Poster: gptCombFuzz: Combinatorial Oriented LLM Seed Generation for effective Fuzzing

Darshan Lohiya, Monika Rani Golla, Sangharatna Godboley, P. Radha Krishna
NITMiner Technologies, Department of Computer Science and Engineering
National Institute of Technology, Warangal, Warangal, Telangana, India
{da22csm1r09,gm720080}@student.nitw.ac.in, {sanghu, prkrishna}@nitw.ac.in

2024 IEEE Conference on Software Testing, Verification and Validation (ICST) | 979-8-3503-0818-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICST60714.2024.00048

***Abstract*—The important contribution that large language models (LLMs) have made to the development of a new software testing era is the main objective of this proposed approach. It emphasizes the role that LLMs play in producing complex and diverse input seeds, which opens the way for efficient bug discovery. In the study we also introduce a systematic approach for combining various input values, employing the principles of Combinatorial testing using the PICT (Pairwise independent Combinatorial testing). By promoting a more varied set of inputs for thorough testing, PICT enhances the seed production process. Then we show how these different seeds may be easily included in the American Fuzzy Lop (AFL) tool, demonstrating how AFL can effectively use them to find and detect software flaws. This integrated technique offers a powerful yet straightforward approach to software Quality.**

***Index Terms*—Large Language Model, AFL, Pairwise independent combinatorial testing**

I. Introduction

Software testing is a critical aspect of ensuring the reliability and security of applications in the rapidly changing field of technology. Among the diverse approaches to software testing, Fuzz testing, often referred to as fuzzing, is a dynamic testing method where vulnerabilities are found by subjecting a software program to a flood of different inputs. Fuzz testing was designed to investigate unexpected ways in the application's execution, in contrast to typical testing techniques that follow established scenarios. The purpose of this input randomness is to replicate real-world situations where unexpected data could be found.

Fuzz testing [1] is effective because it can find software flaws such as memory leaks, security vulnerabilities, and other issues that traditional testing could miss. Fuzzers [2] provide a lot of test inputs, and crashes may be found by observing how the program behaves in response to these inputs. When a crash occurs in the target program, the fuzzing tool quickly saves the triggering input file. This proactive approach allows the tool to quickly detect and categorize crashes, ensuring a successful compilation of input data related to these key occurrences. American Fuzzy Lop (AFL) [3] is a popular and effective fuzz testing tool that is prominent for its creative method of locating software problems. The combination of a genetic algorithm with a fuzzy logic mutation method distinguishes AFL. AFL's fuzzer technique is more clever than that of traditional fuzzers, which usually depend on random mutations.

Initially given a set of seed inputs, AFL uses genetic algorithms to alter and develop these inputs to find new routes inside the application. The fuzzer [2] prioritizes inputs that lead to untested code paths while keeping track of code coverage. The most important initial phase in the fuzz testing procedure is seed generation. It involves creating the initial set of inputs, or "seeds," that the fuzzer will use to start the process of testing. The success of the fuzzing process is directly impacted by the quality and diversity of these seeds.

The program's initial seeds can be manually constructed to reflect valid inputs. However, it is essential to expand and improve the seed set to accomplish a thorough investigation of the application. Large Language Models [4], such as GPT-3 [5], have shown remarkable ability in creating and understanding real language. A new perspective to the process is added when these capabilities are used for fuzz testing seed generation. Unlike manually generated seeds, which could miss some real-world data conditions, LLM can automatically generate a wide variety of complex and diversified input seeds.

By adding complexity and variety to the inputs, the use of LLM in seed generation enables fuzzers such as AFL to investigate a wider range of code paths. The seeds that are created contain odd combinations, edge situations, and possible inputs that would be missed by a human method. This increases the efficiency of the fuzz testing procedure overall and increases the chances of finding hidden vulnerabilities.

Combinatorial testing [6], a systematic technique for software testing, seeks to effectively cover several permutations of input parameters to find potential defects or weaknesses in a program. It tackles the difficulty of testing a large input space by picking a subset of test cases that guarantees coverage of all possible pairwise combinations of input values. By choosing combinations that encompass several pairings of parameters, this strategy can help in improving the variety of inputs in the context of fuzz testing

Fuzzing Process and paired combinatorial testing together provide an effective combination. With the varied seeds produced by LLMs and paired pairings, Fuzzer may effectively explore complex code routes. Combining these techniques guarantees a deeper investigation of the behavior of the application, improving the effectiveness and attention of the fuzz testing procedure.

The rest of the paper is organized as follows: Section 2 presents the related work. Our work is discussed in Section

979-8-3503-0818-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ICST60714.2024.00048

3, and the experimental study is in Section 4. Finally, we conclude the work in Section 5.

II. Related Work

There are seed generation techniques that use several different for effective generation. Skyfire1 [7] shows a cutting-edge data-driven seed generation approach aimed to improve the fuzzing of algorithms that deal with highly structured inputs. The model distinguishes itself by autonomously collecting grammatical and semantic rules from a large sample corpus and applying this knowledge to generate evenly dispersed seed inputs. Furthermore, Skyfire1 [7] broadens the capabilities of previous generation-based fuzzing approaches by going beyond semantic verification and investigating program execution phases. The technique begins with parsing the corpus into abstract syntax trees using a probabilistic context-sensitive grammar, followed by repeated seed input creation and mutation for variety.

SmartSeed [8] is a machine learning-based solution, that conducts a three-step approach. First, it collects training data by fuzzing applications using tools such as AFL, focusing on files that cause novel crashes or expose previously unexplored routes. This first data gathering is critical for the successful launch of this method. Second, It uses a conversion mechanism to convert the training data into generic matrices, resulting in a generative model for seed generation. This enables the model's applicability to a variety of applications. Finally, using the created seeds, It uses fuzzing tools to detect crashes in target programs, resulting in a closed-loop architecture. The machine learning model is constantly refining itself by including fresh crashes and pathways, making the seed generation process dynamic and efficient.

Cheng et al. [9] provide a machine learning-based strategy for improving seed inputs in fuzzing. This system makes use of the relationship between seed inputs and program execution to generate additional inputs, increasing code coverage and improving bug/crash detection. Using a generative model based on recurrent neural networks (RNNs), we generate novel execution pathways for the target program outside of the initial seed corpus. These pathways are translated into acceptable PDF files using a sequence-to-sequence (Seq2seq) transition approach, which generates fresh seed inputs. Both models are trained with the original seed inputs and associated target program executions. When compared to other models such as ARIMA and CNN, RNNs outperform them in learning lengthy sequences with specific syntactic patterns.

III. Proposed Approach

In this section, we proposed our work **gptCombFuzz** a seed generation technique that use large language model to generate diverse seeds values and combinatorial testing for more code coverage the framework is also covered in this section

A. Overview

Figure 1 shows our proposed framework of gptCombFuzz. In our proposed method, we forgo traditional seed generation strategies and instead rely on the immense capabilities of Large Language Models (LLMs), notably the powerful GPT-3.5 [5]. Unlike conventional seed production, which is random, LLMs [4] use their enormous language learnings to create seeds that are not just diverse but also highly complex. This deviation from randomness is critical, especially when dealing with applications of greater complexity.

Our method begins with providing the program together with a guiding prompt [10], which is a set of instructions for the LLM. This prompt acts as critical guidance for the LLM, directing it to generate seeds that are perfectly aligned with the complexities of the provided program. It communicates the desired focus and expectations to the language model. The combination of program and prompt is given as input to the LLM which provides a diverse collection of seeds that contains the seed's value based on the random input, valid and invalid input as well as the Edges cases. This seed ensemble is further given as input to the paired-wise combinatorial testing. By systematically combining variable values, we ensure a thorough investigation of potential interactions, resulting in a diverse collection of seed values. Then these seed values are passed through a fuzzer to detect the bugs.

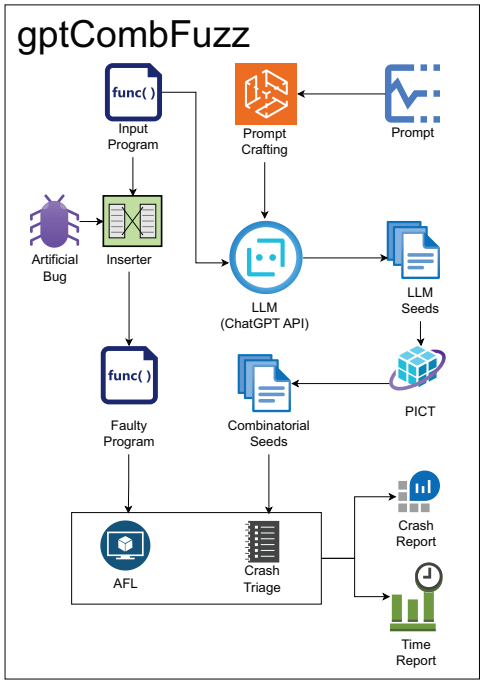

Fig. 1. Framework of gptCombFuzz.

B. LLM-Seed Generation

We perform a thorough assessment of the program being evaluated before entrusting the LLM with seed generation. To do this, one must examine the program's structure, identify important variables and potential points of failure, and understand the complex dependencies that connect the components together. Consider of this as an examination of the program's

internal mechanisms, exposing them for the LLM to better understand.

With this complete understanding, we construct a prompt [10] that serves as the LLM's center of attention. This prompt corresponds to an elaborate unique, constructed with care with all of the program's the complexity, testing goals, and expected outcomes in view. In order to prevent misunderstandings by the LLM, we carefully consider the phrasing and organization, prioritizing precision and clarity. This is where prompt engineering becomes useful, since the prompt's quality directly affects how well the seeds are created.

```
-----------------------------------------------
def generate_seed_generation_prompts():
    prompt_text = (
         "Generate the combination of seed values(
    test case)for scanf variable (give the values
    for them)"
         "with respect to Random Input,Boundary
    Values,valid Input,Invalid Input,Edges cases in
    the json"
         "format generate at least seven testcase
    values and it should cover all the code path"
    )

    length = len(prompt_text)
    response = openai.Completion.create(
        engine="text-davinci-003",
        prompt=prompt_text,
        max_tokens=length, # Prompt length
        n=2,  # Generate 2 different prompts
        stop=None,
        temperature=1.0,
        top_p=0.9
    )
    generated_prompts = [choice['text'].strip() for
    choice in response['choices']]
    return generated_prompts
-----------------------------------------------
```

Listing 1. How LLM process the query?

The Lisitng 1 specifies how we can query our prompt to chatgpt model to generate the diverse seed values prompt. The prompt is stored in the 'prompt_text' variable. In prompt we mention that it should consider the scenarios which includes the case such random input, Boundary values, valid and invalid input and Edges cases. We also specify that the generated output should be in JSON format as we need to parse this program to a proper format to give this input file in the PICT tool. The 'max tokens' parameter is to ensure that the generated responses remain within a reasonable token limit. The parameters 'temperature' and 'top_p' play a crucial roles in influencing the diversity and creativity of the generated responses. Temperature parameter controls the randomness of the generated output. In our experiment, we have choosen the temperature value of 1.0 which leads to random and more diverse responses to cover more cases. The 'top_p parameter sets the cumulative probability threshold for including in sampling process for our experiment it is set to 0.9. The engine parameter specifies which version of GPT-3 model we are using for our experiment we are using the 'text-davinci-003. The stop parameter is used to specify a stopping condition for the generated response.

C. Pairwise independent Combinatorial testing

The generated seeds are then subjected to further improvement using the Pairwise Independent Combinatorial Testing (PICT) tool, which is well-known for its capacity to methodically produce a wide range of input combinations. With its ability to effectively investigate the interactions between different input parameters, PICT makes it possible to investigate program behavior in more detail. This stage contributes to a more thorough and efficient fuzzing process by increasing the diversity and coverage of the seed values.

D. American Fuzzy lop

When our model, gptCombFuzz, customizes seeds according to the specifics of a given program, the focus shifts to making sure these seeds cover the program's code in all of its parts. An important part of this procedure is the usage of AFL, a popular fuzz testing tool. AFL thoroughly examines the faulty software that has been injected with PICT-enhanced variants and LLM-generated seeds.

The AFL serves two purposes. First, it assesses the seeds' capacity to navigate complex code pathways, ensuring a full investigation of the program's behavior. This phase is critical for detecting bugs that may remain hidden with traditional seed sets. Second, AFL acts as an inspection, relentlessly searching for any faults in the software. The dynamic interplay between the various seeds and AFL's intelligent testing procedures results in a mutual connection, which improves the overall bug discovery process. The bugs which are detected during the fuzzing are analyzed by the crash-triage module.

IV. Experimental Study

In this section, we discuss the setup and results analysis.

A. Setup

In our experiment, we have considered the PALS programs[1], with their diverse variables, serve as the testing ground. Utilizing GPT-3.5[2], we query prompts through our OpenAI account, allowing the model to generate seeds with a nuanced understanding of PALS. Pairwise independent combinatorial testing (PICT)[3] systematically explores input combinations, ensuring comprehensive variable interactions. To inject unpredictability, we employ the American Fuzzy Lop (AFL)[4] as our fuzzer. This cohesive approach, combining GPT-3.5 for seed generation, pairwise testing for coverage, and AFL for dynamic exploration, enhances the effectiveness of our fuzz testing on the PALS program.

B. Results

We performed experimentation on 45 programs by injecting a single bug, and checking which tool detects it faster. Fig. 2 shows the total execution time for both approaches. The

[1]https://github.com/tracer-x/PROGRAM-DATABASE-TEACHING/tree/master/AFL-VERSION/PALS
[2]https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates
[3]https://github.com/microsoft/pict
[4]https://github.com/google/AFL

gptCombFuzz is represented by red-crosses winning in all the programs as compared to Random-AFL. Fig. 3 shows the paths found. Our tool gptCombFuzz has a total 32 winning cases (red-crosses) out of 45 programs. Fig. 4 shows the total execution done, where gptCombFuzz takes fewer executions as compared to Random-AFL.

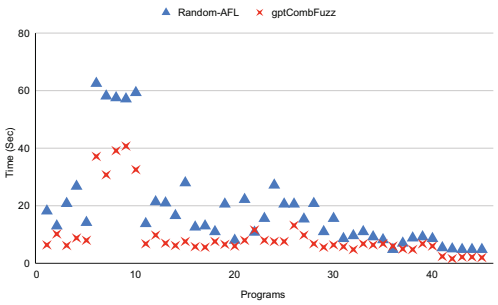

Fig. 2. Random-AFL vs. gptCombFuzz wrt. Time.

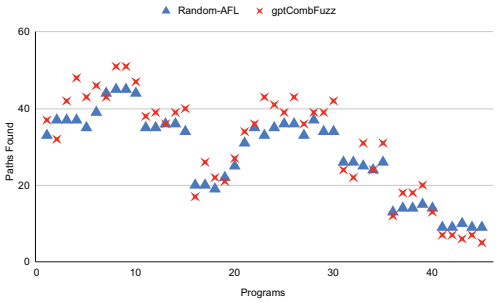

Fig. 3. Random-AFL vs. gptCombFuzz wrt. Paths Found

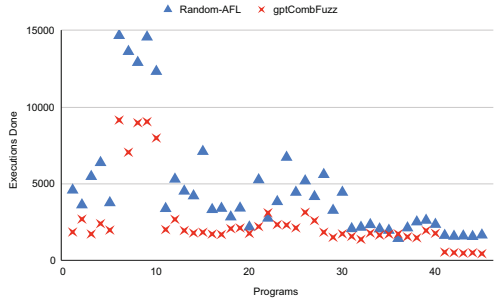

Fig. 4. Random-AFL vs. gptCombFuzz wrt. Executions Done

V. Conclusion

In this study, we focus on the new seed generation strategy proposed in our model, which takes advantage of the capabilities of large language models (LLMs) to generate meaningful and varied seeds. The use of LLMs, specifically GPT-3.5 Turbo, improves the contextual significance of produced seeds, allowing for a more complete analysis of program code pathways. Concurrently, our investigation employs combinatorial testing, notably the Pairwise Independent Combinatorial Testing (PICT) tool. This method methodically mixes several seeds to optimize code coverage effectively. The comparison of our proposed strategy to the usual random seed generation method yielded impressive results. Our approach provided faster bug discovery, highlighting the need to use complex seed generation strategies for effective software testing. As the size of programs grows, finding a larger number of code pathways becomes critical for rapid problem discovery. In this regard, our technique improves the random seed strategy and provides an effective means for speeding up the bug-finding process. The combined capabilities of LLM-driven seed generation and PICT result in a comprehensive and efficient software testing approach that has the potential to improve software system dependability and security. The evaluation needs to include several repetitions to ensure the results have enough statistical power, which we will do in the future.

Acknowledgement

This work is sponsored by IBITF, Indian Institute of Technology (IIT) Bhilai, under the grant of PRAYAS scheme, DST, Government of India.

References

[1] Klees, George & Ruef, Andrew & Cooper, Benji & Wei, Shiyi & Hicks, Michael. (2018). Evaluating Fuzz Testing. 2123-2138. 10.1145/3243734.3243804.

[2] Boehme, Marcel & Cadar, Cristian & Roychoudhury, Abhik. (2020). Fuzzing: Challenges and Reflections. IEEE Software. PP. 10.1109/MS.2020.3016773.

[3] American Fuzzy Lop. Available online: https://lcamtuf.coredump.cx/afl/ (accessed on 1 September 2020)

[4] Zhao, Wayne Xin, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Z. Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jianyun Nie and Ji-rong Wen. "A Survey of Large Language Models." ArXiv abs/2303.18223 (2023): n. pag.

[5] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T.J., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models are Few-Shot Learners. ArXiv, abs/2005.14165.

[6] Bryce, Renée & Lei, yu & Kuhn, D. & Kacker, Raghu. (2009). Combinatorial Testing. Advances in Computers. 99. 10.4018/978-1-60566-731-7.ch014.

[7] J. Wang, B. Chen, L. Wei and Y. Liu, "Skyfire: Data-Driven Seed Generation for Fuzzing," 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 2017, pp. 579-594, doi: 10.1109/SP.2017.23.

[8] Lv, Chenyang, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, Pan Zhou and Jing Chen. "SmartSeed: Smart Seed Generation for Efficient Fuzzing." ArXiv abs/1807.02606 (2018): n. pag.

[9] L. Cheng et al., "Optimizing Seed Inputs in Fuzzing with Machine Learning," 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Montreal, QC, Canada, 2019, pp. 244-245, doi: 10.1109/ICSE-Companion.2019.00096.

[10] Chen, Banghao, Zhaofeng Zhang, Nicolas Langren'e and Shengxin Zhu. "Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review." ArXiv abs/2310.14735 (2023): n. pag.