962.

[50] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In 2018 IEEE Symposium on Security and Privacy (SP), pages 697–710. IEEE, 2018.

[51] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In 29th USENIX Security Symposium (USENIX Security 20), pages 181–198, 2020.

[52] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In Network and Distributed System Security Symposium, NDSS, 2021.

[53] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In NDSS, volume 17, pages 1–14, 2017.

[54] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In 23rd {USENIX} Security Symposium ({USENIX} Security 14), pages 861–875, 2014.

[55] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. A tutorial on thompson sampling. Foundations and Trends® in Machine Learning, 11(1):1–96, 2018.

[56] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-Level Fuzzing. In 30th USENIX Security Symposium (USENIX Security 21), 2021.

[57] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. Mtfuzz: Fuzzing with a multi-task neural network. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, page 737–749, New York, NY, USA, 2020. Association for Computing Machinery.

[58] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In 2019 IEEE Symposium on Security and Privacy (SP), pages 803–817. IEEE, 2019.

[59] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. arXiv preprint arXiv:2203.12064, pages 2194–2211, 2022.

[60] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 347–362, 2017.

[61] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In Proceedings of the 30th acm sigsoft international symposium on software testing and analysis, pages 244–256, 2021.

[62] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 01 2016.

[63] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2021.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[65] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. 2021.

[66] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In 2017 IEEE Symposium on Security and Privacy (SP), pages 579–594. IEEE, 2017.

[67] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In Proceedings of the 41st International Conference on Software Engineering, ICSE '19, page 724–735. IEEE Press, 2019.

[68] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. PloS one, 15(8):e0237749, 2020.

[69] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. Evaluating and improving neural program-smoothing-based fuzzing. In Proceedings of the 44th International Conference on Software Engineering, pages 847–858, 2022.

[70] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. Evaluating and improving neural program-smoothing-based fuzzing. In 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pages 847–858, 2022.

[71] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. arXiv preprint arXiv:2308.04748, 2023.

[72] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool. arXiv preprint arXiv:2305.04764, 2023.

[73] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2313–2328, 2017.

[74] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In 29th USENIX Security Symposium (USENIX Security 20), pages 2307–2324, 2020.

[75] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In Proceedings of the 27th USENIX Security Symposium (Security), pages 745–761, Baltimore, MD, August 2018.

[76] M. Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`.

[77] G Zhang, P Wang, T Yue, X Kong, S Huang, X Zhou, and K Lu. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In Network and Distributed Systems Security (NDSS) Symposium, volume 2022, 2022.

[78] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 2169–2182, 2021.